BSTC: A Novel Binarized-Soft-Tensor-Core Design for Accelerating Bit-Based Approximated Neural Nets

Anonymous Author(s)

ABSTRACT

Binarized neural network (or BNN) promises tremendous performance improvement over traditional DNNs through simplified bit-level computation and significantly reduced memory access/storage cost. In addition, it has other advantages of low-cost, low-energy and high-robustness, showing great utilization potential in resources-constrained, volatile and latency-critical applications, which are critical for future HPC and embedded execution. However, the promised significant performance gain of BNN inference has never been fully demonstrated on general-purpose processors, particularly on GPUs, due to: (i) the challenge to extract and leverage sufficient fine-grained bit-level-parallelism to saturate GPU cores when batch is small; (ii) the fundamental design conflict between bit-based BNN algorithm and underlying word-based architecture; (iii) architecture & performance unfriendly BNN network design. To address (i) and (ii), we propose binarized-soft-tensor-core as a software-hardware codesign approach to construct bit-manipulation capability for modern GPUs to effectively harvest the emerging bit-level-parallelism. To tackle (iii), we propose intra- and inter-layer fusion techniques so that the entire BNN inference can be packed into a single GPU kernel, to avoid high-cost frequent launching. Experiments demonstrate that our design can achieve over 1000x speedup for raw inference latency and 10x for inference throughput over the state-of-the-art full-precision simulated BNN inference for AlexNet on ImageNet.

1 INTRODUCTION

Binarized-neural-networks (BNNs) evolve from the conventional binarized-weight-networks (BWNs) [11, 20, 39], which was first observed that if weights could be binarized into ±1, floating-point mul could be degraded to add (mul +1) and sub (mul -1). Then, it was observed that if both the weights and inputs can be binarized, the floating-point add and sub can be further degraded to logical bit operations (i.e., xnor and popc), known as BNN [12, 20, 39].

BNNs have several advantages over conventional DNNs: (I) *Low computation demand.* Each 32 floating-point or integer mul-add operations can be aggregated into a single popc-xnor operation, significantly reducing latency and hardware design complexity. (II) *Low memory demand.* The entire memory hierarchy (e.g., registers, cache, scratchpad, main memory, etc) may sustain 32x or more storage and bandwidth compared to full-precision DNNs. (III) *Low energy consumption.* Due to reduced hardware complexity and smaller chip-area, BNN-based devices are much more energy efficient. (IV) *Security.* BNNs recently show improved robustness against adversarial attacks than normal DNNs [16] and can be exact encoded for formal verification and property analysis [9, 34].

The primary concern about BNN is accuracy, which is becoming a popular topic for DNN algorithm research. Recently, with advanced training techniques being introduced [2, 3, 20, 42], BNN accuracy has been improved significantly. The top-5/1 training accuracy for BNN-based AlexNet on ImageNet has been enhanced from 50.4/27.9% [12, 42] to 75.7/46.1% [2] within two years, with respect to 80.2/56.6% [2] from the full-precision design. Although BNN's accuracy is still a bit inferior, it shows great advantages in almost all the other computation aspects, e.g., low-cost, low-latency, low-energy and high robustness. BNN is unlikely to replace full-precision DNN strategies, but for many practical utilization such as smart edge and mobile phones, recommendation systems, auto-driving and other resource-constrained environments, when certain accuracy bar is surpassed, other essential metrics such as real-time bound and cost become more significant. BNNs can provide attractive solutions for these special domains [25, 33]. There has also been an effort to map BNN-like learning structures to quantum annealers for demonstrating quantum supremacy [4].

While most existing research either focus on improving BNN accuracy, or exploring the design space for ASIC/FPGA implementation, the promised tremendous gain on BNN inference performance (32x memory and 10x computation reduction) has barely been demonstrated on economy-of-scale general-purpose processors such as GPUs. This is due to the following reasons. First, it is challenging to extract sufficient fine-grained parallelism for small batch or non-batching cases, leading to extremely poor utilization of GPU resources (e.g., as low as 1% for BNN [35]). Second, there is a fundamental design conflict between bit-based algorithms (e.g., in BNNs) and the word-based general-purpose architectures such as CPUs and GPUs, which are traditionally designed and optimized for processing words or subwords and thus lack explicit bit-manipulation support. Third, current BNN network designs are mainly proposed from algorithm & accuracy perspectives, lacking contributions from architecture & performance aspects. Consequently, they are not particularly hardware-friendly and performance-oriented. For example, only convolution and fully-connect layers in BNN models are binarized; the rest layers such as batch-normalization and pooling all consists of expensive full-precision floating-point calculation and memory access. The challenges above lead to inefficient BNN design and implementation on economy-of-scale general-purpose accelerators, significantly reduces its adaptation for scenarios that require high real-time and resource constraints. Meanwhile, the majority of BNN algorithm research is conducted via full-precision simulations in a high-level programming environment (e.g., Tensorflow and PyTorch). While modern general-purpose many-core accelerators such as CPUs and GPUs provide great computation power, due to the poor hardware utilization and extreme real-time latency constraints for inference, conventional wisdom suggests that they are incapable of bringing significant speedups for BNN designs to be applied in the real-world production scenarios.

In this paper, we aim to tackle the three challenges above by proposing the *Singular-Binarized-Neural-Network* (SBNN) design. To systemically build the bit-manipulation capability for modern GPUs while keeping the hardware highly utilized in an efficient manner, we adopt a software-hardware co-design approach: a virtual bit-core named Binarized-Soft-Tensor-Core (BSTC) is constructed upon GPU's Streaming Multiprocessors (SMs), leveraging the native hardware instructions to obtain high efficiency while offering desired bit-manipulation APIs to the application layer for harvesting bit-level-parallelism (BLP) which have recently emerged

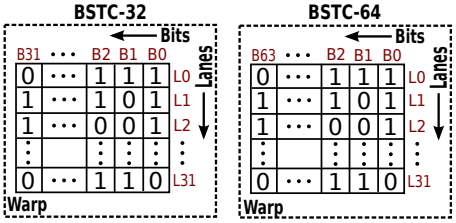

Figure 1: Binarized-Soft-Tensor-Core in 32 and 64 bits mode. Bits are indexed from right to left since in an operand, MSB is usually on the left, while LSB on the right.

as the key optimization target in various domain applications [1, 6, 7, 14, 15, 38, 40, 45]. To create a hardware-friendly and performance-oriented design, we propose several BNN network adjustment measures. For instance, the intra- and inter-layer fusion techniques, which can merge the entire BNN inference process into a single GPU kernel, can help achieve up to *6115x* latency reduction for MLP on MNIST, and *1429x* latency reduction for AlexNet on ImageNet, when comparing with full-precision simulated BNN in a high-level DNN environment (i.e., TensorFlow via cuDNN) running on an NVIDIA Tesla-P100 GPU. In this design, there is no warp-divergence, no shared memory bank conflicts, no non-coalesced global memory accesses, and no additional workspace requirement, which lead to excellent computation efficiency and system resource utilization. Because of the whole-network fusion, all GPU kernel invocation & release overheads from the conventional designs are eliminated. SBNN can complete non-batched AlexNet inference within 1ms. Finally, our proposed design is purely software-based without any external library dependency. It can be directly deployed on the existing HPC and embedded GPU platforms. There is no need to adjust or reconfigure every time the network or input size changes.

2 BINARIZED-SOFT-TENSOR-CORE

As shown in Figure 1, a binarized-soft-tensor-core (BSTC) is defined as *a 2D bit-tile with orthogonal dimensions corresponding to GPU warp lanes and bitwidth per lane.* A BSTC is executed purely via a single warp, with each lane contributing 32 bits or 64 bits, forming a 32x32 or 32x64 matrix; and each matrix element represents a binary state. For BSTC-32, the data type is unsigned int; for BSTC-64, it is unsigned long long int. BSTCs stay entirely in the register file.

BSTC design follows the recently proposed *warp-consolidation* programming model [29], which unifies a thread block and a warp: each thread block contains only a single warp – a BSTC here. This model shows six advantages over the traditional CUDA model: (i) no thread block level synchronization; (ii) independent on-chip resource allocation and release; (iii) simplified kernel configuration and design space; (iv) register-shuffle-based fast inter-lane communication; (v) flexible fine-grained thread cooperation, and (vi) extended register space. Both BSTC and SBNN exploit these features to boost performance.

2.1 BSTC Operations

BSTC relies on fine-grained, highly-efficient inter-lane and inter-bit communication and cooperation mechanisms to design the required functionalities and achieve high performance. Figure 2 illustrates the general primitives to operate a BSTC:

(A): AND, OR and XOR are executed on each bit of the data along the rows, so the communication pattern is *intra-row*. XOR is very useful for ±1 constructing bit-dot-product. OR and AND are used for bit-max and bit-min operations.

(B): __popc() and __popcll(), also known as population-count, return the number of bits whose binary value equals to 1 in a 32/64-bit data. This primitive offers fast accumulation along the SBTC rows. They are the key operations to map from BSTC's binary space to normal full-precision space. popc has widely adopted for aggregating bit-dot-product results.

(C): __brev() and __brevll() reverse the bit sequence of a 32/64 bit row. They offer the ability to fast rotating a bit-row for 180°. The communication pattern is intra-row. They are often used together with __ballot(), see (H).

(D): __any() and __all() are the warp voting operations. They are executed along bit-columns – merging 32 bits of a bit-column into a single bit, and broadcasting this bit to all the 32 thread lanes. __any() returns 1 if any entry of the column is 1 while __all() returns 1 when all of them are 1s.

(E): __shfl() is to exchange bit-rows in a BSTC. Shuffle has four variants: __shfl() performs flexible general bit-row exchanging function. __shfl_up() and __shfl_down() are to rotate bit-rows up and down by a certain interval. __shfl_xor() conducts butterfly bit-row exchanging [36]. The shuffle operations are quite useful for bit-row communication, sharing and achieving register-level data-reuse [6, 29].

(F): Left and right shifting are logical operations. We separate them out to highlight their communication patterns as inter-columns. They are often adopted to extract, exchange and merge bit-columns.

(G): The most interesting operation here perhaps is __ballot(). It returns a 32-bit integer with its Nth bit (from LSB to MSB) contributed as a predicate (1bit) from the Nth lane of the warp. In other words, it offers the ability to convert a bit-column into a bit-row. Recall that in a BSTC, lanes are indexed from top to bottom, while bits are indexed from right to left (Figure 1), __ballot() essentially rotates a bit-column by 90° clockwise to a bit-row.

(H): This operation is the conjugate of (G). In order to rotate a bit-column 90° anticlockwise to a bit-row, one has to combine __ballot() and __brev(). Note that there is no direct operation for rotating from bit-rows to bit-columns; one has to broadcast an entire row to all the lanes via __shfl() and then extract the required bit-column(s), which are quite expensive.

Having these BSTC operations, we can flexibly combine them to design versatile bit-based functions and communication patterns, accelerating various emerging bit-based algorithms [6, 7, 14, 15, 38, 40, 45] on GPUs.

3 BNN BIT FUNCTIONS

We discuss the bit functions in BNN algorithm, where BSTC operations can be applied. Figure 3 shows the network structure for BNN-based LeNet [28]. Note that only the inputs and weights for convolution and fully-connected layers are binarized and their outputs are in full-precision. Other kernels still rely on full-precision calculation and data access.

We first review the differences between BNNs and conventional DNNs/CNNs. As shown in Figure 3, BNNs have a binarization function (i.e., *Bin*) and their own versions of convolution (i.e., *BConv*) and fully-connected functions (i.e., *BMM*). Other functions (e.g., *BN*, *Actv*, *Pool*) are the same as in DNNs/CNNs. The binarization function *Bin* in BNNs is:

$$x^b = sign(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases} \quad (1)$$

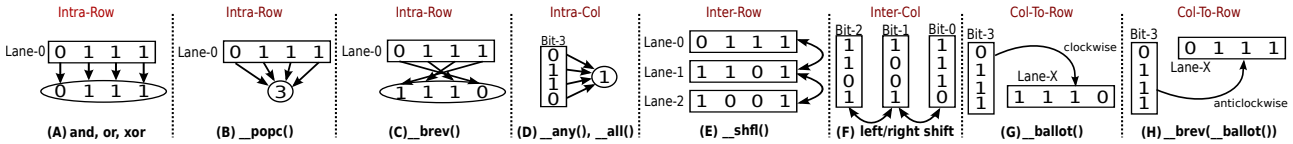

Figure 2: BSTC row-wise and column-wise operations and communication patterns.

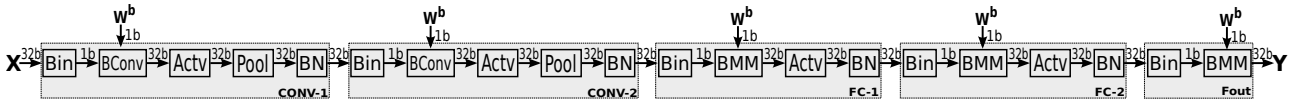

Figure 3: LeNet BNN network structure for inference. W^b refers to binarized weights. "1b" refers to 1-bit, "32b" refers to 32-bit full precision. "X" is the input. "Y" is the output. "Bin" refers to binarization. "BConv" refers to bit-convolution. "Actv" refers to activation, which is ReLU in this paper. "Pool" refers to pooling. "BN" refers to batch-normalization. "BMM" refers to bit-matrix-multiplication or fully-connected layer. Only BMM and BConv are binarized.

Note that the "*sign()*" function in some high-level languages such as Python results not in binarization but in trinarization, as it returns 0 when x is 0. This is not an issue when simulating BNNs through a full-precision implementation (as "$x+0=x$" while "$x\times 0=0$"), but becomes a big problem with padding in low-level implementation. Therefore, using Eq (1) in Python for BNN training may lead to inconsistent results. We solve this issue by adding a small ϵ to x before applying Eq (1) for binarization in Python-based BNN training (e.g, TensorFlow and PyTorch).

For *BConv* and *BMM*, the basic operation is bit dot-product:

$$v = \vec{a}\cdot\vec{b} = popc(\vec{a}\,\&\,\vec{b})$$

where $\vec{a}$ and $\vec{b}$ are bit-vectors, "&" refers to *logical-and*, and *popc* refers to population count. However, this is for a conventional 0/1 dot-product; bit-0 in BNNs is not 0 but -1. Consequently, the dot-product for -1/+1 becomes:

$$v = \vec{a}\cdot\vec{b} = n - 2\times popc(\vec{a}\oplus\vec{b}) = 2\times popc(\vec{a}\bar{\oplus}\vec{b}) - n \qquad (2)$$

where n is the length of $\vec{a}$ and $\vec{b}$; "$\oplus$" refers to *exclusive-or* or XOR, and "$\bar{\oplus}$" refers to *exclusive-nor* or XNOR. As the XNOR gate is widely utilized in digital fabrication, most existing BNN algorithmic studies (e.g., [12, 39]) and FPGA/ASIC based implementations (e.g., [35, 43]) apply XNOR-based design (i.e., $2\times popc(\vec{a}\bar{\oplus}\vec{b}) - n$). But since XNOR is not directly supported on GPUs, we choose to apply XOR approach instead (i.e., $n - 2\times popc(\vec{a}\oplus\vec{b})$). Note that finding the right n is the key for excluding padding bits to ensure the correctness. We will discuss this in detail later.

Overall, the functional differences between BNNs and DNNs fall into four functionalities regarding to bit operations:

Bit-Packing: This corresponds to *Bin* in Figure 3. Following Eq (1), if one lane of a warp reads a datum into its register R0, we then use R0>=0 to generate a bit-column (1 bit per lane), and rely on BSTC op-(H) to rotate the bit-column anticlockwise to a bit-row and distributed it to all 32 lanes of the warp. Here we use op-(H) rather than op-(G) because the result of __ballot() is little-endian, i.e., the first lane corresponds to the LSB of the result. Therefore, we need an additional __brev() to reverse the bit-sequence so that the first lane corresponds to MSB of the 32-bit result. For 64-bit binarization by a warp, __ballot() is repeated twice. After that, the two output unsigned ints are concatenated into a 64-bit unsigned long long int via the following embedded PTX:

```
asm volatile("mov.b64 %0, {%1,%2};":"=l"(l0):"r"(r0),"r"(r1)); //low,high
```

and then reverse as a whole using __brevll(l0).

Bit-Computation: This includes bit-dot-product and other bit operations. Bit-dot-product is used extensively in *BMM* and *BConv*. We rely on BSTC op-(A) and (B) to realize bit-dot-product. OR and AND in op-(A) can be quite useful when dealing with bit-based max- and min-pooling.

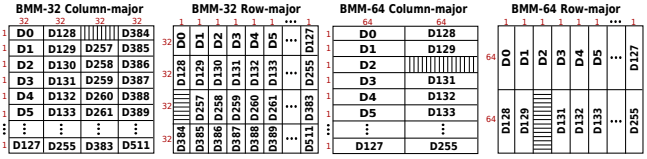

Figure 4: Row-major and Column-major Bit-Packing Format for BMM

Bit-Communication: BSTC op-(E), (F) and (G) are particularly important for this purpose. We will discuss them in detail in Section 4.

Bit-Unpacking: Although this function is not used in BNN inference, it is very useful for debugging as it provides a way to obtain a human-readable result to compare against the correct outputs of a particular layer. Bit-unpacking is the reverse of bit-packing. Op-(F) is adopted here: each lane (corresponding to a bit-row) shifts and extracts a bit, and converts it to full-precision. For the 64-bit version, we can repeat the following routine while altering 31 to 63 for the second pass.

```
C = 2*(int)((R0>>(31-laneid))&0x1)-1;
```

4 SBNN SINGLE LAYER DESIGN

We discuss how to efficiently implement BMM and BConv based on BSTCs.

4.1 Bit Matrix Multiplication (BMM)

Matrix-multiplication comprises both vertical and horizontal data access. To gain performance we propose two bit-packing formats: *row-major* and *column-major*. Figure 4 shows how a 128×128 data block is packed into the two data formats:

- **Column-major**: Each of 32 consecutive raw data from a row of the original data matrix are binarized into 32 0/1 bits via Eq 1 and packed as an *unsigned int* (e.g., D0 in Figure 4), forming a BSTC per warp. These unsigned ints are then organized in a column-major order.
- **Row-major**: Each of 32 consecutive raw data from a column of the original data matrix are binarized and packed as an *unsigned int*, forming a BSTC per warp. These unsigned ints are then organized in a row-major order.

Listing 1 and 2 show the code for binarization into column-major and row-major 32-bit data formats, respectively. Line 11 in Listing 1 binarizes by a warp: 32 lanes cooperatively generate a complete 32-bit binarized result as a bit-row via op-(H) per each iteration, and save the bit-row to corresponding lane in Line 12, building a complete BSTC-32. Line 13 in Listing 2 shows another approach.

```
1  template <typename T>
2  __global__ void PackTo32Col(const T* A,
3  unsigned* B, int A_height, int A_width){
4    unsigned Bval, laneid;//fetch lane-id
5    asm("mov.u32 %0, %%laneid;":"=r"(laneid));
6    #pragma unroll
7    for (int i=0; i<WARP_SIZE; i++){
8      T f0 = A[(blockIdx.x*WARP_SIZE+i)*A_width
9        +blockIdx.y*WARP_SIZE+laneid];
10     //rotate anticlockwise
11     unsigned r0=__brev(__ballot(f0>=0));
12     if (laneid == i) Bval = r0;
13   }
14   B[blockIdx.y*A_height+blockIdx.x*WARP_SIZE
15     +laneid] = Bval;
16 }
```

Listing 1: Binarize into col-major format

```
1  template <typename T>
2  __global__ void PackToC32Row(const T* A,
3  unsigned* B, int A_height, int A_width){
4    unsigned laneid;//fetch lane-id in a warp
5    asm("mov.u32 %0, %%laneid;":"=r"(laneid));
6    unsigned Bval = 0;
7    #pragma unroll
8    //process 1 bit per iteration
9    for (int i=0; i<WARP_SIZE; i++){
10     T f0 = A[(blockIdx.x*WARP_SIZE+i)*A_width
11       +blockIdx.y*WARP_SIZE+laneid];
12     //concatenate a bit on the right
13     Bval = (Bval<<1) | (f0>=0);
14   }
15   B[bx*A_width+by*WARP_SIZE+laneid]=Bval;
16 }
```

Listing 2: Binarize into row-major format

```
1  __global__ void BMM32_Arow_Brow(...){
2  ...
3  register unsigned Cm[32] = {0};
4  for (int i = 0; i < A_width; i++){
5    unsigned r0 = Asub[i*A_height+laneid]);
6    unsigned r1 = Bsub[i*WARP_SIZE+laneid];
7    #pragma unroll
8    for (int j=0; j<WARP_SIZE; j++){
9      //crop a bit-column of B
10     unsigned r2=__brev(__ballot((r1>>j)&0x1));
11     //each lane dot-product with the column of B
12     Cm[j] += __popc(r0 ^ r2);
13 }}  ...
```

Listing 3: BMM by multiplying each A row with each B column

```
1  __global__ void BMM32_Arow_Bcol(...){
2  ...
3  register unsigned Cm[32] = {0};
4  for (int i = 0; i < A_width; i++){
5    unsigned r0 = Asub[i*A_height+laneid]);
6    unsigned r1 = Bsub[i*B_width+laneid];
7    #pragma unroll
8    for (int j=0; j<WARP_SIZE; j++){
9      //broadcast a bit-row of B from lane-j
10     unsigned r2 = __shfl(r1, j);
11     //each lane dot-product with the row of B
12     Cm[j] += __popc(r0 ^ r2);
13 }}  ...
```

Listing 4: BMM by multiplying each A row with each B row

Each lane concatenates a single bit per iteration to its own bit-row and the whole BSTC-32 is constructed after 32 iterations. Note that Listing 1 stores data into B by column-major order while Listing 2 stores data into B by row-major order. The two bit conversion methods have similar delay. The 64-bit versions are analogous.

Having binarized the matrices, we are ready to perform BMM. To simplify the discussion, we first make three assumptions: *(a) A and B are bit matrices stored in row-major data format. (b) A and B are not transposed, and (c) the sizes of A and B are divisible by 32.* With these assumptions, we first partition A and B into 32×32 bit-blocks, corresponding to two BSTCs in two registers. Then, the block-wise BMM can be achieved by fetching a bit-column of BSTC in register B, rotating anticlockwise by 90° via op-(H), then dot-producting with a bit-row of BSTC in register A via op-(A)/(B), and storing the result to C as a full-precision element (Figure 3). Listing 3 shows the major part of the code. Line 10 shows how to crop a bit-column from Bsub (B is row-major). Each lane contributes one bit per iteration, which is then dot-producted with bit-rows from Asub held by R0 of each lane.

Relaxing Assumption-(a): Consider that if B is stored in column-major data-format, we can avoid the bit-column cropping operation by broadcasting a bit-row of Bsub via __shfl() of op-(E). Listing 4 shows the code: if B is column-major, then rather than cropping a bit-column, we now only need to broadcast a bit-row of B per iteration. It is interesting to see that the format transition from row-major to column-major for B is equivalent to each 32×32 bit-tile in B undertaking a tile-wise transpose. Similarly, if A is stored in column-major, B in row-major, or if both A and B are stored in column-major, there are two more compositions. To summarize, there are four alternative approaches (one per combination of data formats):

- $A \rightarrow A^b_{col\text{-}major}, B \rightarrow B^b_{col\text{-}major}, BMM_{row\text{-}by\text{-}col} \rightarrow C$
- $A \rightarrow A^b_{col\text{-}major}, B \rightarrow B^b_{row\text{-}major}, BMM_{row\text{-}by\text{-}row} \rightarrow C$
- $A \rightarrow A^b_{row\text{-}major}, B \rightarrow B^b_{row\text{-}major}, BMM_{col\text{-}by\text{-}row} \rightarrow C$
- $A \rightarrow A^b_{row\text{-}major}, B \rightarrow B^b_{col\text{-}major}, BMM_{col\text{-}by\text{-}col} \rightarrow C$

where A^b refers to the binarization of A. We have evaluated all these four combinations and found that the second choice (Listing 4) shows the best performance (on a Tesla-P100, the delays are 28K, 20K, 28K, 56K cycles, respectively). An explanation is that, compared with __ballot(), communicating through __shfl() appears to be more efficient. Among the four options, only the second does not use __ballot().

```
1 register unsigned Cm[64] = {0};
2 for (int i=0; i<A_width; i++){
3   unsigned long long a0 = Asub[i*A_height+laneid]); //1st half of A
4   unsigned long long a1 = Asub[i*A_height+WARP_SIZE+laneid]);//2nd half of A
5   unsigned long long b0 = Bsub[i*B_width+laneid]);//1st half of B
6   unsigned long long b0 = Bsub[i*B_width+WARP_SIZE+laneid]);//2nd half of B
7   #pragma unroll
8   for (int j=0; j<WARP_SIZE; j++){
9     unsigned long long l0 = __shfl(b0, j); //broadcast 64 bits
10    unsigned long long l1 = __shfl(b1, j); //broadcast 64 bits
11    Cm[j] += (__popcll(a0^l0)<<16) | __popcll(a1^l0);//dot-product and register packing
12    Cm[j+WARP_SIZE] = (__popcll(a0^l1)<<16) | __popcll(a1^l1); }}
```

Listing 5: 64-bits Bit-Matrix-Multiplication Kernel (BMM-64)

Relaxing Assumption-(b): We find that transposition (A, B, or both) can be realized through a different composition of the data format (i.e., row-major, col-major) and BMM approaches (i.e., row-by-row, row-by-col, col-by-row, col-by-col). For example, if A is to be transposed before BMM, it is equivalent to binarizing A and B into row-major and performing row-by-row BMM. For each scenario of transposition (i.e., $A \times B$, $A \times B^T$, $A^T \times B$, $A^T \times B^T$), we measured all 16 combinations of data-formats and BMM approaches and found that row-by-row based tiled-BMM (in Listing 4) always outperforms the other three. Therefore, the optimal design summarizes as:

- $A \times B$: $A^b_{col\text{-}major}, B^b_{row\text{-}major}, BMM_{row\text{-}by\text{-}row} \rightarrow C$
- $A^T \times B$: $A^b_{row\text{-}major}, B^b_{row\text{-}major}, BMM_{row\text{-}by\text{-}row} \rightarrow C$
- $A \times B^T$: $A^b_{col\text{-}major}, B^b_{col\text{-}major}, BMM_{row\text{-}by\text{-}row} \rightarrow C$
- $A^T \times B^T$: $A^b_{row\text{-}major}, B^b_{col\text{-}major}, BMM_{row\text{-}by\text{-}row} \rightarrow C$

Relaxing Assumption-(c): In the case where matrix size is not a factor of 32 (e.g., in the output layer), padding is necessary. For full-precision GEMM, padding with zeros does not affect the correctness. However, for BMM, padding with zeros is equivalent to padding with -1s, so we need an approach to eliminate the effect of padding. Two important observations have been made: (1) Eq 2 essentially has a nice feature: if $\vec{a}$ and $\vec{b}$ are both padded with t bits and all the $2 \times t$ bits remain the same (either 0 or 1), it does not impact $popc(\vec{a} \oplus \vec{b})$. So as long as n is the actual length before padding (i.e., n rather than $n + t$), the padded bits will not affect the correctness of the result. (2) When writing to C, we must avoid writing outside the output matrix boundary.

BMM-64: The major differences between BMM-32 and BMM-64 include: (a) Although in BMM-64 the bitwidth becomes 64-bits, a warp still only has 32 lanes. Therefore, to process a 64×64 bit-block (corresponding to two BSTC-64), a warp has to process it twice: one round for each half of the bit-block (i.e., a BSTC-64). Segmentation and concatenation from/to one 64-bit datum to/from two 32-bit data are required. (b) In Line 3 of Listing 3 and 4, we use per-lane registers to temporarily buffer C's partial accumulation result Cm, with a demand of 1024 registers per warp. However, such a strategy cannot be applied to BMM-64 as it requires too many registers. To address this, we propose register packing: 16 bits is already sufficient to hold C's partial results, as shown in Listing 5.

Extracting Fine-Grained Parallelism: In the current design, each warp processes a 32×32 or 64×64 subblock. However, the latest GPUs contain dozens of streaming multiprocessors (SMs). For example, in order for V100 GPUs to be fully-loaded, with the current occupancy (i.e., 0.5 due to register constraints), it requires the matrix size to be at least 1620×1620 ($32\text{bits} \times \sqrt{80\text{SMs} \times 32\text{warps}} = 1620$)

for BMM-32 and 3240×3240 for BMM-64. This is too much for small BNN fully-connected layers and can easily lead to workload imbalance, especially when the batch size is small. Therefore, we propose a lightweight batched version of BMM-32 and BMM-64, which spreads out the loop in Line 7-8 of Listing 4 and Listing 5 to different warps. This may sacrifice some register-level data reuse, but can effectively extract more fine-grained parallelism to feed all GPU SMs and alleviate potential workload imbalance. Such a more fine-grained design can bring significant speedups for small input matrices.

4.2 Bit Convolution (BConv)

For convolution, the basic operation is still bit dot-product. Again, we need to compact the data first. To be general, we adopt the following API as the interface of BConv, which is similar to Tensorflow's conv2d() API.

```
1 Bconv2d(x, W, stride_h, stride_w, padding, use64bit)
```

where x is the input image, which is a 4D tensor in shape *[batch, in_height, in_width, in_channels]* (i.e., "*NHWC*"); W is the filter, a 4D tensor in shape *[filter_height, filter_width, in_channels, out_channels]*; *stride_w* and *stride_h* are the strides along height and width dimensions; padding option can be "*SAME*" or "*VALID*". The task of BConv is to binarize x and W into x^b and W^b with Eq 1, and convolve x^b with W^b to a generate full-precision 4D output image tensor in the same format as input *[batch, out_height, out_width, out_channels]*:

$$x^b[s,t] * W^b[s,t] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} x^b[m,n] \cdot W^b[s-m, t-n]$$

For a high-performance implementation, identifying the optimal mapping from algorithm parallelism to hardware parallelism is essential, i.e., with the most data reuse. Regarding BConv, since the only connection between x and W is "in_channels", to gain the maximum spatial benefit, we binarize along "in_channels" (except for the input layer). Binarization is similar to BMM following Eq 1. In bit convolution, to gain the most fine-grained parallelism we use each warp to process one element of the output. Therefore, for an image of size 224×224 (batch=1) from ImageNet, we spawn $224 \times 224 \times 1 = 50176$ warps, which is sufficient to feed all SMs. Since every output image element is produced by dot-producting in size of [filter_height, filter_width, in_channels, out_channels], the hardware parallelism, which is warp_size × 32/64 bit = 1024/2048, needs to spread along filter_height×filter_width×in_channels×out_channels. For filter_height and filter_width, the difficulty is the need to handle padding at the boundary; these two dimensions are thus processed sequentially. The remaining issue then becomes how to spread 1024/2048 along in_channels×out_channels more efficiently.

First, let's suppose in_channel is a factor of 32 or 64, Listing 6 shows the BConv code. The design is a direct convolution: it does not follow the implementations of cuDNN [10] or Caffe [22], which flatten the input image first and then go through GEMM [8]. Therefore, we do not require dozens of different strategies to efficiently handle different scenarios as done by cuDNN [10]. In addition, we do not require extra GPU memory as specialized workspace to buffer the flattened matrix, which can be huge in size. As shown in Listing 6, *out_height* is mapped to block-grid Y-dimension (Line 9); *out_width* is mapped to block-grid X-dimension (Line 8); *batch* is mapped to block-grid Z-dimension (Line 10), which is 1 (or very small) for inference. *filter_height* and *filter_width* are mapped to loop r and s, respectively, for sequential execution. For a single

```
1  template <typename T>
2  __global__ void Conv2d_32bit(const T* __restrict__ input, const unsigned* __restrict__ bitfilter ,
3  T* output, const int in_channels, const int out_channels, const int in_width,
4    const int in_height, const int filter_width, const int filter_height,
5    const int batch, const int stride_vertical, const int stride_horizontal,
6    const int out_width, const int out_height, const int pad_h, const int pad_w) {
7      unsigned laneid; asm("mov.u32 %0, %%laneid;":"=r"(laneid));//fetch lane-id in a warp
8      const int bx = blockIdx.x; //over out_width
9      const int by = blockIdx.y; //over out_height
10     const int bz = blockIdx.z; //over batch
11     const int ins = (in_channels >> 5);//number of steps in in_channels
12     extern __shared__ unsigned Csub[];
13     const int ax0 = bx*stride_horizontal-pad_w;//from output image idx to input image index
14     const int ay0 = by*stride_vertical-pad_h;
15     int exclude = 0; //track the number of filter entries that are masked off
16     for (int i=laneid; i<out_channels; i+=WARP_SIZE) Csub[i] = 0;
17     for (int r=0; r<filter_height; r++) { //load a window of data from Input
18       for (int s=0; s<filter_width; s++){
19           const int ay = ay0 + r; //y-coord in Input
20           const int ax = ax0 + s; //x-coord in Input
21           if ( (ay>=0) && (ay<in_height) && (ax>=0) && (ax<in_width) ) {//within Input frame
22               for (int c=0; c<ins; c++) {//per 32-bit in in_channels
23                   //Input[batch, in_height, in_width, in_channels]
24                   T f0 = input[bz*in_width*in_height*in_channels
25                       +(ay*in_width+ax)*in_channels+c*32+laneid];
26                   unsigned r0 = __brev(__ballot(f0>=0));//binarize
27                   for (int i=laneid; i<out_channels; i+=WARP_SIZE) {
28                     unsigned r1 = bitfilter [(r*filter_width+s)*ins*out_channels+c*out_channels+i];
29                     Csub[i] += __popc(r0 ^ r1);}
30           }}} else { //not in frame, so track
31               exclude++; }}} //accumulate
32     for (int i=laneid; i<out_channels; i+=32)
33       output[(bz*out_height*out_width*out_channels) //batch
34           + (by*out_width*out_channels)//out_height
35           + (bx*out_channels) + i] //out_width
36           = in_channels*filter_width*filter_height //flatten matrix size
37             - exclude*in_channels //eliminate padding distoration
38             - (2*Csub[i]); } //n-2popc(a^b) for 0/1 to simulate sum(a*b) for +1/-1
```

Listing 6: 32-bits Bit Convolution Kernel (BConv-32)

warp or thread block (following the warp-consolidation model), we map in_channels to the bit-width (32 or 64), and out_channels to warp-lanes (32). In Line 24-26, each warp fetches 32 coalesced full-precision data from the input image, binarizes them as an unsigned int (i.e., R0) and broadcasts R0 to all 32 lanes through __ballot(). Then in Line 28, each lane fetches 32bits (i.e., R1) with different out_channel ids from the binarized filter. After that, R0 and R1 are dot-producted in Line 29. If in_channel is larger than 32/64, we iterate along in_channel (loop c in Line 22) as a step of 32/64. If out_channel is larger than warp_size, we iterate along out_channel (loop i in Line 27) at a step of warp_size. Note that writes to Csub in shared memory does not incur bank conflicts. Finally, at Line 33, we traverse out_channels by the whole warp to perform efficient coalesced writing.

Padding: The more interesting part of BConv lies in padding. When the padding option is "*VALID*", no padding is needed; but when it is "*SAME*", padding is required. As mentioned in Section 4.2, simply padding 0 is not feasible as 0 is already encoded for -1. For BMM, a unified boundary condition to exclude the padding elements solves the problem, but for BConv, it is more difficult: the input image and filter are not necessarily aligned. This is especially the case for the corner elements and when the stride is not 1. Divergent from existing CPU-based BConv implementations [19], which require pre-processing and/or post-processing, we handle the padding issue on the fly with no extra memory cost. As shown in Line 31, we declare an *exclude* variable to track when the convolving element falls out of the input image frame (Line 21), calculate the actual vector length (the padded bits should not be counted as effective bits, recall Eq 2 and Section 4.2), and make corresponding amendments (i.e.,$-exclude*in\_channels$) when saving the results in Line 36-38. Note that this simple strategy is only feasible for direct convolution rather than the widely-used flatten&GEMM approach [8, 10, 19, 22]. This is because we need to convey the padding information of each element from the flatten kernel to the BMM kernel in order to distinguish a padding 0 from a normal 0. As a consequence, extra space cost and/or pre-/post-processing are inevitable.

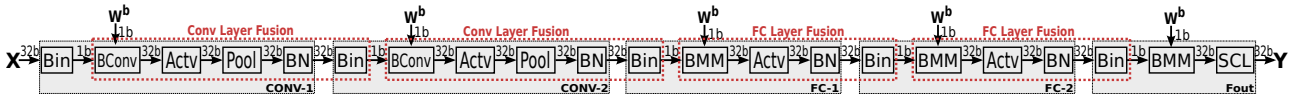

Figure 5: New Layer Partition for BNN LetNet Network.

BConv-64 BConv-64 is similar as BConv-32 except that each lane of a warp processes a 64-bit in_channel per iteration of *c*. In addition, we need two op-(G) for binarization and 64-bit op-(B) for bit accumulation.

5 SBNN INFERENCE IN A KERNEL

In this section, we present how to fuse the whole SBNN network model into a single GPU kernel. First, we discuss why it is possible for conducting all-layer fusion in our SBNN design. Then, we focus on presenting the intra-layer fusion strategy. Finally, we show how to fuse across layers.

5.1 Design Prerequisite

In this section, we propose the novel idea of fusing all DNN layers into a single GPU kernel. First, we will discuss some prerequisites for a successful all-layer fusion:

(1) Hardware support for GPU global synchronization. Prior to the Pascal architecture and CUDA Runtime-9.0, global synchronization among thread blocks is not feasible. Recently, with the introduction of *cooperative-groups*, programmers can declare the grid-group through grid_group grid=this_grid() and use grid.sync() to synchronize across the kernel grid. To enable this functionality, a kernel has to be launched by cudaLaunchCooperativeKernel() and compiled with *-rdc=true* option [36]. Global synchronization is necessary for ensuring data consistency in our SBNN cross-layer fusion.

(2) Novel layer partition design for minimal cross-layer data movement. Figure 5 shows our proposed new layer partition method for SBNN, in comparison to the original design in Figure 3. The dotted-red-boxes mark the new LeNet layer boundaries, where we move binarization (i.e., Bin) of the next layer into the previous layer. In this way, we partition the SBNN network into two major types of layers: FC layers and Conv layers. This new partition approach brings the following benefits. First, only the data movement between Bin and BConv/BMM are binarized; others are full-precision types. Thus, the data write and read between layers can be minimized according to this layer partition. Second, although Actv, Pool, BN, Bin are all element-wise functions, BConv and BMM are not. As workload per Conv/FC layer can be significantly different, to achieve high occupancy for a small batch, we have to re-balance workload as evenly as possible among SMs per layer. As there is no inter-SM communication network, the image data forwarded across layers have to be stored and re-fetched per layer. To avoid potential memory dependency violations, a global synchronization is often required for SMs to stay synchronized.

(3) Unified GPU kernel configuration across fused layers. To fuse two GPU kernels, a basic requirement is that they share the same kernel configuration, i.e., *gridDim* and *blockDim*. "*gridDim*" is often related to input data (e.g, the volume of tasks), and "*blockDim*" is strongly correlated to programming models and occupancy, which may greatly impact performance. Traditional CUDA based designs assign each layer function (e.g., conv2d, gemm, pool, batch-normalization, etc) its individual ideal *blockDim* and *gridDim* for archiving the best performance for the targeted workload, resulting in multiple kernels. On the contrary, since our BSTC-based SBNN kernel design (e.g., BMM, BConv) follows the warp-consolidation model [29], *blockDim* is always 1D with 1 warp (or 32 threads). Regarding to *gridDim*, we adopt elastic kernel [37] or warp-delegation [30] to unify this value across layers. In this way, a unified GPU kernel can be formed across all the fused layers, avoiding multi-kernel launching and release. Specifically, we allocate the maximum number of warps (or thread blocks) the GPU resources can sustain simultaneously as elastic agents, which iteratively fetches the thread block jobs from a task list. A runtime throttling scheme on thread blocks is also provided to control concurrency for best performance.

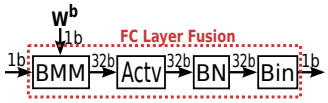

Figure 6: SBNN FC Layer Fusion

5.2 Intra-Layer Fusion

We show how to fuse the functions inside the red-dotted-boxes in Figure 5 into a single GPU kernel.

(a) FC Intra-Layer Fusion. Here, we adopt a backward fusing strategy, as shown in Figure 6. First, we fuse binarization (Bin) with batch-normalization (BN). BN has already been discussed in Eq (1). BN [21] is shown below:

$$y_{i,j} = \left(\frac{x_{i,j} - \mathbb{E}[x_{*,j}]}{\sqrt{Var[x_{*,j}] + \epsilon}} \right) \cdot \gamma_j + \beta_j \tag{3}$$

where $\mathbb{E}$ is the mean value across the batch. Var is the variance. ϵ is a small scalar to avoid zero division. γ and β are the learned scaling factor and bias. i iterates over the batch, j iterates over each output data element (i.e., output_channels×output_height×output_width). Combining Eq (3) and Eq (1), and given the fact that γ is a scaling factor being positive [1], we have

$$y_{i,j} = \begin{cases} 1 & \text{if } x_{i,j} \geq \mathbb{E}_{*,j} - \frac{\beta_j \cdot \sqrt{Var[x_{*,j}]+\epsilon}}{\gamma_j} \\ -1 & \text{otherwise} \end{cases} \tag{4}$$

where $y_{i,j}$ is the output of Bin which is in binary while $x_{i,j}$ is the input of BN which is in full-precision. Then, we integrate Actv. For DNN or BNN, Actv is often ReLU, or $y = max(x, 0)$. Therefore, the combination becomes

$$y_{i,j} = \begin{cases} -1 & \text{if } max(x_{i,j}, 0) < \mathbb{E}_{*,j} - \frac{\beta_j \cdot \sqrt{Var[x_{*,j}]+\epsilon}}{\gamma_j} \\ 1 & \text{otherwise} \end{cases} \tag{5}$$

where $y_{i,j}$ is the output of Bin which is in binary. $x_{i,j}$ is the output of BMM. FC intra-layer fusion is essentially to apply Eq 5 on each output of BMM before storing into binarized output matrix. The threshold $\mathbb{E}_{*,j} - \frac{\beta_j \cdot \sqrt{Var[x_{*,j}]+\epsilon}}{\gamma_j}$ is a floating-point number. But since $max(x_{i,j}, 0)$ in BNN is an integer, this offers opportunities to round up the threshold to an integer. Thus, all parameters in our

[1] This is true for Tensorflow but not necessarily for PyTorch. In that case, we can inverse the corresponding channels of the weight to ensure γ converts to positive since bias=0 for BConv and BMM. Or, we can constraint γ to be always positive during training.

```
unsigned c=0; //amendment to BMM−32
if (bx*32+laneid<A_input){ //after padding, if height is in boundary
  for(int i=0; i<32; i++){//concatenate bits
    c <<=1;// left shift to leave space for the next bit
    if (by*32+i<B_width){//after padding, if width is in boundary
      int t = max( A_width − 2 * Cm[i], 0); //ReLU
      c |= (t < threshold[by*32+i]); }}} //Batch−normalization and Binarization
Csub[laneid] = c; // coalesced writing
```

Listing 7: FC Layer Fusion Amendment to BMM-32

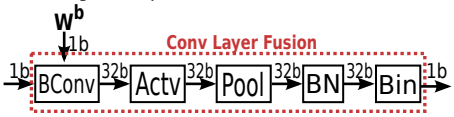

Figure 7: SBNN Conv Layer Fusion

```
__device__ __inline__ void ConvPool32Layer(Conv32LayerParam* p){
  extern __shared__ int Cs[];
  volatile int* Csub = (int*)&Cs;
  for (int bid = blockIdx.x; bid < (p->output_height)*(p->output_width)*(p->batch);
    bid += gridDim.x) {
    int bz = bid / ((p->output_width)*(p->output_height));
    int by = bid % ((p->output_width)*(p->output_height)) / (p->output_width);
    int bx = bid % ((p->output_width)*(p->output_height)) % (p->output_width);
     ... // Original BConv−32 Kernel Call
}
```

Listing 8: Warp-Delegation for SBNN Inter-Layer Fusion

implementation are integers, except the final output layer, where no further binarization can be fused. Listing 7 shows the necessary amendments to the original BMM-32 in Listing 4 when integrating Actv, BN and Bin. Also, the output is binarized data, reducing data writes by 32x.

(b) Conv Intra-Layer Fusion. Convolution intra-layer fusion is similar to FC, except that we have to integrate the pooling function Pool, as shown in Figure 7. Pooling function is typically to pick the maximum or minimum value among a $m \times m$ tile, where m is generally 2. If we move the Pool function after Bin, their combination essentially offers a new opportunity for *branch-pruning*: for max-pooling, if one of the 2×2 elements is shown to be 1 after going through BN and Bin, the computation of the remaining 3 elements in the pooling window can be pruned since Bin already has its maximum possible value of 1. Similar condition applies to min-pooling with value 0.

Since Pool is not strictly element-wise function, one approach is to perform a 2×2 thread coarsening. In other words, each warp handles 4 output elements sequentially. Thus, max-pooling can be achieved by

```
output[((bz*output_height+(by/2))*output_width +(bx/2))*(output_channel/32)+k] |= C;
```

Alternatively, to extract the most fine-grained parallelism, we still spread out a warp for each output image before pooling. To process pooling, we use bitwise OR and AND for max/min pooling when writing binarized results to the pooled output image. However, as the 2×2 data elements are processed by 4 independent warps, a race condition may occur. We apply atomicOr() and atomicAnd() before writing to output image (the latter option is often much faster than the former), e.g.,

```
atomicOr(output[((bz*output_height+(by/2))*output_width
  +(bx/2))*(output_channel/32)+k], C);
```

5.3 Inter-Layer Fusion

Now we are ready to fuse all the layers into a single kernel. Since BMM and BConv functions have been designed based on the warp-consolidation model, *blockDim* for all the layers are already unified to 32. Here we show how to unify *gridDim* across layers via elastic kernel [37] and warp-delegation [30]. Listing 8 shows an example on BConv-32 with pooling. We have developed a corresponding

```
#include <cooperative_groups.h>
//================================== Inference Network ==============================
__global__ void LeNet(InConv32LayerParam* bconv1,Conv32LayerParam* bconv2,Fc32LayerParam* bfc1,
  Out64LayerParam* bout){
  grid_group grid = this_grid();
  InConvPool32Layer(bconv1); //1st Convolution Layer with Pooling
  ConvPool32Layer(bconv2); //2nd Convolution Layer with Pooling
  Fc32Layer(bfc1); //1st Fully−Connnect Layer
  Out32Layer(bout);} //2nd Fully−Connnect Layer
//===================================================================================
int numBlockPerSm; //maximum number of thread−blocks per SM under current occupancy
int numThreads = 32; //warp size following warp−consolidation model
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, GPU_dev); //GPU_dev is GPU device id
cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlockPerSm, LeNet,
      numThreads, shared_memory);
void* args[] = {&bconv1, &bconv2, &bfc1, &bout};//set kernel function parameters
cudaLaunchCooperativeKernel((void*)LeNet, numBlockPerSm*deviceProp.multiProcessorCount,
  numThreads, args, shared_memory);//kernel invokation
```

Listing 9: SBNN Singular Kernel for LeNet Network

parameter object *Conv32LayerParam* to pack all the required parameters of that particular layer. In Line 4-5, we use all the allocated thread blocks to traverse the original *gridDim* space and calibrate the thread block id to its dedicated task in the original space (Line 6-8). After that, we invoke the original BConv-32 kernel for processing. In this way, all the layer kernels can have the identical *gridDim*, getting ready for fusion.

We then fuse all the layers into a single kernel, as shown in Listing 9. This kernel is to be written by the users to describe the network structure, with one parameter object per layer. "grid.sync()" is inserted at the end of each layer function as a global barrier. As can be seen, the fused kernel is very easy to write from users' perspective: an appropriate parameter object for each layer can be simply declared, followed by constructing the __global__() function as Listing 9. To increase hardware occupancy, by default we allocate the maximum number of thread blocks allowed under the present resources consumption in Line 15. However, users can tune this value as needed. The kernel function parameters are configured in Line 17 and the fused-kernel is invoked in Line 18-19.

There are four design considerations in our fused approach. (1) After our new layer partition approach is applied (Section 5.1), the image data to be written by the former layer and immediately read by the following layer are condensed bit-data, which is possible to fit into on-chip memory, e.g., registers and shared memory (e.g., in Line 2-3 of Listing 8, Cs is shared by all the layers). Ideally, there is no global memory image data read and write in each layer. However, one complexity is that BMM and BConv may have data dependency across SMs (one SM may need data from another SM). Unless data are written into the global memory, SMs cannot share on-chip data – there is no inter-SM network for current GPUs. Thus, some global write and read as well as synchronization are necessary. Nevertheless, the overhead should be relatively small. This is because the volume of data has already been reduced by 32x; and since read is often immediately followed by write (of the same image-data), L2 cache rather than off-chip global memory is leveraged for buffering. So the data is essentially still kept on-chip. In fact, we observe very high L2 cache hit rate for the fused approach. (2) The first layer is the input layer, in which the input image is in full-precision and in_channel is generally small (e.g. 1 for gray image and 3 for RGB). In addition, for a large network, the input image is not binarized in the first layer to avoid significant information lost. In our fused design, we have developed special parameter objects and layer implementation for the input and output layers (i.e., the input layer is typically convolution while the output layer is generally fully-connected). (3) For the fully-connected layers, the input image is binarized in column-major (weight in row-major) while for convolution layer, the input image is binarized along in_channel. Therefore, to connect the last convolution layer with the first fully-connected

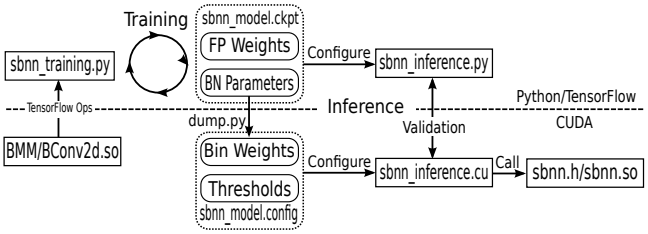

Figure 8: Framework for SBNN training and inference.

layer, the bit image data has to be transposed. We set a configurable option for the convolution layer to correctly write to the output image (i.e., transpose during writing) so that such transposition will not lead to additional overhead. (4) For newer network structures such as ResNet [18], we need to save the full-precision convolution result before binarization for subsequent merging. We are currently looking at how to fuse the entire basic block or bottleneck block into a single function so the residual saving, fetching and processing can be quite efficient.

5.4 SBNN Inference Framework

For validation and wide adaption, we have developed a framework to train an SBNN and configure its weights & thresholds for inference. As shown in Figure 8, we train SBNN in *TensorFlow* [17]. We have encapsulate BMM-32/64 (Listing 4) and BConv-32/64 (Listing 6) as a dynamic library that can be called as TensorFlow operations [17] during training. The intention is to ensure that the behavior of training forward-pass is exactly the same as our SBNN inference. After an SBNN network is trained, a script *dump.py* is employed to extract the weights and batch-normalization parameters (i.e.,$Var, \mathbb{E}, \beta, \gamma$) from the TensorFlow dumped checkpoints (e.g., *sbnn_model.ckpt*), calculate the thresholds, and save weights & thresholds information into an SBNN configuration file (e.g., *sbnn_model.config*). The configuration file is then parsed by SBNN inference program during initialization. We have assigned a certain format for the configuration file and provided APIs to parse such a file in C/CUDA. For validation and debugging purposes, we also develop a TensorFlow inference script in addition to the training Python script (Figure 8).

6 EVALUATION

We evaluate our design of BMM, BConv, and SBNN on four state-of-the-art GPU platforms: an HPC-oriented NVIDIA Tesla-P100, V100, and edge-oriented Jetson-TX1 and TX2, as listed in Table 1. It is expected that the Tesla platforms will be used for cloud or HPC inference while the Jetson platforms will be used for edge inference in an embedded environment.

6.1 Bit-Matrix-Multiplication

First, we measure the latency of BMM with matrix size scaling from 64×64 to 32768×32768. We compare our design against *cuBLAS* (i.e., simulating +1/-1 BMM through full-precision SGEMM) and *bnn-baseline* (i.e., the BMM design from the original BNN work [12]). Note that the cuBLAS scheme does not include binarization. Besides, *bnn-baseline* implementation can only run correctly when the matrix size is a factor of 512 and it does not support transposition. There are 8 different schemes to be evaluated from this study, described in Table 2: *BMM-32* (the 32bit BMM in Listing 4), *BMM-64* (the 64bit BMM in Listing 5), *BMMS-32* (the variant of BMM-32 by extracting fine-grained parallelism, as discussed in Section 4.1), *BMMS-64* (the variant of BMM-64 by extracting fine-grained parallelism), *BMM-32B* (BMM-32 with its two input matrices already binarized before processing and its output matrix binarized before return; i.e., both inputs and output matrices are bit matrices, which is the condition of SBNN inference. Similarly, *BMM-64B*, *BMMS-32B*, and *BMMS-64B* are described in Table 2. We validate the correctness of results by comparing against cuBLAS. The reported figures are the average of multiple runs.

Figure 9, 10, 11 and 12 illustrate the measured latencies of BMM with increased matrix size on the four platforms in Table 1. In order to fit the curves into a single figure, we normalized the latency with respect to *cuBLAS*. On the Maxwell-based Jetson-TX1, *BMMS-64* and *BMMS-64B* are failed to run due to insufficient hardware resources. Overall, *BMM-32B* on the P100 GPU shows the best performance, delivering ~ 10× speedup over cuBLAS simulated BMM, with 1K matrix size. In general, the 32-bit BMM schemes show better performance than their 64-bits counterparts. This is mainly the result of trading-off between *workload balancing* and *register data reuse*. The 64-bits BMM achieve better data reuse than the 32-bits BMM, but when the matrix size is small, it suffers from insufficient workload for all SMs. The fine-grained versions, although do not show an obvious advantage over the coarse-grained version here, appears to give a significantly better performance in SBNN inference, especially for the 64bit versions (as will be seen in Section 6.3). We have also evaluated BMM with transposition. The results show that the transposition overhead is negligible since it is just an alternative selection of data format (see Section-3). If the matrix size S is not a factor of 32 for 32-bit BMM, or 64 for 64-bits BMM, padding is required and some computation will be wasted; the performance is the same as when matrix size is $\lceil S/32 \rceil \times 32$ or $\lceil S/64 \rceil \times 64$.

6.2 Bit Convolution

We then measure the performance of BConv. In contrast with matrix-multiplication, convolution needs many more parameters: *input_size*, *filter_size*, *input_channel*, *output_channel*, *batch*, *stride* and possibly *pooling*. Due to page limitation, we only show the curves with respect to input_channel and output_channel – these are frequently varied with layers of the same network. We set *input size*=64 (medium image size), *filter size*=3 (most frequently used), *stride*=1 (most frequently used), and *batch*=1 (for inference). Evaluation with respect to other parameters (e.g., batch, input_size, filter_size) are provided in the supplementary file. Since there is no existing BConv design for GPUs, we compare our approaches against FP-32 cuDNN simulated BConv.

There are two major implementations for convolution in cuDNN: the basic one without any workspace utilization (i.e., CUDNN_CONVOLUTION_FWD_NO_WORKSPACE, marked as *cuDNN-base*) and a fast one demanding large workspace (i.e., CUDNN_CONVOLUTION_FWD_PREFER_FASTEST, marked as *cuDNN-fast*). Our implementation includes four versions: *BConv-32* and *BConv-64* are 32-bit and 64-bit BConv with input, filter and output in 32-bit full-precision. *BConv-32B* and *BConv-64B* are 32-bit and 64-bit BConv with input, filter and output in bits (so, we avoid the binarization of input and filter, but add the binarization of output).

The results are normalized to *cuDNN-base*, shown in Figure 13, 14, 15 and 16 for *input-channels* (output_channel=64), and in Figure 17, 18, 19 and 20 for *output_channels* (input_channel=64) on the four platforms (Table 1), respectively. As with BMM, the 64-bit version cannot run properly on the Maxwell-based Jetson-TX1 platform due to resource limitation. From these figures, we can observe that

Table 1: GPU SM resource configuration. "Reg" refers to the number of 4B register entries.

GPU	Arch	CC.	Freq.	Rtm.	SMs	CTAs/SM	Warps/SM	Warps/CTA	Reg/SM	Reg/CTA	Reg/Thd	Shared/SM	Shared/CTA
Tesla-P100	Pascal	6.0	1481 MHz	9.1	56	32	64	32	64K	64K	255	64KB	48KB
Tesla-V100	Volta	7.0	1530 MHz	9.0	80	32	64	32	64K	64K	255	96KB	96KB
Jetson-TX1	Maxwell	5.3	998MHz	9.0	2	32	64	32	64K	32K	255	64KB	48KB
Jetson-TX2	Pascal	6.2	1301MHz	8.0	2	32	64	32	64K	32K	255	64KB	48KB

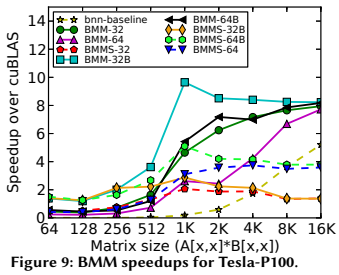

Figure 9: BMM speedups for Tesla-P100.

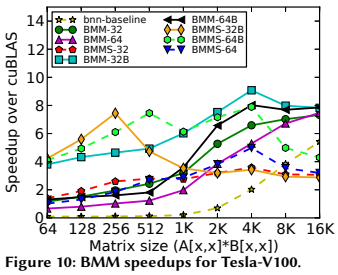

Figure 10: BMM speedups for Tesla-V100.

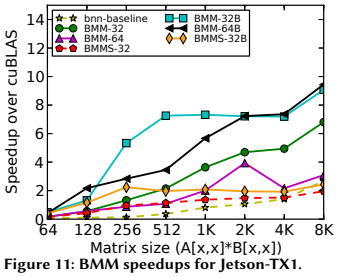

Figure 11: BMM speedups for Jetson-TX1.

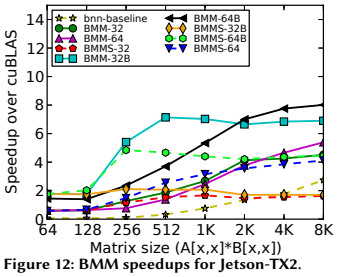

Figure 12: BMM speedups for Jetson-TX2.

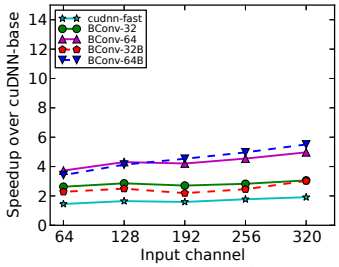

Figure 13: BConv speedups on Tesla-P100.

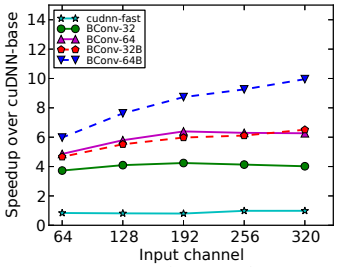

Figure 14: BConv speedups on Tesla-V100.

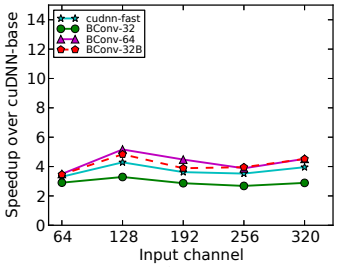

Figure 15: BConv speedups on Jetson-TX1.

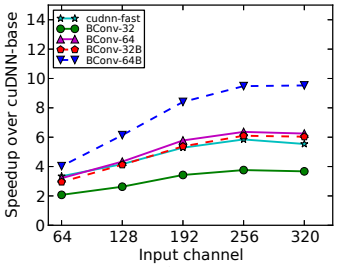

Figure 16: BConv speedups on Jetson-TX2.

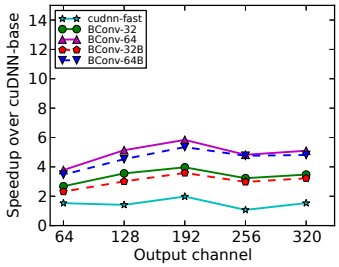

Figure 17: BConv speedups on Tesla-P100.

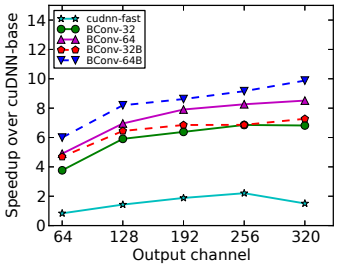

Figure 18: BConv speedups on Tesla-V100.

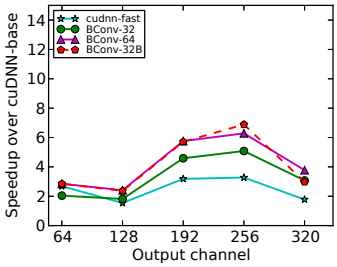

Figure 19: BConv speedups on Jetson-TX1.

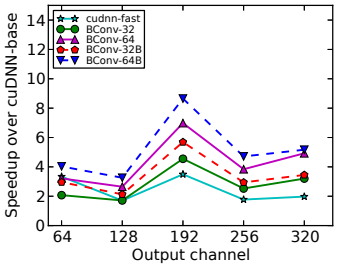

Figure 20: BConv speedups on Jetson-TX2.

Table 2: Performance Comparison. Input and Output refer to bit-width per data element for input data and output data.

Schemes	Description	Algorithm	Input	Output
cuBLAS	Simulating BMM via SGEMM	SGEMM	32bit	32bit
bnn-baseline	BMM from the BNN paper [12]	BMM	32bit	32bit
BMM-32	The 32bit BMM design	BMM	32bit	32bit
BMM-64	The 64bit BMM design	BMM	32bit	32bit
BMMS-32	Fine-grained parallelism BMM-32	BMM	32bit	32bit
BMMS-64	Fine-grained parallelism BMM-64	BMM	32bit	32bit
BMM-32B	Bit input/output BMM-32	BMM	1bit	1bit
BMM-64B	Bit input/output BMM-64	BMM	1bit	1bit
BMMS-32B	Fine-grained parallelism BMM-32B	BMM	1bit	1bit
BMMS-64B	Fine-grained parallelism BMM-64B	BMM	1bit	1bit

unlike those in BMM, the 64-bits BConv versions outperform the 32-bits versions. In particular, our *BConv-64B* achieves 10× speedup over the cuDNN implementation on the latest Volta GPU when input/output_channel = 320. In case the input or output_channel is not a factor of 32/64, padding is required when performing the binarization. This may sacrifice some performance, especially when the image_size and/or filter_size is large. However, most of the modern DNN input/outpu_channels are factors of 32/64 (e.g., AlexNet, VGG, ResNet, etc).

6.3 SBNN Inference

We evaluate our SBNN inference framework on three different datasets using three different network structures, as listed in Table 3. For MLP on MNIST, the input image for the first layer is also binarized in both training and inference. This explains the slight accuracy degradation on our BNN training accuracy (98.6% to 97.4%). For VGG on Cifar10 and AlexNet on ImageNet, the input image of the first layer is in full-precision, but the weight is binarized, similar to BWN. This is due to accuracy consideration. Since there are only 3 input channels, the overhead of the first layer adopting full-precision input is not very high. Divergent from the original BNN design, the final output layer in our implementation is also binarized for both input and weight, followed by a batch normalization. We find the BN here, also cannot be aggregated as a threshold comparison as before (there is no further binarization), is vital to the high training accuracy for large dataset. Overall, after a training period of 2000, 2500 and 500 epochs for MLP, VGG and AlexNet, we achieve comparable or even superior BNN training accuracy with state-of-the-art BNN and full-precision DNN training accuracy. This demonstrates that our intra- and inter-layer fusion techniques will not affect the accuracy of the training process.

Table 4 and 5 show the inference performance results on Tesla-P100 and V100. *SBNN-32/64* refer to the 32/64-bit single-kernel SBNN inference. *SBNN-32/64-Fine* refers to the implementation that the FC layers are using fine-grained 32-bit BMM (i.e., *BMMS-32B*). The TensorFlow results which simulate BNN in 32-bit single-precision floating-point (FP32) via cuBLAS and cuDNN on the same GPUs are used as reference (which is also the general scenario

Table 3: SBNN Inference Evaluation. "1024FC" refers to a fully-connected layer with 1024 neutrons. "2x128C3" refers to 2 convolution layer with 128 output channels and 3x3 filter. "MP2" refers to a 2x2 pooling layer with stride=2. "128C11/4" refers to a convolution layer with 128 output channels, 11x11 filter size and stride=4. "Input size" is in the format of *input_height*×*input_width*×*input_channels*. "Output" is the number of categories for the classification. "Ref" is short for references. "BNN" is the reference BNN training accuracy. "Our BNN" is the SBNN training accuracy we obtained. "Full-Precision" is the reference FP32 training accuracy.

Dataset	Ref	Network	Ref	Network Structure	Input Size	Output	BNN	Our BNN	Full-Precision
MNIST	[27]	MLP	[12]	1024FC-1024FC-1024FC-1024FC	$28x28x1$	10	98.6% [12]	97.4%	99.1% [12]
Cifar-10	[23]	VGG	[11]	(2x128C3)-MP2-(2x256C3)-MP2-(2x512C3)-MP2-(3x1024FC)	$32x32x3$	10	89.9% [12]	90.2%	90.9% [2]
ImageNet	[13]	AlexNet	[24]	(128C11/4)-MP2-(256C5)-MP2-(2x384C3)-(256C3)-MP2-(3x4096FC)	$224x224x3$	1000	75.7/46.1% [2]	71.2/44.7%	80.2/56.6% [2]

Table 4: SBNN Inference Performance on NVIDIA Pascal-based P100 GPU

	MNIST-MLP				Cifar10-VGG				ImageNet-AlexNet			
Schemes	**Raw Latency**	**Speedup**	**Throughput**	**Speedup**	**Raw Latency**	**Speedup**	**Throughput**	**Speedup**	**Raw Latency**	**Speedup**	**Throughput**	**Speedup**
SBNN-32	0.277ms	1678×	3.16×10^6	1090×	1.502ms	76.3×	2.53×10^4	34.5×	3.238ms	674×	1615.3	6.6×
SBNN-32-Fine	0.084ms	5533×	1.20×10^6	416×	1.298ms	88.3×	2.47×10^4	41.5×	2.778ms	786×	1530.8	6.2×
SBNN-64	1.007ms	462×	1.19×10^5	41.2×	1.556ms	73.7×	2.81×10^4	38.4×	13.985ms	156×	1449.2	5.9×
SBNN-64-Fine	0.076ms	6115×	4.06×10^6	1410×	0.611ms	188×	3.73×10^4	50.8×	1.528ms	1429×	1910.3	7.8×
TensorFlow	464.74ms	1.0×	2.89×10^3	1×	114.65ms	1×	733.13	1×	2183.62ms	1×	245.83	1×

Table 5: SBNN Inference Performance on NVIDIA Volta-based V100 GPU

	MNIST-MLP				Cifar10-VGG				ImageNet-AlexNet			
Schemes	**Raw Latency**	**Speedup**	**Throughput**	**Speedup**	**Raw Latency**	**Speedup**	**Throughput**	**Speedup**	**Raw Latency**	**Speedup**	**Throughput**	**Speedup**
SBNN-32	0.183ms	1266×	4.39×10^6	1010×	0.994ms	132×	5.15×10^4	60.4×	2.226ms	523×	4.19×10^3	14×
SBNN-32-Fine	0.04ms	5790×	3.32×10^6	762×	0.833ms	157×	5.10×10^4	59.8×	1.576ms	739×	3.95×10^3	13.2×
SBNN-64	0.896ms	259×	1.88×10^5	43×	1.395ms	94×	4.22×10^4	49.5×	9.134ms	128×	2.87×10^3	9.6×
SBNN-64-Fine	0.04ms	5790×	8.98×10^6	2060×	0.466ms	281×	5.78×10^4	67.9×	0.979	1190×	4.40×10^3	14.7×
TensorFlow	231.6ms	1×	4.36×10^3	1×	131.09ms	1×	851.49	1×	1164.95ms	1×	298.44	1×

for BNN algorithm design and validation), since to the best of our knowledge, there is no mature GPU-based BNN implementation. As can be seen in Table 4 and 5, our single-kernel SBNN inference achieves up to 6115× speedup over the TensorFlow baseline on P100 with a small dataset (i.e., MNIST) and 1429× speedup with a large dataset (i.e., ImageNet) for a single image BNN inference. An inference delay of 40μs for a small network and less than 1ms for a large network are sufficient to satisfy the timing constraints for most latency-critical DNN applications in the real world. We can also observe that the throughput (i.e., images per second) improvement does not match the significant raw latency reduction. In fact, for AlexNet on ImageNet, the throughput improvement is only about an order, implying that GPU is extremely underutilized with non-batched inference in current TensorFlow. Our design dramatically enhances GPU utilization (but not in a traditional way as no floating-point computation is involved). Comparing among various implementations, SBNN-64-Fine outperforms the others, demonstrating the optimal performance among all conditions. This is mainly because: (1) SBNN-64 has a better register data reuse thus a higher computation-to-memory ratio than SBNN-32; (2) For small-batched inference, extracting more fine-grained parallelism from the GPU kernel to feed all GPU SMs is clearly more crucial than better data reuse or cache efficiency.

7 RELATED WORK

The emergence of deep neural networks has brought significant challenges and opportunities for application-specific system and architecture design [5, 41]. Since BNNs were first proposed in 2016 [12, 39], the subsequent research around BNNs can be classified into two major categories: *algorithms* and *implementations*.

Algorithms. The main objective is to improve BNN training accuracy [12, 32, 33, 39, 42, 47], especially for large datasets. XNOR-Net [39] applied BNNs on ImageNet, reporting top-1 accuracies of up to 51.2% for full binarization and 65.5% for partial binarization. DoReFa-Net [47] reported best-case ImageNet top-1 accuracies of 43% for full and 53% for partial binarization. Another work [42] reported 46.6%/71.1% top-1/5 accuracy for AlexNet on ImageNet by taking advantage of their new observation on learning-rate, activation function, and regularizer. Recent works such as ABC-Net [32], Self-Binarizing [3] and BNN+[2] reported even better BNN training accuracy.

Implementation. The goal is to build high-performance BNNs to satisfy the stringent real-time inference constraints of the delay-critical applications in cloud and edge domains, and with as little area and energy cost as possible [19, 31, 33, 35, 43, 46]. The majority of these studies are FPGA-based [31, 35, 43, 46] due to FPGA's flexible and powerful bit-manipulation capability. Recently, a CPU-based BNN implementation was proposed [19]. This work relies on bit-packing and AVX/SSE vector instructions to derive good bit computation performance. However, their implementation mainly focuses on BMM; BConv is converted to BMM through the conventional flatten or unfold approach with expensive pre/post-processing for padding. They do not discuss intra- and inter-layer fusion for whole network optimization. Although recent works have mainly focused on optimizing individual layers, we believe that inter-layer and whole-network optimization are becoming more essential. For example, the TensorFlow XLA [26] and Tensor Comprehensions [44] recently compiled the entire neural network graphs at once, performing various transformations and achieving 4x speedup over manually tuned individual layers. Our design follows this emerging research trend by merging all the layers into a single GPU kernel, achieving significant inference speedups.

8 CONCLUSION

In this paper, we propose binarized-soft-tensor-core to build strong bit-manipulation capability for modern economy-of-scale general-purpose GPU architectures. We use BSTC to accelerate the bit functions such as bit-packing, bit-matrix-multiplication and bit-convolution of binarized-neural-networks. To further accelerate BNN inference performance, we propose intra- and inter-layer fusion techniques that can merge the whole BNN inference process as a single GPU kernel. For non-batched BNN inference tasks, our design demonstrates more than three orders of magnitude latency reduction compared to the state-of-the-art full-precision simulated BNN inference on GPUs.

REFERENCES

[1] Saman Ashkiani, Martin Farach-Colton, and John D Owens. 2018. A dynamic hash table for the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 419–429.

[2] Anonymous authors. 2019. BNN+: Improved Binary Network Training. *Under reivew for ICLR-19* (2019).

[3] Anonymous authors. 2019. Self-Binarizing Networks. *Under reivew for ICLR-19* (2019).

[4] Carlo Baldassi and Riccardo Zecchina. 2018. Efficiency of quantum vs. classical annealing in nonconvex learning problems. *Proceedings of the National Academy of Sciences* (2018), 201711456.

[5] Tal Ben-Nun and Torsten Hoefler. 2018. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *arXiv preprint arXiv:1802.09941* (2018).

[6] Eli Ben-Sasson, Matan Hamilis, Mark Silberstein, and Eran Tromer. 2016. Fast multiplication in binary fields on GPUS via register cache. In *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 35.

[7] Benjamin Block, Peter Virnau, and Tobias Preis. 2010. Multi-GPU accelerated multi-spin Monte Carlo simulations of the 2D Ising model. *Computer Physics Communications* 181, 9 (2010), 1549–1556.

[8] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft.

[9] Chih-Hong Cheng, Georg Nührenberg, Chung-Hao Huang, and Harald Ruess. 2018. Verification of Binarized Neural Networks via Inter-neuron Factoring. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 279–290.

[10] 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).

[11] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*. 3123–3131.

[12] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. Ieee, 248–255.

[14] Wenbin Fang, Mian Lu, Xiangye Xiao, Bingsheng He, and Qiong Luo. 2009. Frequent itemset mining on graphics processors. In *Proceedings of the fifth international workshop on data management on new hardware*. ACM, 34–42.

[15] Francesco Fusco, Michail Vlachos, Xenofontas Dimitropoulos, and Luca Deri. 2013. Indexing million of packets per second using GPUs. In *Proceedings of the 2013 conference on Internet measurement conference*. ACM, 327–332.

[16] Angus Galloway, Graham W Taylor, and Medhat Moussa. 2017. Attacking Binarized Neural Networks. *arXiv preprint arXiv:1711.00449* (2017).

[17] Google. 2018. TensorFlow: Adding a New Op. http://www.tensorflow.org/extend/adding_an_op

[18] 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.

[19] Yuwei Hu, Jidong Zhai, Dinghua Li, Yifan Gong, Yuhao Zhu, Wei Liu, Lei Su, and Jiangming Jin. [n. d.]. BitFlow: Exploiting Vector Parallelism for Binary Neural Networks on CPU. ([n. d.]).

[20] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In *Advances in Neural Information Processing Systems*. 4107–4115.

[21] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[22] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[23] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. 2014. The CIFAR-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html* (2014).

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

[25] Jaeha Kung, David Zhang, Gooitzen van der Wal, Sek Chai, and Saibal Mukhopadhyay. 2018. Efficient object detection using embedded binarized neural networks. *Journal of Signal Processing Systems* 90, 6 (2018), 877–890.

[26] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled. *TensorFlow Dev Summit* (2017).

[27] Yann LeCun, Corinna Cortes, and CJ Burges. 2010. MNIST handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist* 2 (2010).

[28] Yann LeCun, LD Jackel, Leon Bottou, A Brunot, Corinna Cortes, JS Denker, Harris Drucker, I Guyon, UA Muller, Eduard Sackinger, et al. 1995. Comparison of learning algorithms for handwritten digit recognition. In *International conference on artificial neural networks*, Vol. 60. Perth, Australia, 53–60.

[29] Ang Li, Weifeng Liu, Linnan Wang, Kevin Barker, and Shuaiwen Song. 2018. Warp-Consolidation: A Novel Execution Model for GPUs. In *Proceedings of the 2018 International Conference on Supercomputing (ICS)*.

[30] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. 2017. Locality-aware cta clustering for modern gpus. *ACM SIGOPS Operating Systems Review* 51, 2 (2017), 297–311.

[31] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. 2018. FP-BNN: Binarized neural network on FPGA. *Neurocomputing* 275 (2018), 1072–1086.

[32] Xiaofan Lin, Cong Zhao, and Wei Pan. 2017. Towards accurate binary convolutional neural network. In *Advances in Neural Information Processing Systems*. 345–353.

[33] Bradley McDanel, Surat Teerapittayanon, and HT Kung. 2017. Embedded binarized neural networks. *arXiv preprint arXiv:1709.02260* (2017).

[34] Nina Narodytska, Shiva Prasad Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. 2017. Verifying properties of binarized deep neural networks. *arXiv preprint arXiv:1709.06662* (2017).

[35] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. 2016. Accelerating binarized neural networks: comparison of FPGA, CPU, GPU, and ASIC. In *Field-Programmable Technology (FPT), 2016 International Conference on*. IEEE, 77–84.

[36] NVIDIA. 2018. CUDA Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide

[37] Sreepathi Pai, Matthew J Thazhuthaveetil, and Ramaswamy Govindarajan. 2013. Improving GPGPU concurrency with elastic kernels. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 407–418.

[38] Martín Pedemonte, Enrique Alba, and Francisco Luna. 2011. Bitwise operations for GPU implementation of genetic algorithms. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*. ACM, 439–446.

[39] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 525–542.

[40] Jingkuan Song. 2017. Binary generative adversarial networks for image retrieval. *arXiv preprint arXiv:1708.04150* (2017).

[41] Ion Stoica, Dawn Song, Raluca Ada Popa, David Patterson, Michael W Mahoney, Randy Katz, Anthony D Joseph, Michael Jordan, Joseph M Hellerstein, Joseph E Gonzalez, et al. 2017. A berkeley view of systems challenges for ai. *arXiv preprint arXiv:1712.05855* (2017).

[42] Wei Tang, Gang Hua, and Liang Wang. 2017. How to train a compact binary neural network with high accuracy?. In *AAAI*. 2625–2631.

[43] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 65–74.

[44] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[45] Kefu Xu, Wenke Cui, Yue Hu, and Li Guo. 2013. Bit-parallel multiple approximate string matching based on GPU. *Procedia Computer Science* 17 (2013), 523–529.

[46] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. 2017. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 15–24.

[47] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160* (2016).