RETHINKING FLOATING POINT OVERHEADS FOR MIXED PRECISION DNN ACCELERATORS

Hamzah Abdel-Aziz [1] **Ali Shafiee** [1] **Jong Hoon Shin** [1] **Ardavan Pedram** [1] **Joseph H. Hassoun** [1]

ABSTRACT

In this paper, we propose a mixed-precision convolution unit architecture which supports different integer and floating point (FP) precisions. The proposed architecture is based on low-bit inner product units and realizes higher precision based on temporal decomposition. We illustrate how to integrate FP computations on integer-based architecture and evaluate overheads incurred by FP arithmetic support. We argue that alignment and addition overhead for FP inner product can be significant since the maximum exponent difference could be up to 58 bits, which results into a large alignment logic. To address this issue, we illustrate empirically that at least 8 bits of alignment logic are required to maintain inference accuracy. We present novel optimizations based on the above observations to reduce the FP arithmetic hardware overheads. Our empirical results, based on simulation and hardware implementation, show significant reduction in FP16 overhead. Over a typical mixed precision implementation, the proposed architecture achieves area improvements of up to 25% in TFLOPS/mm^2 and up to 46% in TOPS/mm^2 with power efficiency improvements of up to 40% in TFLOPS/W and up to 63% in TOPS/W.

1 INTRODUCTION

Deep Neural Networks (DNNs) have shown tremendous success in modern AI tasks such as computer vision, natural language processing, and recommender systems (LeCun et al., 2015). Unfortunately, DNNs success comes at the cost of significant computational complexity (e.g., energy, execution time, etc.). Therefore, DNNs are accelerated on specialized hardware units (DNN accelerators) to improve both performance and energy efficiency (Jouppi et al., 2017; ten, 2017; Reuther et al., 2019). DNN accelerators may utilize quantization schemes to reduce DNNs memory footprint and computation time (Deng et al., 2020). A typical quantization scheme compresses all DNN layers into the same low-bit integer, which can be sub-optimal, as different layers have different redundancy and feature distributions (Wang et al., 2019; Wu et al., 2018a). On the other hand, mixed precision quantization schemes assign different precisions (i.e., bit width) for different layers and show remarkable improvement over uniform quantization (Song et al., 2020; Wang et al., 2019; Chu et al., 2019; Cai et al., 2020). Therefore, mixed-precision quantization schemes (Song et al., 2020; Wang et al., 2019; Chu et al., 2019; Cai et al., 2020) or hybrid approaches where a few layers are kept in FP and the rest are quantized to integer are considered to maintain FP32-level accuracy (Zhu et al., 2016; Venkatesh et al., 2017).

Half precision floating point (FP16) and custom floating point data types (e.g., bfloat16 (Abadi et al., 2016)) are adopted for inference and training in several cases when quantization is not feasible (online learning, private dataset, supporting legacy code, etc.). They could reduce memory footprint and computation by a factor of two, without significant loss of accuracy and they are often obtained by just downcasting the tensors. FP16 shows remarkable benefits in numerous DNN training applications where FP16 is typically used as the weights and activation data type and FP32 is used for accumulation and gradient update (Micikevicius et al., 2017; Jia et al., 2018; Ott et al., 2019).

Data precision varies significantly from low-bit integer to FP data types (e.g., INT4, INT8, FP16, etc.) within or across different DNN applications. Therefore, mixed-precision DNN accelerators that support versatility in data types are crucial and sometimes mandatory to exploit the benefit of different software optimizations (e.g., low-bit quantization). Moreover, supporting versatility in data types can be leveraged to trade off accuracy for efficiency based on the available resources (Shen et al., 2020). Typically, mixed-precision accelerators are designed based on low precision arithmetic units, and higher precision operation can be supported by fusing the low precision arithmetic units temporally or spatially.

The arithmetic operations of DNNs boil down to dot prod-

[1]Samsung Semiconductor, Inc. San Jose, CA. Correspondence to: Hamzah Abdel-Aziz <hamzah.a@samsung.com>.

Proceedings of the 4th MLSys Conference, San Jose, CA, USA, 2021. Copyright 2021 by the author(s).

ucts as the basic operations. Typically, dot products are implemented either by temporally exploiting a multiply-accumulate (MAC) unit in time or by spatially utilizing an inner product (IP) unit with multipliers followed by an adder tree. The multiplier and adder precisions (i.e., bit widths) are the main architectural decisions in implementing the arithmetic unit of the dot product operation. The multiplier precision is a key factor for the architecture performance, and its efficiency for both IP and MAC based arithmetic units. For example, a higher multiplier precision (e.g., 8×8) limits the benefit of lower-bit (e.g., INT4) quantization. On the other hand, while lower precision multipliers are efficient for low-bit quantization, they incur excessive overhead for the addition units. Therefore, multipliers precision is decided based on the common case quantization bit width. The adder tree precision in integer IP based architecture matches the multiplier output bit width. Thus, they can improve energy efficiency by using smaller adder and sharing the accumulation logic. However, in multiply-and-accumulate (MAC) based architectures (Chen et al., 2016), adders precision are larger than products bit width to match the accumulator size. Their overheads are more pronounced in low-power accelerators with low-precision multipliers optimized for low-bit quantized DNNs.

Implementing a floating point IP (FP-IP) operation requires alignment of the products before summation, which require large shift units and adders. Theoretically, the maximum range of alignment between FP16 products requires shifting the products up to 58-bit. Thus, the adder tree would impose an additional 58 bits in its input precision. Such alignments are only needed for FP operations and appear as significant power and area overhead for INT operations, especially when IP units are based on low-precision multipliers.

In this paper, we explore the design space trade-offs of IP units that support both FP and INT based convolution. We make a case for a dense low-power convolution unit that intrinsically supports INT4 operations. Furthermore, we go over the inherent overheads to support larger INT and FP operations. We consider INT4 for two main reasons. First, this data type is the smallest type supported in several modern architectures that are optimized for deep learning (e.g., AMD MI50 (amd), Nvidia Turing architecture (Kilgariff et al., 2018) and Intel Spring Hill (Wechsler et al., 2019)). Second, recent research on quantization report promising results for 4-bit quantization schemes (Fang et al., 2020; Jung et al., 2019; Nagel et al., 2020; Choukroun et al., 2019; Banner et al., 2019b; Wang et al., 2019; Choi et al., 2018; Zhuang et al., 2020). In spite of this, the proposed optimization is not limited to INT4 case and can be applied for other cases (e.g., INT8) as we discuss in Section 4.

The contributions of the paper are as follows:

1. We investigate approximated versions of FP-IP operation with limited alignments capabilities. We derive the mathematical bound on the absolute error and conduct numerical analysis based on DNN models and synthetic values. We postulate that approximate FP-IP can maintain the GPU-based accuracy if it can align the products by at least 16 bits and 27 bits, for FP16 and FP32 accumulators, respectively.

2. We demonstrate how to implement large alignments using smaller shift units and adders in multiple cycles. This approach decouples software alignment requirements from the underlying IP unit implementation. It also enables more compact circuits at the cost of FP task performance.

3. Instead of running many IP units synchronously in one tile, we decompose them into smaller clusters. This can isolate FP-IP operations that need a large alignment and limits the performance degradation to one cluster.

4. We study the design trade-offs of our architecture.

The proposed architecture, implemented in standard 7nm technology, can achieve up to 25% in TFLOPS/mm^2 and up to 46% in TOPS/mm^2 in area efficiency and up to 40% in TFLOPS/W and up to 63% in TOPS/W in power efficiency.

The rest of this paper is organized as follows. In Section 2, we present the proposed architecture of mixed-precision inner product unit (IPU) and explain how it can support different data types including FP16. In section 3, we first review the alignment requirement for FP16 operations and offer architecture optimization to reduce FP16 overheads. Section 4 goes over our methodology and discusses the empirical results. In Section 5, we review related work. We conclude the paper in Section 6.

2 Mixed-precision Inner Product Unit

To support different data types and precisions, we use a fine-grain convolution unit that can run INT4 intrinsically and realize larger sizes temporally. We consider INT4 as the default common case since several recent research efforts are promoting INT4 quantization schemes for efficient inference (Jung et al., 2019; Nagel et al., 2020). However, the proposed architecture can be applied to other cases such as INT8 as the baseline.

Figure 1 shows the building blocks of the proposed mixed-precision n-input IPU, which is based on 5b$\times$5b sign multipliers. The proposed IPU allows computing INT4 IPU multiplications, both signed or unsigned, in a single cycle. In addition, larger precision operations can be computed in multiple ***nibble iterations***. The total number of nibble iterations is the multiplication of the number of nibbles of

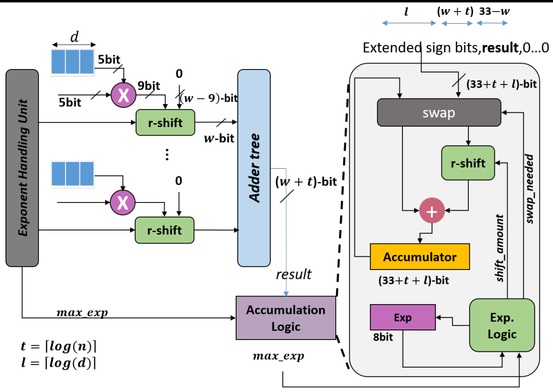


Figure 1. Microarchitecture of the proposed mixed-precision IPU data path with n inputs and w-bit IPU precision.

the two multipliers operands. Products are passed to a local right shift unit which used in FP-mode for alignment, and the shifted outputs are connected to an adder tree. The adder tree results are fed to the accumulator. In the next two subsection, we illustrate the mircoarchitecture in details for both INT and FP modes; respectively.

2.1 INT Mode

The IPU is based on INT4 and the computation of higher INT precision is based on ***nibble iterations***. For example, if the multipliers operands are INT8 and INT12, six nibble iterations are required to complete INT8 by INT12 multiplication for a single dot product operation. The local shift amount is always 0 since there is no alignment required in INT mode. The result of the adder tree is concatenated with $(33 - w)$ bits of zeros on the right side and always fed to the accumulator shift unit through the swap unit. The amount of shift depends on the significance of the nibble operands. For instance, suppose N_k refers to the nibbles of a number (i.e., N_0 is the least significant nibble), the amount of shift for the result of IPU operation of nibble N_i and N_j for the first and the second operands is $4 \times ((K_a - i - 1) + (K_b - j - 1))$, where K_a and K_b are the total number of nibbles for operand a and b, respectively. The accumulator can add up to $n \times d$ multiplications, where n is the number of IPU inputs and d is the maximum number of times IPU can accommodate accumulation without overflow. In this scenario, the accumulator size should be at least $33 + t + l$, where $l = \lceil \log_2 d \rceil$. In INT mode, exp and max_exp in Figure 1 are considered zero.

2.2 FP Mode

In FP-mode, the mantissa multiplication is computed similar to INT12 IPU but with the following additional operations.

Converting numbers: Let's define the magnitude of FP number as 0.mantissa for subnormal and 1.mantissa for normal FP numbers. We also call it the signed magnitude when the sign is considered. Suppose $M[11:0]$ is the 12-bit signed magnitude for an FP16 number, it can be converted to the following three 5-bit nibbles: $N_2 = \{M_{11} - M_7\}$, $N_1 = \{0, M_6 - M_3\}$, and $N_0 = \{0, M_2 - M_0, 0\}$. This decomposition introduces a zero in the least significant position of N_0. Since the FP-IP operation relies on right shifting and truncation to align the products, the implicit left shift of operands preserves more accuracy.

Local alignment: The product results should be aligned with respect to the maximum exponent of all products (see Appendix A for more details). Therefore, each of the multiplier outputs is passed to a local right shift unit that receives the shift amount from ***the exponent handling unit*** (**EHU**). The EHU computes the product exponents by doing the following steps, in order: (1) element-wise summation of the operands' unbiased exponents, (2) computing the maximum of the product exponents, and (3) computing the alignment shift amounts as the difference between all the product exponents and the maximum exponent. Note that, a single EHU can be shared between multiple IPUs to amortize its overhead (i.e., multiplexed in time between IPUs), since a single FP-IP operation consists of multiple nibble iterations with the same exponent computation.

The range of the exponent for FP16 products is $[-28, 30]$, thus the exponent difference (i.e., the right shift amount) between two FP16 products can be up to 58-bit. In general, the bit width of the product increases based on the amount of right shift (i.e., alignment with the max exponent). However, due to the limited precision of the accumulator, an approximate computation is sufficient where the product alignment can be bounded and truncated to a smaller bit width. We define this width as ***the IPU precision*** and use it to parametrize IPUs. The IPU precision is also the maximum amount of local right shift as well as the bit-width of the adder tree. We quantify the impact of the IPU precision on the computation accuracy in Section 3.1.

The accumulator operations: During the computation for one output, the IP accumulator keeps two values: accumulator's exponent and its non-normalized signed magnitude. Once all input vector pairs are computed and accumulated, the result in the accumulator is normalized and reformatted to the standard representation (i.e., FP16 or FP32).

The details of the accumulation logic are depicted in the right side of Figure 1. The accumulator has a $(33 + t + l)$-bit register and a right shift unit (see Figure 1 for t and l definitions). Therefore, the register size allows up to 33 bits of right shift, which is sufficient to preserve accuracy as discussed in Section 3.1.

In contrast to the INT mode, where the accumulator's right shift only shifts by $4k$ ($k \in 1, 2, .., 6$), the FP-IP accumulators may right shift by any number between 0 and $33 + t + l$. The shift amount is computed in exponent logic and is equal to $4 \times ((3 - i - 1) + (3 - j - 1)) + |max\_exp - exp|$, where

i, and j are input nibble indices, exp is the accumulator's exponent value and max_exp is the adder tree exponent (i.e., the max exponent). A swap operation followed by a right shift is applied whenever a left shift is needed, hence, a separate left shift unit is not needed. In other words, the swap operation is triggered only when $max\_exp > exp$.

With respect to exp, the accumulator value is a signed $(33 + t + l)$ bits fixed point number with $(3 + t + l)$ bits in integer positions and 30 bits in fraction positions. Note that, the accumulator holds an approximate value since the least significant bits are discarded and its bit-width is provisioned for the practical size of IPUs. Before writing back the result to memory, the result is rounded to its standard format (i.e., FP16 or FP32).

For the rest of this paper, we define an $IPU(w)$ as an inner product unit with 5-bit signed multipliers, a w-bit adder tree, and local right shifters that can shift and truncate multipliers' outputs by up to w bits. We refer to w as the IPU's adder tree precision or **IPU precision** for brevity. In general, the result of $IPU(w)$ computation might be inaccurate, as only the w most significant bits of the local shifter results are considered. However, there are exceptions:

Proposition 1 *For $IPU(w)$, truncation is not needed and the adder tree result is exact if the amount of alignments of all the products are smaller than $w - 9$. We refer to $w - 9$ as* **the safe precision of the IPU**.

It is clear that the area and power overhead increase as the IPU precision increases (See Section 4.2). The maximum required precision is determined by the software accuracy requirement and the accumulator precision (See Section 3.1).

3 Optimizing Floating Point Logic

In this section, we address the overhead of alignment and adder tree precision by, first, evaluating the minimum shift and adder precision required to preserve the accuracy (Section 3.1) for both FP16 and FP32 accumulators. Based on the evaluation, we propose optimization methods to implement FP IPUs with relatively smaller shifters and adders (Section 3.2 and Section 3.3).

3.1 Precision Requirement for FP16

As shown in Section 2, an FP-IP operation is decomposed into multiple nibble iterations. In a typical implementation, the multiplier's output of each iteration requires large alignment shifting and the adder tree has high precision inputs. However, this high precision would be discarded due to the limited precision of the accumulator (FP16 or FP32), hence, an approximated version of FP-IP alignment can be used without significant loss of accuracy. Figure 2 shows the pseudocode for the approximate FP-IP operation

```
Approximate_nibble_iteration(A, B, i, j, precision)
%Input1: $A =< a_0, \dots, a_{n-1} >$
%Input2: $B =< b_0, \dots, b_{n-1} >$
% $a_k^i$ and $b_k^j$ are the $i-th$ and $j-th$ nibble of $a_k$ and $b_k$
$a_k^i =$ nibble(i, signed magnitude($a_k$)) % $a_k^i$ has 5 bits
$b_k^j =$ nibble(j, signed magnitude($b_k$)) % $b_k^i$ has 5 bits
% assume bias is already removed from exponents
1: for k in 0 to $n-1$
2:   $c_k \leftarrow \exp(a_k) + \exp(b_k)$ % $-28 \le c_i \le 30$

3: $max \leftarrow maximum(c_k)$
4: for k in 0 to $n-1$
5:     $d_k = a_k^i \times b_k^j$ % $d_k$ has 9bits (including sign)
6:     $d_k = d_k \ll (precision - 9)$ % $d_k$ has precision bits
       %if $(max - c_i) > precision \rightarrow d_k = 0$
7:     $d_k = d_k \gg (max - c_i)$
%Sum has precision + log(n) bits
8: $Sum = (d_0 + \cdots + d_{n-1})$
9: return (max, sum)
```

```
FP-IP(A, B, acc, exp, precision)
% (acc, exp) is current accumulator's non_normalized
% signed magnitude and its exponent value.
%Input1: $A =< a_0, \dots, a_{n-1} >$
%Input2: $B =< b_0, \dots, b_{n-1} >$
1: for i in 0 to 3
2:   for j in 0 to 3
3:     (max, sum) =
4:         approximate_nibble_iteration(A, B, i, j, precision)
           % update logic is explained in Fig 4.b
5:         update_accumulator(max, sum, acc, sum)
```

Figure 2. Pseudocode for the approximate version of nibble iteration (top) and FP-IP operation with the approximate nibble iteration method (bottom). Precision is the IPU precision.

customized for our nibble-based IPU architecture. The approximate FP-IP computes only most significant $precision$ bits of the products (Lines 5-7). The $precision$ parameter allows us to quantify the absolute error.

Theorem 1 *For FP-IP with n pairs of FP16 inputs, the absolute error due to $approx\_nibble\_iteration(i, j, precision)$, called $abs\_error(i, j)$ is no larger than $225 \times 2^{(4\times(i+j)-22)} \times 2^{max-precision} \times (n-1)$, where max is the maximum exponent of all the products in the FP operation.*

Proof: Due to space limitations, we only provide an outline of the proof. The highest error occurs when, except for one product, all $n - 1$ others are shifted $precision$ to the right, and thus appear as errors. For maximum absolute error, these products should all have the same sign and have the maximum operand (i.e., 15). Hence their product would be $15 \times 15 = 225$. The term $2^{(4\times(i+j))}$ is applied for proper alignment based on nibble significance. The term 2^{-22} is needed, since each FP product has 3-bit integer and 22-bit fraction positions, with respect to its own exponent.

Remark 1 *Iterations of the most significant nibbles (i.e., largest $i + j$) have the highest significant contributions to the absolute error.*

The FP-IP operation is the result of nine approximate nibble

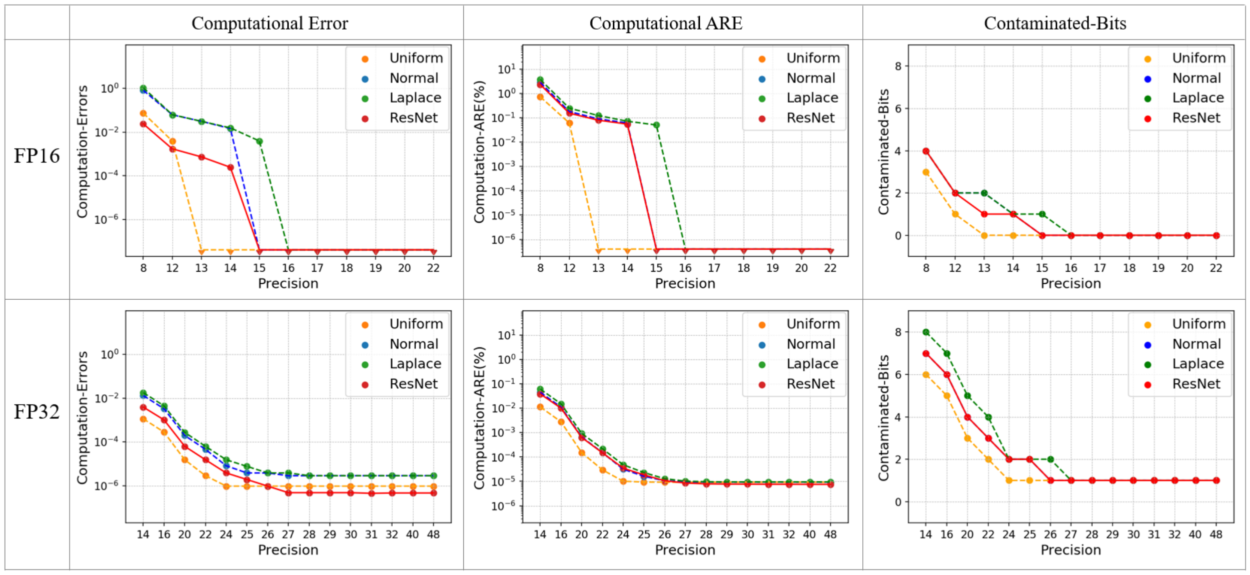


Figure 3. Left to Right: Absolute error, percentage of absolute relative error (ARE), and the number of contaminated bits for different distributions and different accumulators: FP16(top) and FP32(bottom). The first two error graphs in each row use log-scale Y-axis.

iterations added into the accumulator. However, only 11 or 24 most significant bits of the accumulated result are needed for FP16 or FP32 accumulators, respectively. Unfortunately, the accumulator is non-normalized and its leading non-zero position depends on the input values. As a result, it is not possible to determine a certain precision for each approximate nibble iteration to guarantee any loss of significance. Therefore, we use numerical analysis to find the proper shift parameters. In our analysis, we consider both synthetic input values and input values sampled from tensors found in ResNet-18 and ResNet-50 convolution layers. We consider Laplace and Normal distributions to generate synthetic input vectors, as they resemble the distribution of DNN tensors (Park et al., 2018) and uniform distributions for the case that tensor is re-scaled, as suggested for FP16-based training (Micikevicius et al., 2017). In our analysis, we consider 1M samples generated for our three distributions and 5% data samples of ResNet-18 and ResNet-50 convolution layers. For different IPU precisions, we measure the median for three metrics: absolute computation error, absolute relative error (in percentage) compared with FP32 CPU results, and the number of contaminated bits. The number of contaminated bits refers to the number of different bits between the result of approximated FP-IP and the FP32 CPU computation. Figure 3 includes the error analysis plots for both FP16 and FP32 accumulator cases. Based on this analysis, we found that both the relative and the absolute errors are less than 10^{-6} for 16-bit IPU precision in FP16 case. Moreover, the median number of contaminated bits is zero (mean = 0.5). For FP32 accumulator case, both errors drop to less than 10^{-5} for $IPUprecision \geq 26$-bit. However, the minimum median value of the number of contaminated bits starts at 27b IPU precision. We conclude that ***in order to maintain FP32 CPU accuracy, FP16 FP-IP operations require at least 16b and 27b IPU precision for accumulating into FP16 and FP32, respectively***.

We also evaluate the impact of IPU precision on Top-1 accuracy of ResNet-18 and ResNet-50 on ImageNet (He et al., 2016). We observe that, when the FP16 uses IPU precision of 12 or more, it maintains the same accuracy (i.e., Top-1 and Top-5) as FP32 CPU for all batches. IPU precision of 8-bit also shows no significant difference with respect to the final average accuracy compared to CPU computation. However, we observe some accuracy drops of up to 17% for some batches (batch_size = 256), and some accuracy improvements up to 17% for other batches. This improvement may be just a random behavior, or because lower precisions may have a regularization effect as suggested by (Courbariaux et al., 2015b). At any rate and despite these results, 8-bit IPU precision is not enough for all CNN inference due to the fluctuation in the accuracy for individual batches.

3.2 Multi-Cycle IPU

As we showed in Section 3.1, approximate nibble iteration requires 27-bit addition and alignment to maintain the same accuracy as CPU implementations for FP32 accumulation. As we illustrate in Section 4, the large shifter and adder take a big portion of area breakdown of an IPU and an overhead when running in the INT mode. In order to maintain both high accuracy with low hardware overhead, we propose using multiple cycles when a DNN requires large alignment, using *multi-cycle IPU(w)*, (MC-IPU(w)), where w refers to the adder tree precision (i.e., input bit width). Hence, designers can consider lower MC-IPU precision, when the convolution tile is used more often in the INT than the FP mode.

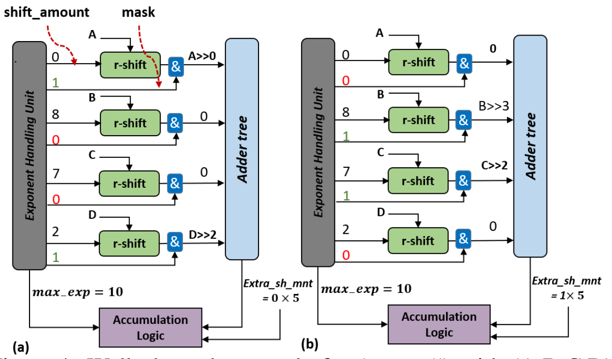
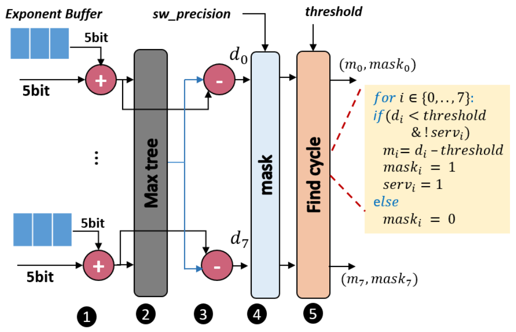


Figure 4. Walk-through example for ($sp = 5$) with (A,B,C,D) as magnitudes and (10,2,3,8) as exponents. The exponent can be written as (0,-8,-7,-2) with respect to $max\_exp = 10$. (a) First cycle: MC-IPU only executes products A and D since their right shift is in $P_0 = [0, 5]$ (b) Second cycle: MC-IPU computes products B and C as their right shift is in $P_1 = [5, 10]$.

MC-IPU relies on Proposition 1 that if all the alignments are smaller than the safe precision (sp), summation is exact. Otherwise, the MC-IPU performs the following steps to maintain accurate computation. First, it assigns products into different sets, such that products whose required shift amounts belong to $[k \times sp, (k+1) \times sp]$ are in set k (P_k). Second, all products in set k are added in the same cycles and all other products are masked. Notice that all products P_k in set k require at least $k \times sp$ shifting. Thus MC-IPU decomposes the shift amount into parts: (1) $k \times sp$ that is applied after the adder tree and (2) the remaining parts that is applied locally. Since the remaining parts are always smaller than sp, they can be done with small local shift units without any loss in accuracy (Proposition 1).

Figure 4 illustrates a walk-through example of MC-IPU(14) with $sp = 5$. In this example, we denote the products in summation as A, B, C, and D with exponents 10, 2, 3, and 8, respectively. Thus, the maximum exponent is $max\_exp = 10$. Before the summation, each product should be aligned ($w.r.t. max\_exp$) by the right shift amount of 0, 8, 7, and 2, accordingly. The alignment and summation happens in two cycles as follows: In the first cycle, A and D are added after zero- and two- bit right shifts, respectively. Notice that, the circuit has extra bitwise AND logic to mask out input B and C in this cycle. In the second cycle, B and C are added and they need eight- and seven- bit right shifts, respectively. While the local shifter can only shift up to five bits accurately, we perform the right shift in two steps by locally shifting by $(8-5)$ and $(7-5)$ bits, followed by five bit shifts of the adder tree result.

In general, the Multi-Cycle IPU imposes three new overheads to IPUs: (1) Bitwise AND logic per multiplier; (2) updating shifting logic, where the shared shifting amount would be given to the accumulation logic (*extra_sh_mnt* in Figure 4), for each cycle; and (3) modifications to the EHU unit. The EHU unit for MC-IPU is depicted in Figure 5. It consists of five stages. The first stage receives activation



```
for i ∈ {0,..,7}:
if ($d_i$ < threshold
        & ! $serv_i$)
  $m_i$= $d_i$ − threshold
  $mask_i$ = 1
  $serv_i$ = 1
else
  $mask_i$ = 0
```

Figure 5. EHU Data path for MC-IPUs.

and weight exponents and adds them together to calculate the product exponents. In the second and third stages, the maximum exponent and its differences from each product exponent are computed. In the fourth stage, the differences that exceed the required software precision are masked (see Section 3.1). The first four stages are common for both IPUs and MC-IPUs. However, the last stage is only needed for MC-IPU and might be called multiple times, depending on the required number of cycles for MC-IPU. This stage keeps a single bit for each product to indicate whether that product has been aligned or not ($serv_i$ in Figure 5). For the non-aligned ones, this stage checks the exponent difference value with a threshold. The threshold value equals $(k+1) \times sp$ in cycle k (see the code in Figure 5).

3.3 Intra-Tile IPU clustering

In the previous Section, we show how the MC-IPU can run the FP inner product by decomposing it into nibble iterations and computing each iteration in one or multiple cycles. In a convolution tile that leverages MC-IPUs, the number of cycles per iteration depends on two factors: (1) the precision of the MC-IPU (i.e., adder tree bit width). (2) the maximum alignment needed in all the MC-IPUs in the convolution tiles. When a MC-IPU in the convolution tile requires a large alignment, it stalls others so they stay synchronized.

Therefore, we introduce another design factor by grouping MC-IPUs in smaller clusters and running them independently. This way, if one MC-IPU requires multiple cycles, it stalls only the MC-IPUs in its own cluster. To run clusters independently, each cluster should have its own local input and output buffers. The output buffer is used to synchronize the result of different clusters before writing them back into memory. Notice that, inputs are broadcasted from memory to all clusters' input buffers and would stop broadcasting if any input buffer is full, which may stall some clusters or the entire tile. Based on a statistical simulation, we estimate that buffers with a depth of two is sufficient to keep all clusters synchronized and balanced with a probability of 0.98.

4 METHODOLOGY AND RESULTS

In this section, we illustrate the top level architecture and experiment setup. Then, we evaluate the hardware overhead

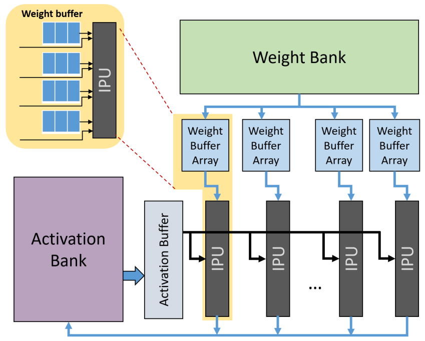


Figure 6. Top level architecture of DNN accelerator.

and performance impact of our proposed architecture. We also compare our architecture with related work.

4.1 Top Level Architecture

We consider a family of high-level architectures designed by IP-based tiles. IP-based tiles are crucial for energy efficiency, especially when low-precision multipliers are used. An IP-based convolution tile consists of multiple IPUs and each IPU is assigned to one output feature map (OFM) (i.e., unrolling in output channel (K)). All IPUs share the same input vectors that come from different channels in the input feature map (IFMs) (i.e., unrolling in the input channel dimension (C)). As depicted in Figure 6, the data path of the accelerator tile consists of the following components: (1) Inner Product Unit: an array of multipliers that feeds into an adder tree. The adder tree's result is accumulated into the partial sum accumulator. (2) Weight Bank: contains all the filters for the OFMs that are assigned to the tile. (3) Weight buffer: contains a subset of filters that are used for the current OFMs. Each multiplier has a fixed number of weights, which is called the depth of the weight buffer. Weight buffer are only needed for weight stationary (WS) (Chen et al., 2016) architecture and is either implemented with flip-flops, register files, or small SRAMs. The number of elements per weight buffer determines the output/partial bandwidth requirements. (4) Activation Bank: contains the current activation inputs, partial, and output tensors. (5) Activation Buffer: serves as a cache for the activation bank and broadcast inputs to the IPUs. It also helps to reduce activation bank bandwidth.

We consider, two types of tiles, big and small, based on INT4 multipliers. Both tiles are weight stationary with weight buffer depth of 9B. The big and small tiles are unrolled $(16, 16, 2, 2)$ and $(8, 8, 2, 2)$ in (C, K, H_o, W_o) dimensions. We consider these two tiles because they offer different characteristics while achieving high utilization. The IPUs in the big tile have twice as many multipliers as in the small tile (16 vs. 8). The 16-input IPUs have smaller accumulator overhead but larger likelihood of multiple cycles alignment as compared to 8-input IPUs. For comparison, we consider two baselines: Baseline1 and Baseline2 for the small and the big tiles, respectively. Each baseline has four tiles with a 38b wide adder tree per IPU. Hence, these baselines do not need MC-IPU (Section 3.2) and IPU clustering (Sectoin 3.3) and they can achieve (1 TOPS, 113 GFLOPS) and (4 TOPS, 455 GFLOPS), respectively (OP is a 4×4 MAC).

The performance impact of the proposed designs (i.e., MC-IPUs and clustering the IPUs) depends on the distribution of inputs. We developed a cycle-accurate simulator that models the number of cycles for each convolution layer. The simulation parameters include the input and weight tensors. The simulator receives, the number of tiles, the tile dimension (e.g., (8, 8, 2, 2) for the small tiles), and the number of clusters per tile. We simulate Convolution layers as our tiles are customized to accelerate them. In addition, we assume an ideal behavior for the memory hierarchy to single out the impact of our designs. In reality, non-CNN layers and system-level overhead can impact the overall result. Moreover, the area and power efficiency improvements might decline due to the limitations of DRAM bandwidth and SRAM capacity (Pedram et al., 2017). Such scenarios are beyond the scope of our analysis.

In the simulation analysis, we use data tensors from ResNet (He et al., 2016) and InceptionV3 (Szegedy et al., 2016). We consider four study cases which are: (1) ResNet-18 forward path, (2) ResNet-50 forward path, (3) InceptionV3 forward path, and (4) ResNet-18 backward path of training. In our benchmarks, we consider at least 16b and 28b software precision (Section 3.1) that is required for FP16 and FP32 accumulation to incur no accuracy loss.

4.2 Hardware Implementation Results

In order to evaluate the impact of FP overheads, we implemented our designs in SystemVerilog and synthesized them using Synopsys DesignCompiler with $7nm$ technology libraries (DC). We consider 25% margin and $0.71V$ Voltage for our synthesis processes. Figure 7 illustrates the breakdown of area and power for a small and big tile. We also include a design point without FP support, shown as INT in Figure 7. In addition, we consider one design with a 38-bit adder tree, similar to NVDLA (NVD), for our baseline configuration. We highlight the following points in Figure 7 as follows: (1) By just dropping the adder tree precision from 38 to 28 bits, which is the minimum precision to maintain CPU-level accuracy for FP32 accumulations (see Section 3.1), the area and power are reduced by 17% and 15% for 16-input and 8-input MC-IPU tiles, respectively. (2) We can reduce the adder tree precision even further at the cost of running alignment in multiple cycles. The tile area can be reduced by up to 39% when reducing adder tree precision to 12 bits. (3) In comparison with INT only IPU, MC-IPU(12) can support FP16 with a 43% increase in area.

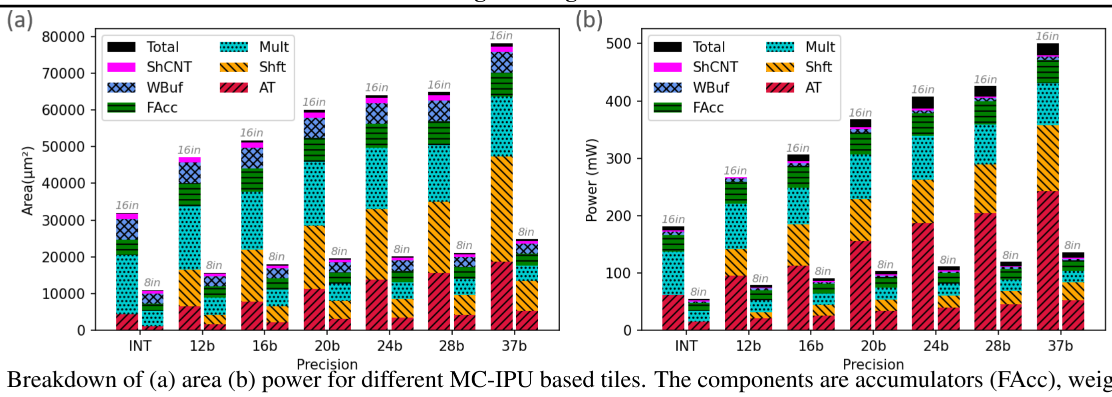


Figure 7. Breakdown of (a) area (b) power for different MC-IPU based tiles. The components are accumulators (FAcc), weight buffers (WBuf), EHUs (ShCNT), multipliers (MULT), local shifters (Shft), and adder trees (AT).

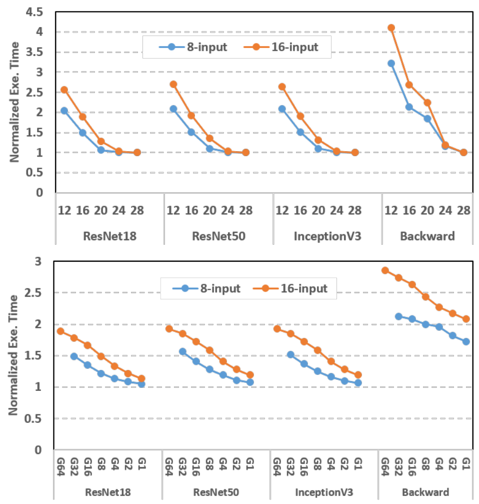


Figure 8. (a) Impact of different precision on the performance of MC-IPUs. Backward refers to the back propagation error in ResNet-18.(b) Impact of cluster size on the performance for MC-IPU(16).

4.3 Performance Result

FP16 operations with FP16 accumulations: As shown in Section 3.1, there is no need for more than 16-bit precision for FP16 accumulation. Therefore, IPUs with a 16b or greater adder tree precision take exactly one cycle per nibble iteration. However, MC-IPU(12) may require multiple-cycle alignment execution, which causes performance loss. Compared to Baseline1 (Baseline2), when MC-IPU(12)s are used, the performance drops by 47% (50%), on average, when no IPU clustering is applied (Section 3.3). If we choose a cluster of size one, (i.e., MC-IPUs perform independently), the performance drop is 26% (38%), compared to Baseline1 (Baseline2).

FP16 operations with FP32 accumulations: As we showed in Section 3.1, FP32 accumulation requires 28-bit IPU precision. Thus, an MC-IPU with precision less than 28-bit might require multiple cycles, causing performance loss. Figure 8 shows the normalized execution time for different precision values for the forward path of ResNet-18, ResNet-50, and InceptionV3 as well as the backward path of ResNet-18. We observe that all epochs have a similar trend, thus we only report data for Epoch 11. In this figure, we present two sets of numbers: ones for the tiles with 8-input MC-IPUs, normalized to Baseline1 and one for the tiles with 16-input MC-IPUs, normalized to Baseline2.

According to Figure 8 (a), the execution time can increase dramatically when small adder trees are used and 28-bit IPU precision is required. The increase in the latency can be more than 4× for a 12b adder tree in the case of computation of back propagation (backprop). Intuitively, increasing the adder bit width reduces the execution time. In addition, since 8-input MC-IPUs have fewer products, it is less likely that they need multiple cycles. Thus, 8-input MC-IPUs (Baseline1) outperform 16-input MC-IPUs (Baseline2). We also observe that backprop computations have more dynamic range and more variance in the exponents.

To evaluate the effect of clustering, we fix the adder tree bit-width to 16 bits and vary the number of MC-IPUs per cluster. Figure 8 (b) shows the efficiency of MC-IPU clustering, where the x-axis and y-axis represents the cluster size and the execution of 8-input (16-input) MC-IPUs(16) normalized to Baseline1 (Baseline2), respectively. According to this figure, smaller clusters can reduce the performance degradation significantly due to multi-cycling in the case of forward computation using 8-input MC-IPUs. However, in 16-input cases, there is at least 12% loss even for cluster of size 1. Backward data has more variation and, even for one MC-IPU per cluster, there is at least 60% increase in the execution time. The reason for such behavior can be explained using the histogram of exponent difference of 8-input MC-IPUs for Resnet-18 in the forward and backward paths, illustrated in Figure 9. As shown in this figure, the forward path exponent differences are clustered around zero and only 1% of them are larger than eight. On the other hand, the products of backward computations have a wider distribution than forward computations.

4.4 Overall Design Trade-offs

Figure 10(a,b) visualize the power and area efficiency design spaces for INT vs. FP modes, respectively. In these

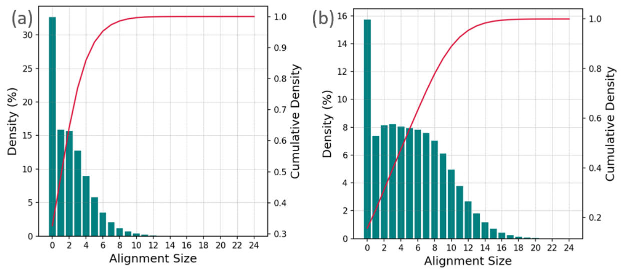


Figure 9. The distribution of exponent difference ($Max.exp - exp$, or alignment size) of ResNet-18 training computations. (a) forward-propagation, (b) back-propagation.

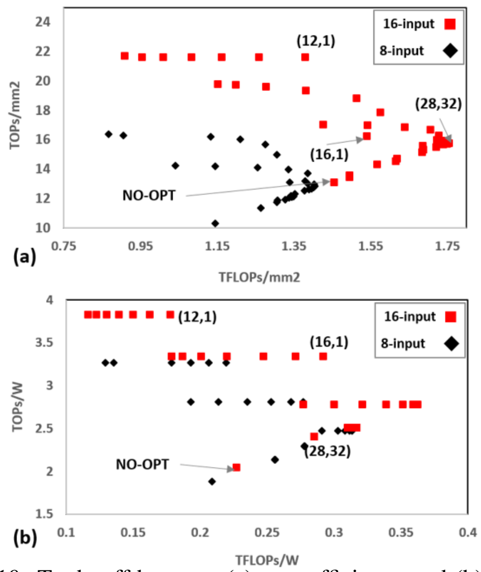


Figure 10. Trade-off between (a) area efficiency and (b) power efficiency. Each design point (p,c) represents tiles with the p-bit adder tree MC-IPUs with c MC-IPUs per cluster (only labeled for 16-input MC-IPUs). NO-OPT is Baseline2.

figures, we consider the average effective throughput, using our simulation results, for FP throughput values. The numbers associated with some of the design points refer to the ordered pair of MC-IPU precision and the cluster size. For designs with 8-input (16-input), approximation can boost power efficiency of INT and FP mode by 14% (17%) and improve area efficiency by 17.8% (20%). The overall improvement is the combination of all the optimizations. The two design points (12,1) and (16,1) are on the power efficiency Pareto optimal curve. For example, the design points with one MC-IPU per cluster and 12-bit (16-bit) IPU precision, achieve 14% (25%) in TFLOPS/mm^2 and up to 46% (46%) in TOPS/mm^2 with our 8-input (16-input) IPU architectures over typical mixed precision implementation in area efficiency and up to 63% (40%) in TFLOPS/W and up to 74% (63%) in TOPS/W in power efficiency.

4.5 Sensitivity Analysis

In this paper, we mainly consider INT4 as the common case, however, it is still possible to consider different precision as the baseline for different targeted quantization schemes, data types, or application domain (i.e., edge vs cloud). Therefore, we evaluate the performance of the proposed approach using four designs with different multiplier precisions. The first design (MC-SER) is based on serial multipliers (i.e., 12×1) similar to Stripes (Judd et al., 2016) but MC-SER supports FP16 using the proposed optimizations. Note that, FP16 operation requires at least 12 cycles per inner product in the case of 12×1 multiplier. The second design (MC-IPU4) is optimized for INT4 as discussed earlier and it is based on 4×4 multipliers. The third design (MC-IPU84) is optimized for INT8 for activation and INT4 for weights, and it is based on 8×4 multipliers. The fourth design (MC-IPU8) is optimized for INT8 for activation and weights, and it is based on 8×8 multipliers. We also compare against other mixed precision designs including: NVDLA, typical FP16 implementation and mixed precision INT-based designs which do not support FP16. We show the comparison between these designs in terms of TOPS/mm^2 and TOPS/W for different types of operations as shown in Table 1. The results show that MC-IPU mitigates the overhead of the local shift units and adder trees when FP16 is required. This overhead becomes relatively more significant as the precision of the multiplier decreases and the optimization benefit decreases as we increase the baseline multiplier precision. However, designs with high multiplier baseline (e.g., 8×8) limits the benefits of low-bit (e.g., INT4) software optimization.

5 Related Work

Previous studies on CNN accelerators exploit two major approaches to their ALU/FPU datapath, MAC-based (Jouppi et al., 2017; Chen et al., 2016; Gao et al., 2017; Lu et al., 2017; Kim et al., 2016; Venkataramani et al., 2017; Yazdanbakhsh et al., 2018) and Inner Product-based (Chen et al., 2014; NVD; Eth; Venkatesan et al., 2019; Shao et al., 2019; Liu et al., 2016; Kwon et al., 2018). Unfortunately, most of these approaches exploit INT-based arithmetic units and rely on quantization to convert DNNs from FP to INT.

The INT-based arithmetic unit can also support different bit widths. Multi-precisions of operands for INT-based architectures has been already addressed in both spatial and temporal decomposition. In the spatial decomposition approach, a large arithmetic unit is decomposed into multiple finer grain units (Sharma et al., 2018; Camus et al., 2019; Mei et al., 2019; Moons et al., 2017). Since the Pascal architecture, Nvidia GPUs implement spatial decomposition via DP4A and DP2A instructions, where INT32 units are decomposed into 4-input INT8 or 2-input INT16 inner products. This approach is different than ours, as we support FP16 and use inner product rather than MAC units. On the other hand, the temporal decomposition approach performs the sequences of fine-grain operations in time to mimic a coarse-grain operation. Our approach resembles this ap-

Table 1. TOPs/W and TOPs/mm^2 for different multipliers (MUL) and adder trees (ADT) precision. A and W are activation and weight precisions

	MC-SER	MC-IPU4	MC-IPU84	MC-IPU8	NDVLA	FP16	INT8	INT4
ADT	16b	16b	20b	23b	36b	36b	16b	9b
MUL	12×1	4×4	8×4	8×8	8×8	12×12	8×8	4×4
$A \times W$	$TOPS/mm^2$ or $TFLOPS/mm^2$							
4×4	5.5	18.8	14.3	11.4	9.7	6.9	18.5	30.6
8×4	5.5	9.4	14.3	11.4	9.7	6.9	18.5	15.3
8×8	2.8	4.7	7.2	11.4	9.7	6.9	18.5	7.7
$FP16 \times FP16$	0.9	1.6	1.8	5.4	4.9	6.9	–	–
$A \times W$	$TOPS/W$ or $TFLOPS/W$							
4×4	1.4	3.3	2.4	1.8	1.5	0.9	2.8	5.6
8×4	1.4	1.7	2.4	1.8	1.5	0.9	2.8	2.8
8×8	0.7	0.8	1.2	1.8	1.5	0.9	2.8	1.4
$FP16 \times FP16$	0.2	0.3	0.3	0.8	0.7	0.9	–	–

proach with 4-bit operations as the finest granularity. Other works that use this approach prefer lower precision (Judd et al., 2016; Lee et al., 2019; Eckert et al., 2018; Sharify et al., 2018). Temporal decomposition has also been used to avoid ineffectual operations by dynamically detecting fine-grain zero operands and discarding the operation (Delmas et al., 2018; Albericio et al., 2017; Sharify et al., 2019). In contrast to us, these approaches do not support FP16 operands. In addition, we only discuss the dense architectures; however, the fine-grain building block can also be used for sparse approaches. We leave this for future.

The approaches listed above rely on quantization schemes to convert FP32 DNNs to integer-based ones (Krishnamoorthi, 2018; Lee et al., 2018; Nagel et al., 2019; Zhuang et al., 2018; Wang et al., 2018; Choi et al., 2018; Hubara et al., 2017). These schemes are added to DNN software frameworks such as TensorFlow Lite. Recent advancements show that 8-bit post-training quantization (Jacob et al., 2018) and 4-bit retaining-based quantization can achieve almost the same performance as FP32 (Jung et al., 2019). However, achieving high accuracy is less trivial for shallow networks with 2D Convolution operations (Howard et al., 2017; Sheng et al., 2018). There is also work to achieve high accuracy at lower precision (Zhu et al., 2016; Zhuang et al., 2019; Banner et al., 2019a; Choukroun et al., 2019; Courbariaux et al., 2015a; Zhou et al., 2016; Zhang et al., 2018; Rastegari et al., 2016). A systematic approach to find the correct precision for each layer has been shown in (Wang et al., 2019; Dong et al., 2019; Cai et al., 2020). Dynamic multi-granularity for tensors is also considered as a way of computation saving (Shen et al., 2020). Several quantization schemes have been proposed for training (Wu et al., 2018b; Banner et al., 2018; Das et al., 2018; De Sa et al., 2018; Park et al., 2018).

Recent industrial products support mixed-precision arithmetic, including Intel's Spring Hill (Wechsler et al., 2019), Huawei's DaVinci (Liao et al., 2019), Nvidia's TensorCore (ten, 2017), Google's TPU (Jouppi et al., 2017), and Nvidia's NVDLA (NVD). While most of these architectures use FP16, BFloat16 and TF32 are selected for the large range in some products (Abadi et al., 2016; tf3). Using the current structure, our approach can support both BFloat16 and TF32 by modifying the EHU to support 8-bit exponents and using only four nibble iterations. Similar to our approach, NVDLA supports FP-IP operations. In contrast, it decomposes an FP16 unit into two INT8 unit spatially. Additionally, NVDLA does not allow computations of FP-IP operations with different-type operands. It also does not support INT4 and provides a large precision (38-bit) for its adder tree, which we demonstrate is not efficient. Our proposed architecture optimization can also applied to spatial decomposition and it is orthogonal to the decomposition scheme (i.e., temporal, serial, spatial).

There are also proposals to optimize the microarchitecture of FP MACs or IPUs. LMA is a modified FP units that leverages Kulisch accumulation to improve FMA energy efficiency (Johnson, 2018). An FMA unit with fixed point accumulation and lazy rounding is proposed in (Brunie, 2017). A 4-input inner product for FP32 is proposed in (Sohn & Swartzlander, 2016). The spatial fusion for FMA is presented in (Zhang et al., 2019). Finally, a mixed precision FMA that supports INT MAC operations is presented in (Zhang et al., 2020). As opposed to the proposed architecture, most of these efforts do not support INT-based operations or are optimized for FP operation with high overhead that hinder the performance of the INT operations.

6 Conclusion

In this paper, we explored the design space of the structure of an inner product based convolution tile and identified the challenges to support the floating-point computation and its overhead. Further, from the software perspective, we investigated the minimum requirements for achieving the targeted accuracy. We proposed novel architectural optimizations that mitigate the floating-point logic overheads in favor of boosting computation per area for INT-based operations. We showed that for an IPU based on low-precision multipliers, adder and alignment logic overhead due to supporting FP operations is substantial. We conclude that the differences between product exponents are typically smaller than eight bits allowing the use of smaller shift units in FPUs.

References

Synopsys design compiler. URL https://www.synopsys.com/.

Arm ethos n77. URL https://developer.arm.com/ip-products/processors/machine-learning/arm-ethos-n/ethos-n77.

Nvidia deep learning accelerator (nvdla). URL http://nvdla.org/.

Amd radeon instinct™ mi50 accelerator. URL https://www.amd.com/system/files/documents/radeon-instinct-mi50-datasheet.pdf.

Nvidia a100 tensor core gpu architecture. URL https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

NVIDIA TESLA V100 GPU ARCHITECTURE. Technical report, Nvidia, 08 2017.

Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

Albericio, J., Delmás, A., Judd, P., Sharify, S., O'Leary, G., Genov, R., and Moshovos, A. Bit-pragmatic deep neural network computing. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 382–394, 2017.

Banner, R., Hubara, I., Hoffer, E., and Soudry, D. Scalable methods for 8-bit training of neural networks. In *Advances in neural information processing systems*, pp. 5145–5153, 2018.

Banner, R., Nahshan, Y., and Soudry, D. Post training 4-bit quantization of convolutional networks for rapid-deployment. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 7950–7958. Curran Associates, Inc., 2019a.

Banner, R., Nahshan, Y., and Soudry, D. Post training 4-bit quantization of convolutional networks for rapid-deployment. In *Advances in Neural Information Processing Systems*, pp. 7950–7958, 2019b.

Brunie, N. Modified fused multiply and add for exact low precision product accumulation. In *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, pp. 106–113, 2017.

Cai, Y., Yao, Z., Dong, Z., Gholami, A., Mahoney, M. W., and Keutzer, K. Zeroq: A novel zero shot quantization framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 13169–13178, 2020.

Cambier, L., Bhiwandiwalla, A., Gong, T., Nekuii, M., Elibol, O. H., and Tang, H. Shifted and squeezed 8-bit floating point format for low-precision training of deep neural networks, 2020.

Camus, V., Enz, C., and Verhelst, M. Survey of precision-scalable multiply-accumulate units for neural-network processing. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 57–61, 2019.

Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., et al. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622. IEEE, 2014.

Chen, Y.-H., Emer, J., and Sze, V. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 44(3):367–379, 2016.

Choi, J., Wang, Z., Venkataramani, S., Chuang, P. I.-J., Srinivasan, V., and Gopalakrishnan, K. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

Choukroun, Y., Kravchik, E., Yang, F., and Kisilev, P. Low-bit quantization of neural networks for efficient inference. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pp. 3009–3018, 2019.

Chu, T., Luo, Q., Yang, J., and Huang, X. Mixed-precision quantized neural network with progressively decreasing bitwidth for image classification and object detection. *arXiv preprint arXiv:1912.12656*, 2019.

Courbariaux, M., Bengio, Y., and David, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 28*, pp. 3123–3131. Curran Associates, Inc., 2015a.

Courbariaux, M., Bengio, Y., and David, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pp. 3123–3131, 2015b.

Das, D., Mellempudi, N., Mudigere, D., Kalamkar, D., Avancha, S., Banerjee, K., Sridharan, S., Vaidyanathan, K., Kaul, B., Georganas, E., et al. Mixed precision training of convolutional neural networks using integer operations. *arXiv preprint arXiv:1802.00930*, 2018.

De Sa, C., Leszczynski, M., Zhang, J., Marzoev, A., Aberger, C. R., Olukotun, K., and Ré, C. High-accuracy low-precision training. *arXiv preprint arXiv:1803.03383*, 2018.

Delmas, A., Judd, P., Stuart, D. M., Poulos, Z., Mahmoud, M., Sharify, S., Nikolic, M., and Moshovos, A. Bit-tactical: Exploiting ineffectual computations in convolutional neural networks: Which, why, and how. *arXiv preprint arXiv:1803.03688*, 2018.

Deng, L., Li, G., Han, S., Shi, L., and Xie, Y. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proceedings of the IEEE*, 108 (4):485–532, 2020.

Dong, Z., Yao, Z., Gholami, A., Mahoney, M. W., and Keutzer, K. Hawq: Hessian aware quantization of neural networks with mixed-precision. In *The IEEE International Conference on Computer Vision (ICCV)*, October 2019.

Drumond, M., Tao, L., Jaggi, M., and Falsafi, B. Training dnns with hybrid block floating point. In *Advances in Neural Information Processing Systems*, pp. 453–463, 2018.

Dumoulin, V. and Visin, F. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

Eckert, C., Wang, X., Wang, J., Subramaniyan, A., Iyer, R., Sylvester, D., Blaaauw, D., and Das, R. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 383–396. IEEE, 2018.

Fang, J., Shafiee, A., Abdel-Aziz, H., Thorsley, D., Georgiadis, G., and Hassoun, J. H. Post-training piecewise linear quantization for deep neural networks. In *European Conference on Computer Vision*, pp. 69–86. Springer, 2020.

Gao, M., Pu, J., Yang, X., Horowitz, M., and Kozyrakis, C. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 751–764, 2017.

Gustafson, J. L. and Yonemoto, I. T. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations*, 4(2):71–86, 2017.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

Hill, P., Jain, A., Hill, M., Zamirai, B., Hsu, C., Laurenzano, M. A., Mahlke, S., Tang, L., and Mars, J. Deftnn: Addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 786–799, 2017.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017. URL `http://arxiv.org/abs/1704.04861`.

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, 2018.

Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., Xie, L., Guo, Z., Yang, Y., Yu, L., et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

Johnson, J. Rethinking floating point for deep learning, 2018.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–12, 2017.

Judd, P., Albericio, J., Hetherington, T., Aamodt, T. M., and Moshovos, A. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12. IEEE, 2016.

Jung, S., Son, C., Lee, S., Son, J., Han, J.-J., Kwak, Y., Hwang, S. J., and Choi, C. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4350–4359, 2019.

Kilgariff, E., Moreton, H., Stam, N., and Bell, B. Nvidia turing architecture in-depth. 2018.

Kim, D., Kung, J., Chai, S., Yalamanchili, S., and Mukhopadhyay, S. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. *SIGARCH Comput. Archit. News*, 44(3): 380–392, June 2016. ISSN 0163-5964. doi: 10.1145/3007787.3001178. URL https://doi.org/10.1145/3007787.3001178.

Köster, U., Webb, T., Wang, X., Nassar, M., Bansal, A. K., Constable, W., Elibol, O., Gray, S., Hall, S., Hornof, L., et al. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In *Advances in neural information processing systems*, pp. 1742–1752, 2017.

Krishnamoorthi, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

Kwon, H., Samajdar, A., and Krishna, T. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ACM SIGPLAN Notices*, 53 (2):461–475, 2018.

LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *nature*, 521(7553):436–444, 2015.

Lee, J., Kim, C., Kang, S., Shin, D., Kim, S., and Yoo, H. Unpu: An energy-efficient deep neural network accelerator with fully variable weight bit precision. *IEEE Journal of Solid-State Circuits*, 54(1):173–185, 2019.

Lee, J. H., Ha, S., Choi, S., Lee, W.-J., and Lee, S. Quantization for rapid deployment of deep neural networks. *arXiv preprint arXiv:1810.05488*, 2018.

Liao, H., Tu, J., Xia, J., and Zhou, X. Davinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pp. 1–44. IEEE Computer Society, 2019.

Liu, S., Du, Z., Tao, J., Han, D., Luo, T., Xie, Y., Chen, Y., and Chen, T. Cambricon: An instruction set architecture for neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 393–405. IEEE, 2016.

Lu, J., Lu, S., Wang, Z., Fang, C., Lin, J., Wang, Z., and Du, L. Training deep neural networks using posit number system. *arXiv preprint arXiv:1909.03831*, 2019.

Lu, W., Yan, G., Li, J., Gong, S., Han, Y., and Li, X. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 553–564. IEEE, 2017.

Mei, L., Dandekar, M., Rodopoulos, D., Constantin, J., Debacker, P., Lauwereins, R., and Verhelst, M. Sub-word parallel precision-scalable mac engines for efficient embedded dnn inference. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 6–10, 2019.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

Moons, B., Uytterhoeven, R., Dehaene, W., and Verhelst, M. 14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 246–247, 2017.

Nagel, M., Baalen, M. v., Blankevoort, T., and Welling, M. Data-free quantization through weight equalization and bias correction. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1325–1334, 2019.

Nagel, M., Amjad, R. A., van Baalen, M., Louizos, C., and Blankevoort, T. Up or down? adaptive rounding for post-training quantization. *arXiv preprint arXiv:2004.10568*, 2020.

Ott, M., Edunov, S., Baevski, A., Fan, A., Gross, S., Ng, N., Grangier, D., and Auli, M. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038*, 2019.

Park, E., Yoo, S., and Vajda, P. Value-aware quantization for training and inference of neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 580–595, 2018.

Pedram, A., Richardson, S., Galal, S., Kvatinsky, S., and Horowitz, M. Dark memory and accelerator-rich system optimization in the dark silicon era. volume 34, pp. 39–50, 2017. doi: 10.1109/MDAT.2016.2573586.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pp. 525–542. Springer, 2016.

Reuther, A., Michaleas, P., Jones, M., Gadepally, V., Samsi, S., and Kepner, J. Survey and benchmarking of machine learning accelerators. *arXiv preprint arXiv:1908.11348*, 2019.

Shao, Y. S., Clemons, J., Venkatesan, R., Zimmer, B., Fojtik, M., Jiang, N., Keller, B., Klinefelter, A., Pinckney, N., Raina, P., et al. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 14–27, 2019.

Sharify, S., Lascorz, A. D., Siu, K., Judd, P., and Moshovos, A. Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6. IEEE, 2018.

Sharify, S., Lascorz, A. D., Mahmoud, M., Nikolic, M., Siu, K., Stuart, D. M., Poulos, Z., and Moshovos, A. Laconic deep learning inference acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 304–317, 2019.

Sharma, H., Park, J., Suda, N., Lai, L., Chau, B., Chandra, V., and Esmaeilzadeh, H. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 764–775, 2018.

Shen, J., Fu, Y., Wang, Y., Xu, P., Wang, Z., and Lin, Y. Fractional skipping: Towards finer-grained dynamic cnn inference. *arXiv preprint arXiv:2001.00705*, 2020.

Sheng, T., Feng, C., Zhuo, S., Zhang, X., Shen, L., and Aleksic, M. A quantization-friendly separable convolution for mobilenets. In *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, pp. 14–18, 2018.

Sohn, J. and Swartzlander, E. E. A fused floating-point four-term dot product unit. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 63(3):370–378, 2016.

Song, Z., Fu, B., Wu, F., Jiang, Z., Jiang, L., Jing, N., and Liang, X. Drq: Dynamic region-based quantization for deep neural network acceleration. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1010–1021. IEEE, 2020.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

Tambe, T., Yang, E.-Y., Wan, Z., Deng, Y., Reddi, V. J., Rush, A., Brooks, D., and Wei, G.-Y. Adaptivfloat: A floating-point based data type for resilient deep learning inference. *arXiv preprint arXiv:1909.13271*, 2019.

Venkataramani, S., Ranjan, A., Banerjee, S., Das, D., Avancha, S., Jagannathan, A., Durg, A., Nagaraj, D., Kaul, B., Dubey, P., and Raghunathan, A. Scaledeep: A scalable compute architecture for learning and evaluating deep networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pp. 13–26, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450348928. doi: 10.1145/3079856.3080244. URL https://doi.org/10.1145/3079856.3080244.

Venkatesan, R., Shao, Y. S., Zimmer, B., Clemons, J., Fojtik, M., Jiang, N., Keller, B., Klinefelter, A., Pinckney, N., Raina, P., Tell, S. G., Zhang, Y., Dally, W. J., Emer, J. S., Gray, C. T., Keckler, S. W., and Khailany, B. A 0.11 pj/op, 0.32-128 tops, scalable multi-chip-module-based deep neural network accelerator designed with a high-productivity vlsi methodology. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pp. 1–24, 2019.

Venkatesh, G., Nurvitadhi, E., and Marr, D. Accelerating deep convolutional networks using low-precision and sparsity. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2861–2865. IEEE, 2017.

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

Wang, P., Hu, Q., Zhang, Y., Zhang, C., Liu, Y., and Cheng, J. Two-step quantization for low-bit neural networks. In *Proceedings of the IEEE Conference on computer vision and pattern recognition*, pp. 4376–4384, 2018.

Wechsler, O., Behar, M., and Daga, B. Spring hill (nnp-i 1000) intel's data center inference chip. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pp. 1–12, 2019.

Wu, B., Wang, Y., Zhang, P., Tian, Y., Vajda, P., and Keutzer, K. Mixed precision quantization of convnets via differentiable neural architecture search. *arXiv preprint arXiv:1812.00090*, 2018a.

Wu, S., Li, G., Chen, F., and Shi, L. Training and inference with integers in deep neural networks. *arXiv preprint arXiv:1802.04680*, 2018b.

Yazdanbakhsh, A., Samadi, K., Kim, N. S., and Esmaeilzadeh, H. Ganax: A unified mimd-simd acceleration for generative adversarial networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 650–661, 2018.

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

Zhang, H., Chen, D., and Ko, S. Efficient multiple-precision floating-point fused multiply-add with mixed-precision support. *IEEE Transactions on Computers*, 68(7):1035–1048, 2019.

Zhang, H., Chen, D., and Ko, S. New flexible multiple-precision multiply-accumulate unit for deep neural network training and inference. *IEEE Transactions on Computers*, 69(1):26–38, 2020.

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

Zhu, C., Han, S., Mao, H., and Dally, W. J. Trained ternary quantization. *CoRR*, abs/1612.01064, 2016. URL `http://arxiv.org/abs/1612.01064`.

Zhuang, B., Shen, C., Tan, M., Liu, L., and Reid, I. Towards effective low-bitwidth convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7920–7928, 2018.

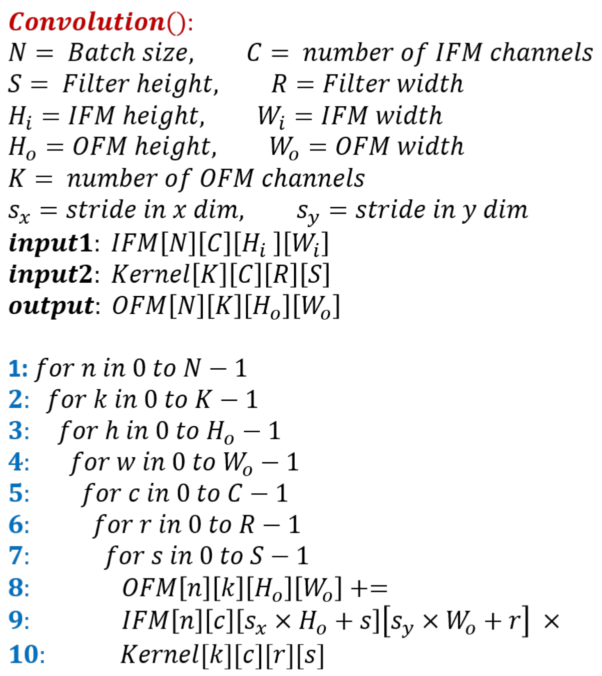
Zhuang, B., Shen, C., Tan, M., Liu, L., and Reid, I. Structured binary neural networks for accurate image classification and semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

Zhuang, B., Liu, L., Tan, M., Shen, C., and Reid, I. Training quantized neural networks with a full-precision auxiliary module. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1488–1497, 2020.

A Background

A.1 Convolution Layer Operation

A typical Convolution Layers (CL) operates on two 4D tensor as inputs (Input Feature Map (IFM) tensor and Kernel tensor) and results a 4D tensor (Output Feature Map (OFM) tensor). The element of IFMs and OFMs are called *pixels* or *activations* while the elements of Kernel are known as *weights*. Figure 11 shows simplified pseudocode for CL. The height and width of an OFM is typically determined by the height and width of IFMs, padding and strides. The three innermost loops (Lines 5-7) compute one output pixel and they can be realized as one or multiple inner product operations. The other four loops are independent, hence they can be implemented so to boost parallelism. More details are presented in (Dumoulin & Visin, 2016). A fully connected layer can be considered as a special case of convolution where the height and the width of IFM, OFM and Kernel are all equal to 1. Fully connected layers are used frequently in natural language processing and in the final layers of Convolutional Neural Networks (CNNs).



```
Convolution():
$N =$ Batch size,    $C =$ number of IFM channels
$S =$ Filter height,    $R =$ Filter width
$H_i =$ IFM height,    $W_i =$ IFM width
$H_o =$ OFM height,    $W_o =$ OFM width
$K =$ number of OFM channels
$s_x =$ stride in x dim,    $s_y =$ stride in y dim
input1: $IFM[N][C][H_i][W_i]$
input2: $Kernel[K][C][R][S]$
output: $OFM[N][K][H_o][W_o]$

1: for n in 0 to $N-1$
2:  for k in 0 to $K-1$
3:   for h in 0 to $H_o-1$
4:    for w in 0 to $W_o-1$
5:     for c in 0 to $C-1$
6:      for r in 0 to $R-1$
7:       for s in 0 to $S-1$
8:        $OFM[n][k][H_o][W_o]$ +=
9:        $IFM[n][c][s_x \times H_o + s][s_y \times W_o + r] \times$
10:       $Kernel[k][c][r][s]$
```

Figure 11. Pseudocode of a convolution layer.

A.2 Floating-Point Representation

Typical floating-point (FP) numbers are an IEEE standard to represent real numbers (876, 2019). DNNs take advantage of floating point arithmetic for training and highly accurate inference tasks. In general, an FP number is represented with three parts: (sign, exponent, and mantissa), which have (1,5,10), (1,8,23), (1,8,7), and (1,8,10) for FP16, FP32, Google's BFloat (BFloat16) (Abadi et al., 2016) and Nvidia's TensorFloat32 (TF32) (tf3).

Table 2. Different types of FP numbers. ($exp \neq 0$, $exp \neq 1...1$). $bias = 15(127)$ for FP16 (FP32).

type	(sgn, exp, man)	Value
zero	$(sgn, 0...0, 0...0)$	zero
INF	$(sgn, 1...1, 0...0)$	$\pm$ infinity
NaN	$(sgn, 1...1, man)$	$man \neq 0$, Not-a-Number
normal	(sgn, exp, man)	$(-1)^s \times 2^{exp-bias} \times 1.man$
subnormal	$(sgn, 0...0man)$	$(-1)^s \times 2^{-bias+1} \times 0.man$

For IEEE standard FP, the (sign, exponent, and mantissa) parts are used to decode five types of FP numbers as shown in Table 2. We define the ***magnitude*** as 0.mantissa for subnormal numbers and 1.mantissa for normal numbers. We also call it the ***signed magnitude*** when signed values are considered.

For deep learning applications, the inner product operations can be realized in two ways: (1) by iteratively using fused-multiply-add (FMA) units, i.e., performing $A \times B + C$ or (2) by running multiple inner product operations in parallel. In the latter case, the inputs would be two vectors $\langle a_0, \ldots, a_{n-1} \rangle$ and $\langle b_0, \ldots, b_{n-1} \rangle$ and the operation results in one scalar output. In order to keep the most significant part of the result and guarantee an absolute bound on the computation error, the products are summed by aligning all the products relative to the product with the maximum exponent. Figure 12 shows the required steps, assuming there is neither INF nor NaN in the inputs. The result has two parts: an exponent which is equal to the maximum exponent of the products, and a signed magnitude part which is the result of the summation of the aligned products.

The range of the exponent for FP16 numbers is [-14,15], hence, the range of the exponent for the product of two FP16 number is [-28,30]. The product result also has up to 22 bits of mantissa before normalization. This means that the accurate summation of such numbers requires 80-bit wide adders (58+22=80). However, smaller adders might be enough depending on the accuracy of the accumulators. For example, FP32 accumulators may keep only 24 bits of the result's sign magnitude. Therefore, it is highly unlikely that the least significant bits in the 80-bit addition contribute to the 24 bit magnitude of the accumulator and an approximate version of this operation would be sufficient. We will discuss the level of approximation in Section 3.1.

B Hybrid DNNs and Customized FP

The temporal INT4-based decomposition allows the proposed architecture to support different data types and pre-

```
FP Inner Product(A, B)
%Input1: $A = < a_0, \dots, a_{n-1} >$
%Input2: $B = < b_0, \dots, b_{n-1} >$
% $\exp(a_i) = a_i$'s exponent $-$ bias
% $\exp(b_i) = b_i$'s exponent $-$ bias
1: $a'_i$ = signed magnitude $(a_i)$ % $a'_i$ has 12 bits
2: $b'_i$ = signed magnitude $(b_i)$ % $b'_i$ has 12 bits

3: for $i$ in $0$ to $n-1$:
4:   $c_i \leftarrow \exp(a_i) + \exp(b_i)$ % $-28 \le c_i \le 30$

5: $max \leftarrow \max_i(c_i)$ % $0 \le i < n$

6: for $i$ in $0$ to $n-1$:
7:   $d_i = a'_i \times b'_i$ %$d_i$ has 23 bits
8:   $d_i = d_i \gg (max - c_i)$ % $0 \le (max - c_i) \le 58$

9: $sum = (d_0 + \cdots + d_{n-1})$ % sum has $80 + \log(n)$ bits

% exp(result) = max
% signed magnitude(result) = sum
%normalize back to FP
10: return normalize(max, sum)
```

Figure 12. Pseudocode for FP-IP operation (FP16). In a hardware realization, the loops would be parallel. Note, $exp(x) = x'sexponent - bias + 1$ for subnormal numbers but we omit it for simplicity.

cisions per operand per DNNs' layer. In the case that at least one of the operands is FP, the IPU runs in the FP mode. Depending on the input data types, the convolution results would be accumulated in a large INT or non-normalized FP register, which should be converted back to the next layer precision (INT or FP16 type). The conversion unit is not part of the IPU and thus not in the scope of this paper.

The proposed architecture can also support custom FP format, as we mentioned in Section A.2, BFloat16 and TF32 have 8-bit exponents. We can support these types with two modifications. (i) The EHU should support 8-bit exponents and (ii) larger shift units and adders might be needed.

Beside FP16 and BFloat16, there are some efforts to find the most concise data type for DNN applications. Flexpoint is a data type at the tensor level, where the all the tensor elements share an exponent and are 2s complement numbers (Köster et al., 2017). The same concept is used in (Drumond et al., 2018; Cambier et al., 2020). Some studies shows how to train using shared exponent and FP. Deft-16 is introduced to reduce memory bandwidth for FP32 training (Hill et al., 2017). Posit introduces a new field, called regime, to increase the range of numbers (Gustafson & Yonemoto, 2017; Lu et al., 2019). Other studies show how to train using shared exponent and FP. Deft-16 is introduced to reduce memory bandwidth for FP32 training (Hill et al., 2017). Posit introduced a new field, called regime, to increase the range of numbers and shows efficacy in DNN training as well (Gustafson & Yonemoto, 2017; Lu et al., 2019). Custom floating point representations are also proposed and they can be more effective compared to INT quantization in compressing DNNs with wide weight distributions such as transformers (Tambe et al., 2019).