Architecting Effectual Computation for Machine Learning Accelerators

Hang Lu, Mingzhe Zhang, *Member, IEEE*, Yinhe Han, *Senior Member, IEEE*, Qi Wang, Huawei Li, *Senior Member, IEEE*, and Xiaowei Li, *Senior Member, IEEE*

***Abstract*—Inference efficiency is the predominant design consideration for modern machine learning accelerators. The ability of executing multiply-and-accumulate (MAC) significantly impacts the throughput and energy consumption during inference. However, MAC operation suffers from significant ineffectual computations that severely undermines the inference efficiency and must be appropriately handled by the accelerator. The ineffectual computations are manifested in two ways: first, zero values as the input operands of the multiplier, waste time and energy but contribute nothing to the model inference; second, zero bits in nonzero values occupy a large portion of multiplication time but are useless to the final result. In this article, we propose an ineffectual-free yet cost-effective computing architecture, called split-and-accumulate (SAC) with two essential bit detection mechanisms to address these intractable problems in tandem. It replaces the conventional MAC operation in the accelerator by only manipulating the essential bits in the parameters (weights) to accomplish the partial sum computation. Besides, it also eliminates multiplications without any accuracy loss, and supports a wide range of precision configurations. Based on SAC, we propose an accelerator family called Tetris and demonstrate its application in accelerating state-of-the-art deep learning models. Tetris includes two implementations designed for either high performance (i.e., cloud applications) or low power consumption (i.e., edge devices), respectively, contingent to its built-in essential bit detection mechanism. We evaluate our design with Vivado HLS platform and achieve up to 6.96× performance enhancement, and up to 55.1× energy efficiency improvement over conventional accelerator designs.**

***Index Terms*—Accelerator architectures, neural network hardware, multiplying circuits.**

Manuscript received February 15, 2019; revised June 19, 2019; accepted September 22, 2019. Date of publication October 11, 2019; date of current version September 18, 2020. This work was supported in part by the National Natural Science Foundation of China under Grant 61432017, Grant 61602442, Grant 61834006, and Grant 61876173, in part by the National Key Research and Development Project under Grant 2018AAA0102700, in part by the Beijing Municipal Science and Technology Commission under Grant Z181100008918006, and in part by the Strategic Priority Research Program of Chinese Academy of Sciences under Grant XDPB12. This article was recommended by Associate Editor C. Coelho. *(Corresponding authors: Hang Lu; Mingzhe Zhang; Xiaowei Li.)*

The authors are with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China (e-mail: luhang@ict.ac.cn; zhangmingzhe@ict.ac.cn; yinhes@ict.ac.cn; wangqi08@ict.ac.cn; lihuawei@ict.ac.cn; lxw@ict.ac.cn).

Digital Object Identifier 10.1109/TCAD.2019.2946810

I. Introduction

Deep convolutional neural networks (DCNNs) have driven significant progress in machine learning applications, such as real-time image recognition and detection, neural language processing, etc. In order to bolster the increasing accuracy demand, state-of-the-art DCNN models embrace more complex connections and ever increasing number of neurons and synapses to deal with complicated machine learning tasks. DCNN is composed of multiple consecutively connected layers, from tens [1]–[4] to even hundreds [5], [6], and in each layer, the input feature activations and weights perform convolutions for each channel in filters, which takes nearly 98% computations in the overall DCNN, accompanied by nonlinear activations, such as ReLu and pooling. Therefore, improving the computational efficiency of convolutions without compromising the robustness of the learning model is a critical step to enable efficient inference, especially on lightweight devices with limited resources and power budget like smartphones and autonomous robotics.

Given the limitations of the conventional general-purpose architectures, many researchers propose specialized accelerators targeting convolutions. Conventionally, a plethora of techniques are proposed to utilize the irrelevance of each weight-activation pair, seeking to mine the potentials of multiply-and-accumulate (MAC) operations that could be executed in parallel, for as many as possible to attain an optimal computational throughput of the accelerator [7]–[9]. However, due to the characteristics of DCNNs, inference efficiency is susceptible to substantial *ineffectual computations* [10], [11], which lies in two aspects: first, the zero operands, in both weights and activations generated in the previous layer, are accepted as input for MAC operations in the current layer. These zero values are multiplied and added together with other nonzero operands, wasting time and energy but contribute nothing to the final output feature map. To address this issue, some approaches leverage the sparsity of the input data by intentionally skipping zero values [11]–[14] or prune away near-zero values at the software level [15]–[18] as an easy way to reduce the computational intensity without hurting accuracy. Despite these effective solutions targeting zero values, the "zero-valued bits," as the second form factor, however, also occupies a large fraction of the input data set, and its consequence to the inference efficiency is not easy to be mitigated. The reason is that the structural and functional design of DCNNs relies on the multiplication and takes it as one portion to form the final partial sum. Zero values

0278-0070 © 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on November 29,2021 at 06:37:09 UTC from IEEE Xplore. Restrictions apply.

could be conveniently ignored at the input of the multiplier; zero-valued bits, however, cannot be directly skipped over in performing multiplications in the multipliers. Following this computing paradigm, modern accelerators allocate plenty of multipliers in the processing elements (PEs) to maximize the throughput [19], but according to our exploration in Section II, nonessential bits (or zero bits) in the parameters contribute as high as 68.9% ineffectual computations, which further exacerbates this problem.

Targeting this formidable challenge, in this article, we propose a novel computing architecture for accelerating convolutions, termed as split-and-accumulate (SAC), in replacement of the widely used "MAC" in modern accelerators. As a new computing paradigm, it rearchitects the computation by only embracing the essential bits (bit "1 s") in the parameters to obtain the final output activations. By decomposing the fixed-point multiplication, it accumulates the segment summation of each bit lane in advance by manipulating the essential bit, instead of direct multiplying the weight and activation pair as in MAC operation. SAC does not entail any multiplication, but replaces it with segment adding and final shifting, only once, to obtain the output activations so the inference efficiency is significantly boosted at a wide range of precision configurations, i.e., fixed point 16 (fp16), integer 8 (INT8), etc.

Besides, we implement an accelerator family, called *Tetris*, to mine the maximum potential of SAC. As an accelerator family, Tetris involves two implementations designed for different purposes, marked by its built-in essential bit detection mechanism, namely, *weight kneading* and check window (CW) *sliding*. The first one reserves the essential bits by "kneading" a batch of weights in each lane to eliminate zero bits, and the second one instantiates a CW that tries to frame the essential bits at the maximum probability at each time. Each implementation has its own pros and cons: the first one emphasizes high throughput during inference that thereby could be used in cloud datacenters, but compromises the storage and further—the energy consumption; on the contrary, the second one focuses on higher energy efficiency that is imperative for edge devices, but with the cost of moderately degraded throughput. We issued wide-scale design space exploration for the two Tetris implementations, and evaluated them via high-level synthesis tool to prove its efficacy compared with the canonical state-of-the-art accelerator designs.

The rest of this article is organized as follows. Section II quantitatively illustrates the ineffectual computation problem by presenting its existence in the parameters of various DCNN models. Section III elaborates our methodology, including the weight kneading mechanism and SAC micro-architecture. Section IV, on top of Section III, elaborates the CW mechanism, including the potential design tradeoff and how it affects the performance and energy consumption. Section V details the implementation of the Tetris accelerator, including the backbone and INT8 mode acceleration. Section VI gives the evaluations, in terms of the parameter scaling, inference throughput, power efficiency, and area overhead. Section VII concludes the whole article.

II. Background and Motivation

A. *Ineffectual Computations*

Convolution calculus is all about performing MACs in DCNN models. In order to accelerate this kind of operation, classic machine learning accelerators are architected by deploying multipliers and adders at each activation and weight lane [11], [12], [20], [21]. The multiplications could be either for floating-point 32 operands or, in most modern accelerator designs, 16-bit fixed point or even lower precisions to acquire a balanced inference efficiency and accuracy [22]–[26]. Compared to the fixed point adder, multiplier dominates the critical path of MACs. For a 2-operand multiplying, it takes 12.3% more time over the adder with even 16 adding operands as demonstrated in Fig. 1. The latency stems from the shifting of weights iteratively from the LSB of the activation till the MSB during multiplication, and worse still, different DNN models have various precision requirements at even a per-layer basis, so the multiplier designed for MAC must be able to cover the worst-case latency, even for most of the time, the shifting and summation of intermediate values are not always contributive to the final result, also known as *ineffectual computations*.

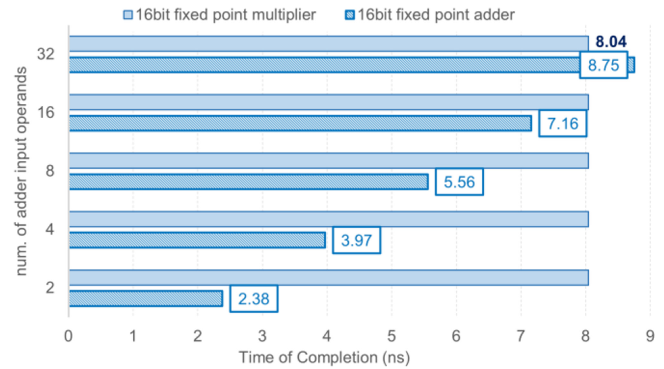

Fig. 1. Temporal overhead of 16-bit fixed point adder w/ varied input operands, versus 16-bit fixed point multiplier with only two operands. The data is obtained from RTL simulation of Xilinx Z7020 FPGA using Vivado HLS tool.

TABLE I
Fraction of Zero-Valued Weights and Zero Bits in All Weights

Models	Zero Weights (%)	Zero BITs in Weights (%)
AlexNet	0.093	70.52
GoogleNet	0.050	65.23
VGG-16	0.156	70.52
VGG-19	0.182	71.09
NiN	0.193	67.02
GeoMean	0.135	68.88

As the major problem of MAC, ineffectual computations could be manifested in two aspects: the operands are *zero values* or including a large portion of *zero bits*. Compared with zero bits, zero values occupy a trivial slice of input weights, as shown in Table I. These small portion of 0 s can be easily avoided as the input of the multipliers through advanced micro-architectural design, or compression techniques at the memory level [16]. However, in fixed-point multiplication, shifting to obtain each intermediate segment is agnostic of the zero-valued bits, another major source of the ineffectual computation. Table I shows that compared with the essential bits (or 1 s), the fraction of zero bits is as high as 68.9% on average, which means avoiding the impacts of zero bits could have potentials in boosting the inference efficiency in both performance and power, significantly.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on November 29,2021 at 06:37:09 UTC from IEEE Xplore. Restrictions apply.

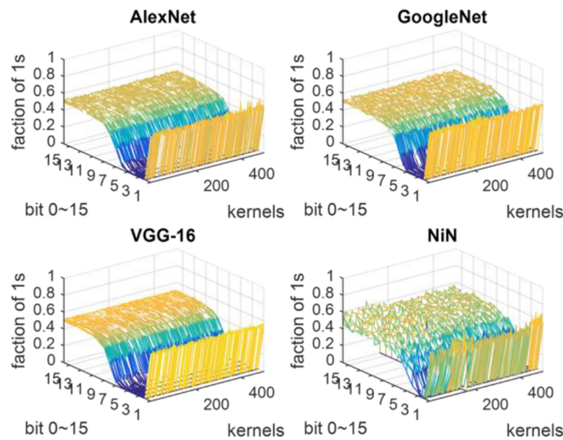

Fig. 2. Essential bit (1 s) distribution, across bit position 0∼15 for fp16 weights extracted from 500 kernels of four DCNN models.

In order to minimize the zero-bit computation, some prior approaches suggest to use bit-level serialization to perform MACs in a more fine-grained manner [10], [27], taking advantage of the feature that fixed point multiplication could be decomposed into a series of shifts and additions of single-bit multiplications and only the essential bits are involved in computing MACs. However, the essential bit may emerge at any location of a parameter value, so such bit-serialized computation scheme must rely on large shifters that must be capable to step over a series of zero bits and cover the worst-case position of the essential bit "1 s," i.e., the 16th bit in fixed point 16 ("fp16" hereafter) weights. Different values may yield unpredictable latencies in approaching the essential bits, so it requires the hardware design can also tackle this unbalanced scenario that not only increases the design complexity but also suppresses the frequency potentials of the accelerator.

B. Harnessing Slacks

Conventional CNN accelerators seek to process a series of weights and activations in the lane in batches by allocating a certain amount of PEs capable of absorbing as large as 256 weight/activation pairs in total [19]. Each pair is feed into its PE and finishes MAC within one cycle, with ineffectual computation of zero bits also accounted. If we could make use of the time allocated for ineffectual computation but use it for essential computation, it would be definitely beneficial for the throughput. In this article, we term the zero bits in weights as *slacks* hereafter for simplicity.

In Fig. 2, we evaluate the proportion of essential bits across the entire bit positions in the weights of four commonly used DCNN models. We selected 500 convolutional kernels across different layers and found that the distribution of 1 s demonstrates two similar behaviors: 1) the portion of essential bits remains nearly identical at each bit location, around 50%∼60%, which also means 40%∼50% portion are slacks at these locations. No bit position exhibits radical spikes of essential bits and 2) certain bit positions exhibit a large portion of slacks. For example, position 3∼5 only have less than 1% essential bits. The "cliff" at these positions indicates that it is almost composed of slacks in this bit lane, but the multiplier does not differentiate them with essential bits when performing MAC. If we want an augmented inference efficiency, the slacks must be harnessed.

The stable distribution of essential bits in Fig. 2 provides a unique opportunity to squeeze out the slacks at a per-bit level. In specific, if the slacks presented in previous weight could be replaced by essential bits (1 s) of the subsequent weight, we could replace the ineffectual computations with essential contributive computations and process multiple pairs in one cycle. Fig. 2 has proved the headroom of acceleration could be as high as 50%, and it does not emerge any roofline at any bit position so the overall weights could be compressed into nearly half of their initial volume. In other words, we could save 50% time during inference. However, it would be tricky to achieve this goal because we need to modify the existing computational architecture originally designed for MAC, and rearchitect it to support new computing patterns. In the next section, we will elaborate how our scheme is designed for this purpose.

III. Enforcing Effectual Computation

A. Prerequisite

As described in many previous [10], [27], fixed point multiplication could be decomposed into a series of *shift-and-adds*, governed by the following:

$$A \times W = \sum_{b=0}^{B-1} 2^b \times (A \times W^b). \tag{1}$$

If we have B length fixed point weight (W), the activation (A) is shifted b bits at each addition. The summation of these intermediate shifting segments denotes the final result. Similarly, we could extend this equation to multiple A/W pairs

$$\sum_{i=0}^{N-1} A_i \times W_i = \sum_{b=0}^{B-1} 2^b \times \sum_{i=0}^{N-1} (A_i \times W_i^b). \tag{2}$$

In the above (2), we first add all N number of As according to the bth bit of Ws, which could be either 0 or 1, and then perform shift-and-accumulate for the final summation. As can be seen, the value of W_i^b determines if the summation of A_i is an ineffectual computation or vice versa. We aim to replace these ineffectual bits with the essential bits of subsequent weights as described in Section II-B, but first, we must be able to detect the essential bits in the weights, which will be specified in the next section.

B. Weight Kneading

Following the (2), if the bth bit of weight W_i is a slack, we seek to explore subsequent weight in the lane, i.e., W_j, and if its bth bit is an essential bit, the slack is replaced with W_j^b, and make the corresponding activation A_j contribute to the current summation within the same cycle. In other words, effectual computation is also enforced at the maximum possibility.

Fig. 3 shows the general concept of this method. If we group six weights as a batch, conventionally it will take six cycles to accomplish six weight/activation MACs, because they are fed to the PE one after another from the on-chip eDRAM. Many previously proposed designs follow this paradigm [19], [21]. If we further interpret the weights and blind the zero bits, the slacks emerge at two orthogonal dimensions: 1) in the intraweight dimension, i.e., w_1 and w_6, slacks demonstrates arbitrary distribution and 2) on the other hand, the slacks also

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on November 29,2021 at 06:37:09 UTC from IEEE Xplore. Restrictions apply.

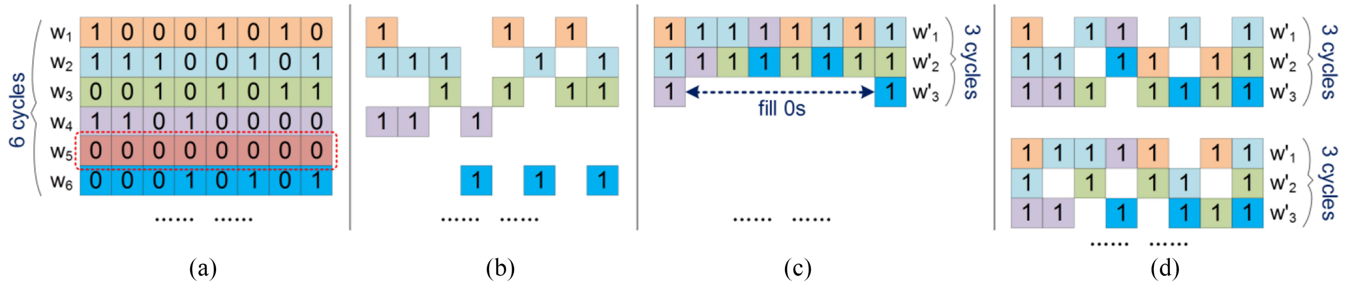

Fig. 3. Weight Kneading. The design philosophy lies on squeezing the slacks in consecutive weights. (a) Scenario of input weights w/ zero slacks before kneading; eliminating slacks and only essential bits are left in (b); weights after kneading is shown in (c); various equivalents of (c) are shown in (d); and (c) and (d) have equal computing cycles after kneading.

Algorithm 1 Weight Kneading Mechanism. The Goal Is Squeezing Out the Slacks in Each Bit Column, Only Leaving the Essential Bits

Require: ks number of original weights: W=[w_0, w_1,..., w_{ks-1}], with precision B (B could be either fp16 or INT8, etc);
Ensure: n number of kneaded weights: W' = [w'_0, w'_1, ..., w'_{n-1}], $n = ks$; and indices: P = [p_0, p_1, ..., p_{n-1}];

```
1:  n = 0; b = 0; i = 0; //initializations
2:  ¹foreach bit b in B { //iterate each bit column in Figure 3(a)
3:    foreach weight w_i in W {//iterate each weight in the bth
4:                              //column
5:      if (w_i[b] == 1) {//identify essential bit
6:        w'_n[b] = 1; //store essential bit in the kneaded weight
7:        p_n[b] = i; //store the index indicating the
8:                    //corresponding activation
9:        n++; //accumulate the index of the kneaded weight
10:     }//end if judgement
11:   }//end each weight in W
12:   n = 0; //reset the index for the next bit column
13: }//end all bit columns
14: Output: P and W'; //we only need the kneaded weights
15:                   //and the indices
```

[1]The outer "**foreach**" loop is actually executed in parallel in hardware.

show up at interweight dimension, i.e., w_5, who is an all-zero-bit (zero-value) weight, so it does not emerge at Fig. 3(b). If we elevate the essential bits taking the space previously occupied by the slacks, the computation cycles of six MACs will be decreased to only three cycles, as shown in Fig. 3(c). This time we obtain w'_1, w'_2, and w'_3, but each one is combined with the essential bits of previous w_1~w_6. w'_3 has slacks because we only take 6 weights as the example. If we allow more weights involved, these filled zero bits are likely to be replaced by more essential bits, and we term this whole operation as *Weight Kneading* in this article.

Obviously, the benefits of weight kneading are, on one side, it automatically eliminates the impact of zero values before feeding them into PE, without introducing extra operations or specialized hardware. As a small slice of total weights, the accelerator does not need to specifically deal with zero values, so the design complexity has the opportunity to decrease. On the other side, zero bits are also replaced by essential bits after kneading, so it avoids the impact of slacks at two dimensions within one mechanism. However, kneaded weights indicate the current summation is a combination of multiple corresponding activations instead of one activation alone, so the A_i in (2) must be able to reference a set of activations according to the bth bit of w'_i. For example, if we configure six kneaded weights, then six corresponding activations must be reachable even if each kneaded weight may not need all of them, as Fig. 3(c) shows: w'_1 only needs A_1, A_2, and A_4. The number of weights for kneading is a design parameter, termed as kneading stride (KS) and we will evaluate its scaling to the inference efficiency in our evaluation section. Algorithm 1 details the weight kneading procedure. It is worth mentioning that in line 2 the outer *foreach* loop depicts the procedure for each bit column, while in practical hardware implementation such procedure could be issued in parallel in each weight lane. After kneading all columns, we get P indicating the indices matrix and W' indicating the kneaded weights matrix.

C. Split-and-Accumulate

Equation (2) indicates that we could only account for the essential-bit related activations in computing the partial sum, and the result is the same. Therefore, having introduced our mechanism for detecting the essential bits in the previous section, we propose the key micro-architecture designed to support such highly efficient computation.

Intuitively, the computing architecture might be significantly distinct with conventional ones performing classic MACs, because in here it must be capable to manipulate the individual bit, rather than the entire weight datum. Re-examining (2), we can see that there are two "Σ"s in the equation on the right-hand side, with the first one responsible for "shift-and-add" based on index b and the second one for adding contributive "A_i" on the bth position for all the weights. Since accumulating "A_i" is independent, we can parallelize the second Σ using "B" number of adders, and perform the first Σ right after we obtain B number of summations ($\sum_{i=0}^{N-1}(A_i \times W_i^b)$). This induces our proposed computing architecture that is different from the conventional MAC and canonical bit serialization schemes [10], [27], and we term it as "SAC" in this article.

SAC denotes "Split-and-ACcumulate," and Fig. 4 shows the architecture of SAC. An SAC operation first splits the kneaded weight, references the essential activations and finally accumulates each activation to the certain segment registers "S." SAC targets each bit in the weight, not the whole value so it does not involve multipliers in its architecture; instead, it instantiates a "splitter" according to the precision of the weight: if we use fp16 precision for each weight, we need 16 segment registers ($p = 16$ in the figure) together with 16 adders with two input ports each. Typically, the splitter is responsible for

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on November 29,2021 at 06:37:09 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2 CW Mechanism. The Goal Is Identifying the Essential Bit Framed by the CW

Require: *ks* number of original weights: W = $[w_0, w_1, \ldots, w_{ks-1}]$, with precision B (B could be either fp16 or INT8, etc); check window size: *ck*; Activations: A = $[A_0, A_1, \ldots, A_{ks-1}]$;
Ensure: the chosen activation in the window: Y, and the starting point of the next cycle: *start*;

```
1:  start = 0; //initialization
2:  ¹foreach bit b in B {//iterate each bit column in Figure 5
3:    ²while(start = ks-ck){X = 0; //reset the flag
4:      for (i = start; i < start + ck; i++) { //within ck
5:        if (w_i[b] == 1) {X = 1; Y = A_i; break; }
6:      }
7:      if (X == 1) {X = 0; //reset the flag
8:        for (i = i+1; i < start + ck; i++) {
9:          if (w_i[b] == 1) {X = 1; start = i; break; } //Case A
10:       }
11:       if (X == 0) { start = start + ck;} //Case B
12:     } else {Y = 0; //Case C
13:                 start = start + ck; //update the starting point of
14:                                     //the check window
15:     } //end if - else judgement
16:     Output: Y and start;
17:   } //end while loop
18:   start = 0; //reset the starting point for the next column
19: } //end all bit columns
```

[1]The outer "**foreach**" loop is actually executed in parallel in hardware.
[2]The inner "**while**" loop dominates the critical path.

dispersing each activation to its corresponding adder as governed by (2); for example, if the second bit of a weight is detected as an essential bit, activation is delivered to S_1 in the figure. The same operation applies to other bits.

After "splitting" this weight, subsequent weights are handled in the same way, so each segment register *S* is accumulated with new activation if its associate weight has an essential bit. After dealing with the lane outright, subsequent adder tree performs shift-and-add, once and for all, to obtain the final partial sum. Different from MAC, SAC does not perform any shifting to obtain the intermediate pair-wise A/W multiplication. The reason is that in the real CNN model, the output feature map only accounts for the "final" partial sum, that is, the convolution of all filter channels and its corresponding input feature map. As for the intermediate partial sum, the value is not that useful so it is totally superfluous to waste time and energy procuring these useless values as conventional MAC does. SAC relies on this fact and move all shifting operations to the rear adder tree, so it also shortens the critical path thereof and the main frequency of the accelerator has the potential to increase.

The splitter for weight kneading is also shown in Fig. 4. Weight kneading requires the splitter could accept a set of activations for one kneaded weight, and each essential bit must be recognizable to indicate the relevant activation within the KS. Therefore, we use <w'$_i$, p> tuple in the splitter to represent the essential bit and its index (p). The bit-length of *p* is a proxy of KS, i.e., 4 bits *p* refers to a KS of 16 weights. It allocates one comparator to determine if the input bit of a kneaded weight is zero, because it is possible that some bit positions are still ineffectual even after kneading, i.e., w'3 in

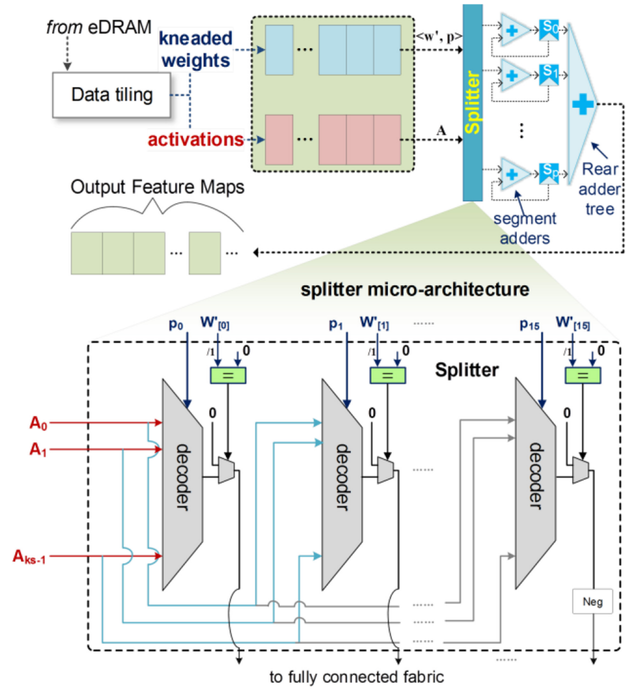

Fig. 4. SAC. Differed from MAC, it does not aim at the exact pair-wise multiplications; instead, it first splits the weight in the splitter, and the activation is summed in the corresponding segment adders. Final shift-and-add for a partial sum is performed at the rear adder tree, after addable pairs accomplish splitting and accumulating.

Fig. 3(c). If it is a slack, the multiplexer after the comparator ("=") will output zero to the fully connected fabric. Otherwise, it will decode *p* and output corresponding activation among $A_0 \sim A_{ks-1}$.

IV. Optimizations for Energy and Storage

In weight kneading mechanism, the generated kneaded weight w' is a combination of essential bits that previously belong to the original weights. Still taking w'$_1$ as the example, when $b = 0$, A_i in (2) then indicates A_1, but w'$_1$ cannot recognize it because all the bits are the same bit "1." That is why we must allocate "indices" to each bit for this purpose. As the additional enrolled factor, if we use KS = 6 as the example in Fig. 3, 3 bits are enough to represent the indices: A_1 could be represented as "000" and A_2 "001," etc. The indices are used to decode the activations when performing (2). Obviously, when KS is set larger, the newly introduced indices will consume larger storage as well, i.e., 4 bits for KS = 16, 5 for KS = 32, which also means the storage resources must be expanded to 4 times in order to store the kneaded weights. Assuming that weight kneading could reduce 50% of the original weights (2× storage reduction), the indices, however, expand 4× the storage so the overall storage still increases 2×. Increased storage inevitably leads to an increased power consumption, so in this section, we specify how to optimize this parameter and propose the *CW mechanism* for the essential bit detection.

A. Check Window Sliding

Weight kneading could eliminate the slacks, and the final cycles consumed for computing the partial sum within KS is decided by the bit column with the maximum essential bits.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on November 29,2021 at 06:37:09 UTC from IEEE Xplore. Restrictions apply.

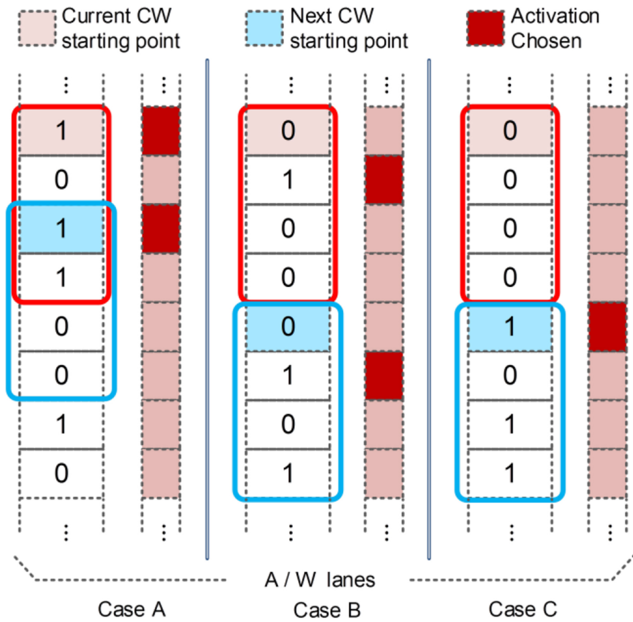

Fig. 5. Concept of CW Sliding. Red line frames the bits in the current computing cycle, while the blue line frames the bits of the next cycle. The chosen output activations are marked by the red blocks.

Still taking Fig. 3 to exemplify this statement, 3 cycles in Fig. 3(c) are concluded by the rightmost or the leftmost bit columns. The bit columns in between all demonstrate two cycles with an extra slack. When computing this group of kneaded weights, actually we do not need to care if the former bit 1 s in the middle lanes are computed in the first, second, or the third cycle, because anyway the total computing cycles are 3, dominated by the side columns. In other words, the slack bit could bubble up to the top w'1 or the middle w'2 while kicking down the essential bit to w'3. Even if the first cycle might compute a slack, but the overall computing cycles remain the same, as shown in Fig. 3(d).

With this mindset, we seek to alleviate the tight constraint a little bit in the previous weight kneading, that is, squeezing out all the slacks and leaving the irremovable slacks at the rear of the column. Instead, we try our best to output the essential bit to SAC for each column within the current computing cycle, but still allow in some of the cycles that it outputs a slack, and the overall computing cycles are governed to be nearly identical to the strict weight kneading mechanism. In specific, we introduce a new paradigm in Fig. 5: there are three scenarios indicating three different bit columns and before kneading, we can see that slacks are arbitrarily located and interleaved with essential bits (bit 1 s). Instead of direct kneading, we assign a CW that is able to slide in the lane, from the top to the bottom within a certain range. The bits framed within the window could be referenced within one clock cycle. Intuitively, we want these bits most possibly contain an essential bit just like in weight kneading. However, it is also highly possible that the window contains only slacks. For example in Fig. 5(a), the window with a length of 4 (marked in red line) frames 4 bits, with the first bit an essential bit, so it outputs this bit in current computing cycle. However, in case B, the first bit is a slack, but there is an essential bit in the second place within the CW, so this bit can also be selected and output by the window. For the extreme case C, the 4 bits are all slacks, so in this computing cycle, it can only output a slack, but in our CW mechanism, we allow this scenario to happen but control it as minimum as possible.

B. Design Tradeoff

If we want to maximally output essential bits in the same cycle, it imposes two requirements to the mechanism. First, the window must be capable to reference all the bits within its frame, which is just the greatest benefit of the scheme because the CW inherently recognizes the location of the essential bit. It references the bit one after another so it is totally unnecessary to record the indices. Still taking case B as an example, when the window recognizes that the second bit is 1, it directly outputs the second activation in accordance with the position of this essential bit. In previous weight kneading, it only needs to reference one bit in the kneaded weight within one cycle, but this time it needs to traverse multiple bits, up to the length of the CW. The prolonged clock cycle influences the main frequency and cripples the model inference speed. This is a tradeoff between storage and computation speed during inference. For some cases (i.e., computing huge DNN model like object detection and semantic segmentation), speedup is the predominant design constraint so we could allow the moderate increase in storage in exchange of fast inference, but in other scenarios, power consumption is the predominant factor and we cannot tolerate sharp increase in memory especially in battery-powered edge devices like security cameras and robotics, because the increased storage inevitably leads to an increased power consumption. This flexibility also provides a unique feature that is apart from existing DCNN accelerators. The user is able to choose if the accelerator works at high-performance mode, i.e., in the cloud, or power-efficient mode, i.e., in edge devices. The former could configure the accelerator using weight kneading, while the latter could use fixed-length CWs.

The mechanism should not only precisely identify output activations but also determine the sliding of the CW. Therefore, the second requirement is supposed to be fixing the starting point of the window for the next computing cycle. In Fig. 5, the windows marked in blue denote the framed bits of the next cycle. We stipulate the starting point is the second essential bit in the previous red window. Otherwise, if there is no second essential bit, i.e., case B, or no essential bit at all in the entire window, i.e., case C, the starting point is stipulated to be the first bit beyond the window length, as demonstrated in Fig. 5. The activation selecting procedure is totally identical for the current cycle. The sliding control logic works iteratively until the end of the bit column. We could also use the parameter KS to indicate the maximum range that the CW could slide, but this time it does not mean the stride of kneading but the stride of sliding. Algorithm 2 details the procedure of this detection mechanism. Three cases elaborated in Fig. 5 are involved in the algorithm. Apart from weight kneading (Algorithm 1), the algorithm outputs no weight matrix, but directly outputs the target activation denoted by the essential bit in the window, and the starting point for the next cycle. The inner *while* loop determines the critical path in the practical hardware implementation, which also reflects the design tradeoff as mentioned above. Upon its significance, a feasibility analysis is necessary.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on November 29,2021 at 06:37:09 UTC from IEEE Xplore. Restrictions apply.

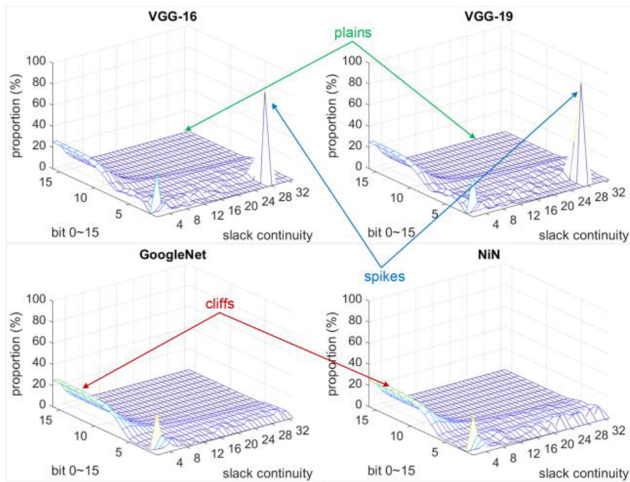

Fig. 6. Feasibility analysis. We characterize the behavior of slack continuity, which means the number of zero bits that are continuous in this lane. We analyze the continuity ranging from 1∼32. The mesh plot of 4 DCNN models exhibits similar behavior. Higher proportions concentrate at slack continuity from 1 to 3 across almost all the bit lanes, followed by a relatively plain area from 5∼32.

C. Feasibility Analysis

Even though we do not need to record the indices in the CW mechanism, but the risk is the output bit is not guaranteed to be essential. The window length (*ck* in Algorithm 2) has a significant impact on this concern and should be appropriately configured off-line making the slack output as minimum as possible. Intuitively, it is preferable that the CW frames at least one essential bit at each cycle, but the distribution of the slacks is highly unpredictable and might be different across the bit columns. It is possible that setting the size too short makes the output almost slacks and further undermines the effectual computation goal; on the other side, setting it too long will elongate the traversal time, and as the critical path, it may potentially drag down the main frequency the accelerator could attain.

To explore the proper configurations, we seek to figure out the distribution of contiguous slacks, that is, the number of slacks that are contiguous in the column dimension and its emergence. For example, if we find that eight contiguous slacks are the mostly observed case for each lane, the window size could be set to 9 or 10. That is because if the window encounters an 8-slack case, the window is large enough to step across all the slacks in one cycle and absorb the essential bit at the highest probability. Even if we cannot completely avoid the slack output, we could decrease this scenario to the minimum. Concretely in Fig. 6, we explore each bit column using fp16 precision (x-axis), and stat the continuity of slacks ranging from 2∼32 (y-axis), for 4 state-of-the-art DNN models. The z-axis indicates the proportion of continuous slacks over all the cases. Starting from VGG-16, we find there is a cliff across bit 8∼15 and slack 0∼3. The top proportion of the cliff stables at nearly 25%, while the proportion of the plain area however is around 1% for y-axis scaling from 8∼24. It means a continuity of 0∼3 dominates the distribution, which also provides a vital evidence in setting the CW size. Supposing we set the window size equals to 4 or 5, for most of the cases, the window will frame an essential bit. Another observation in Fig. 6(a) is a sharp spike at 27 on the y-axis. The proportion is around 85% at bit lane 0∼2, but other columns do not exhibit this phenomenon. Similar scenario also shows in Fig. 6(b) for VGG-19. Setting the window size around this value is supposed to be more appropriate for these bit columns, but increased window size also harms the critical path delay as mentioned, and a majority of columns do not benefit from large window size because they must wait for the worst-case scenario even if an essential bit has been identified already, wasting the rest of the time in the computing cycle. For the NiN and GoogleNet model, the cliff is more obvious with no spikes at larger continuity. We will show in Section VI-A that the configuration of window size and its impact on the accelerator performance is highly aligned with the observations in Fig. 6. We can also conclude from the uniform behavior of various DNN models: CW mechanism is feasible in detecting essential bits, by setting the same window size across bit columns.

D. Micro-Architecture Support

In terms of the hardware design, it is slightly different from the weight kneading. The indices are no longer stored in the throttle buffer because the CW could reflect the chosen activation. As shown in Fig. 7, the input activations are still constrained by the KS parameter for the least modification of the hardware. "W_{ck}" in the figure denotes the input weights in each lane, and its length is consistent with KS as well. Omega module ("Ω") is specially designed for the CW sliding logic and starting point selection logic, which could be implemented using cost-effective combinatorial circuits. The output of Ω is twofold: 1)the selector of the decoder for this bit column (also the "Y" in Algorithm 2), used for selecting the target activation based on the essential bit framed in the CW and 2) if the CW outputs a slack, the enabler (also the "X" in Algorithm 2) bans the activation and outputs 0 to the fully connected fabric.

The implementation details of Ω only involve several AND and NOT gates, which aims to select the first and second essential bit from I_0∼I_{ck}, denoted by the output Y. The first bit is used to select the target activation, and the second bit is to identify the starting point after sliding for the next computing cycle. If all the input Is are slacks, the enabler is 0, denoted by X in the figure. Although Ω is very cost-effective and composed of only combinatorial circuits, it is crucial to explore its impact on the main frequency because selection and control logic will, to some extent, increase the critical path latency and as the key design tradeoff, we will thoroughly evaluate it in Section VI-A.

V. Tetris Accelerator

A. Backbone

For the practical accelerator design, it should leverage the essential bit detection mechanisms, weight kneading or CW sliding, combined with their respective hardware architecture to achieve highly efficient inference. In this section, we elaborate our Tetris accelerator design, which includes two implementations with respect to the built-in detection mechanism differentiated by the splitter. Fig. 8 shows the backbone of Tetris. Each SAC unit accepts addable A/W lane. Specifically, the accelerator consists of symmetric SAC

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on November 29,2021 at 06:37:09 UTC from IEEE Xplore. Restrictions apply.

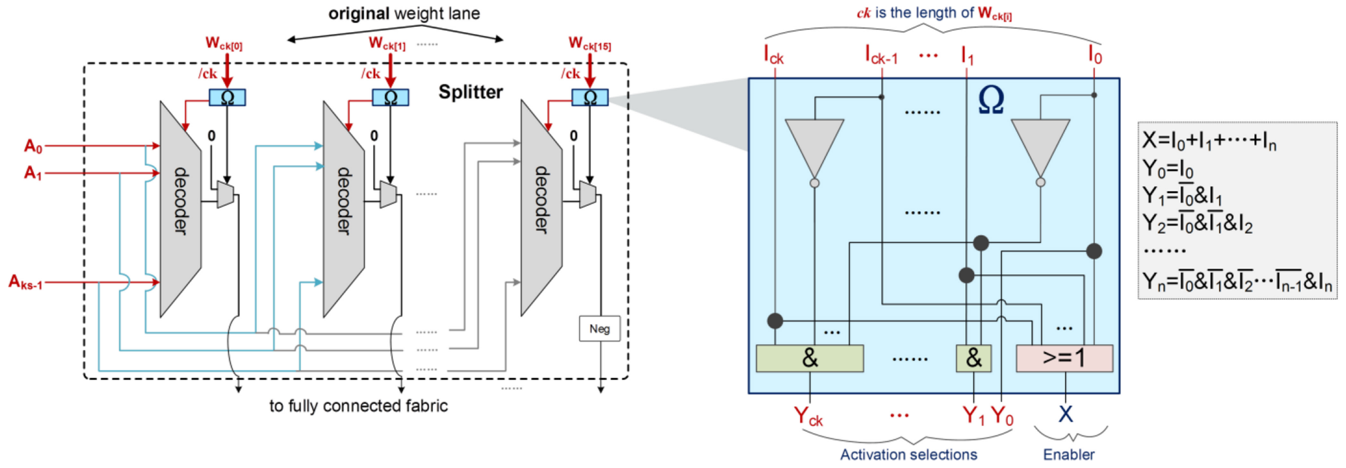

Fig. 7. Splitter micro-architecture for the CW mechanism. "Ω" is the newly introduced module that has two purposes: the first one is selecting a target activation (Y_n) or slack (enabler X) according to the input "I" with "ck" length. "ck" is also the length of the CW; X Y here is consistent with Algorithm 2; the second one is to identify the starting point of the window for the next cycle. "Ω" is purely combinatorial logics.

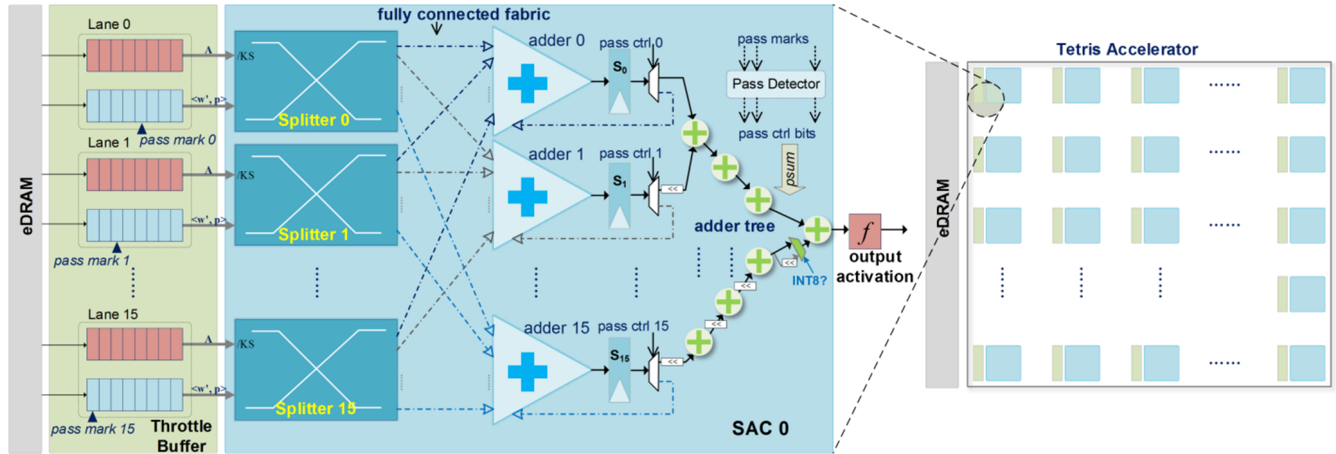

Fig. 8. Tetris accelerator architecture. Each backbone SAC unit is composed of 16 parallel organized splitters accepting a series of (kneaded) weight and activation batches denoted by parameter KS, and the same number of segment adders as well as a rear adder tree are responsible for calculating the final partial sum.

units, connected with the throttle buffer accepting a series of (kneaded) weights from the on-chip eDRAM. Each unit contains 16 splitters comprising a splitter array, if we use fixed point 16 weights. Each splitter is able to reference multiple impending activations according to the parameter KS. There are 16 output data paths in each splitter for the activation to reach its target segment register S, so it forms a fully connected fabric between the splitter array and segment adders. For each segment adder, it receives activations from all 16 splitters for adding, as well as the values from the local segment register. The intermediate segment sum is stored in $S_0 \sim S_{15}$ register, and once all addable channel lanes are accomplished, "pass control signals" inform the multiplexer to pass each segment values to the rear adder tree. The last level of the adder tree generates the final partial sum and passes it on to the nonlinear activation function and pooling. In the throttle buffer shown in the figure, the pass mark denotes the end of the addable A/W pairs, which will be notified to the pass detector in each SAC unit. If all the pass marks reach the end, the rear adder tree is valid for the shifting summation to obtain the final partial sum.

Since we use KS as a parameter to control the number of weights in the lane for the two mechanisms (though for different purposes), different lanes will have different number of left weights because we cannot guarantee the essential bits are spawned in synchronization for each lane, so the pass marks, for most of the time, are not synchronized as well and may reside at any location of the throttle buffer. We can allow new addable A/W pairs filled into the throttle buffer once it is empty and do not need to wait for other lanes to finish, so this load imbalance does not impact the calculation of each segment partial sum. Quite on the contrary, it pipelines the memory accesses and benefits the throughput enhancement compared with the strictly synchronized MAC.

B. 8-Bit Quantization Acceleration

Tetris is capable of precision tunable acceleration—another benefit brought by SAC. Some prior work has proved the accuracy and precision of the DCNN model demonstrate a tradeoff, and a graceful accuracy degradation is acceptable when the precision is tuned and decreased a little bit for each layer [27]. Tetris does not need to modify its intrinsic architecture to fulfill this purpose, because the contributive activations are bit dependent instead of value-dependent. If we shrink the length

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on November 29,2021 at 06:37:09 UTC from IEEE Xplore. Restrictions apply.

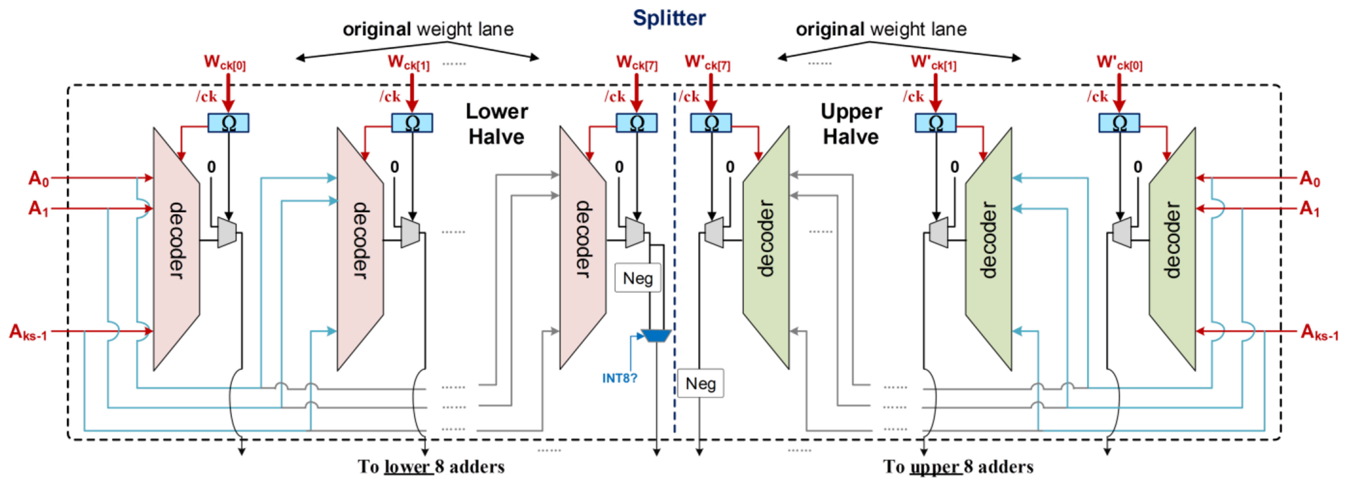

Fig. 9. Modification of the splitter for accelerating INT8 inference. Differed from fp16 mode, the splitter is halved into two parts with each one handling an 8-bit width weight. The figure employs the splitter designed for the CW mechanism. For the weight kneading-based Tetris, the design concept is similar, following Fig. 4.

of weights from fixed point 16 to arbitrary lengths, i.e., 8, 9 or even 4 bits, SAC operates itself in the same way as fp16, with only one distinction that not all the adders after fully connected fabric are active. If we use 4-bit weight, only adder0 ∼adder3 in Fig. 8 remain activated because they accumulate segment 0 to 3 according to the weight precision.

In recent years, 8-bit integer quantization (INT8 hereafter) is proved to be amenable to high throughput inference and even training. Many deep learning frameworks like Tensorflow and Caffe tends to incorporate INT8 arithmetic as the basic operation in stochastic gradient descent calculation. It is also leveraged by many types of GPUs and software programmable engines like TensorRT [28] to accelerate the inference of DCNN models. As the unique feature of Tetris, it could provide a theoretically doubled inference efficiency compared with fp16 mode which makes Tetris promising in deploying DCNN models in lightweight devices. By simply configuring the splitter array and segment adders, Tetris could be configured to INT8 mode as Fig. 9 shows. The fundamental architecture of the SAC unit remains nearly unchanged except for the splitter. The splitter is first divided into two halves, with each half accepting one weight. It turns out that the upper eight adders will deal with the upper half activations, similar for the lower eight adders. Besides, only the last level of the rear adder tree needs to be configured to distinguish between the two modes. By this manipulation, the segment adders and rear adder tree are both sufficiently utilized without idle components. Since each splitter accepts two weights as input, the throughput of the SAC unit is also doubled under the same KS setting, which means with the same amount of activations, the inference efficiency would also become twice in theory compared with fp16 mode. Note that INT8 acceleration could be supported in both weight kneading based (Fig. 4) and CW-based version of Tetris (Fig. 9), with the only difference in the splitter design.

VI. Evaluation

In this section, we evaluate the proposed detection mechanisms, the SAC architecture and the Tetris accelerator family. The evaluations are issued based on the following aspects.

1) *DNN Models:* The DNNs and their pretrained parameters are obtained from Caffe Model Zoo. Tetris embraces the model trained with any batch size. We quantize the initial floating point 32 weights into fixed point 16 and integer 8 precision. Then, we fine-tune the weights using framework Caffe [29] to maintain the Top-1 classification accuracy. The obtained fp16 and INT8 weights are used in evaluating the inference efficiency under various DNN models.
2) *Baselines:* In recent years, many literatures leverage the sparsity in the parameters and intend to skip the zero value operands before computing the partial sum. Almost all of them strive to eliminate zeros right before the operands are handed to the multiplier. For instance, cnvlutin [11] optimizes both zero values and near-zero activations based on a DaDianNao [19] alike accelerator. Also based on the DaDianNao, Stripes [27] further transforms the parallel partial sum computation into the bit-serial manner. Its design goal is accommodating different per-layer precision requirements of DNNs. PRA [10] further improves the stripes design by only counting on the pragmatic essential bit, but still ties to the bit-serial design concept. There are also accelerators that are not based on the DaDianNao backbone. For instance, EIE [14] is specially designed for accelerating the compressed DNN models, by avoiding the arithmetic of zero activations and pruned weights. SCNN [12] further eliminates the zero activations in addition to the zero weights. The fundamental design includes a scatter on-chip network and a tiled architecture. Although these works are effective in reducing the ineffectual zero-value computations, the acceleration, however, is disabled facing zero bits. Therefore, we employ two baselines in our evaluations: a) DaDianNao [19] and b) PRA [10], because first, the backbone of Tetris is based on DaDianNao, also allocating weight/activation lanes symmetrically, and second, Tetris also solves the bit-level ineffectual computation which is also the same problem PRA intends to tackle. We implement Tetris with two configurable precision modes, fp16 and INT8, as mentioned in Section V. The weight kneading-based Tetris

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on November 29,2021 at 06:37:09 UTC from IEEE Xplore. Restrictions apply.

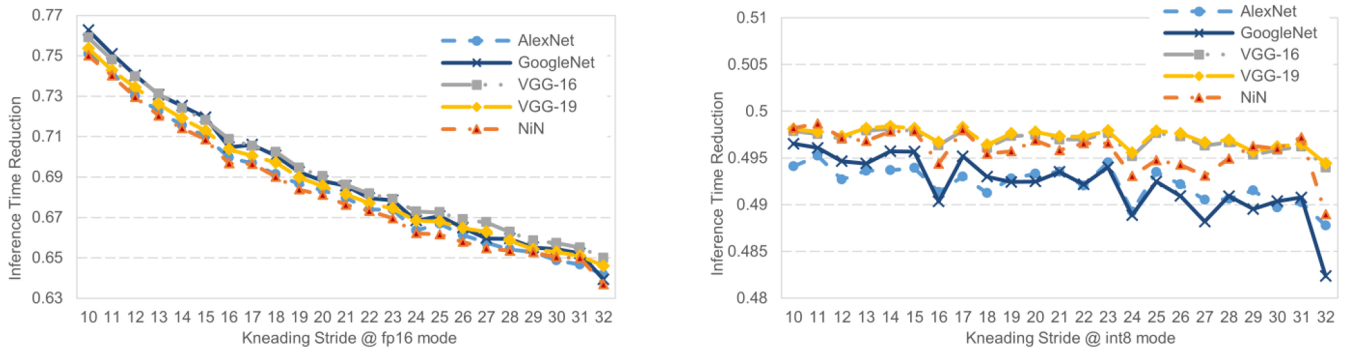

Fig. 10. Performance under different KS configurations for fp16 (left) and INT8 (right) mode. We use Tks/Tbase in plotting this figure, where Tks is the time consumed after kneading KS number of weights, while Tbase is the time that weight kneading is not applied.

is termed as Tetris-KN, as the representative of high-performance DCNN acceleration, while the CW-based Tetris is termed as Tetris-CW, as the representative of power-efficient DCNN acceleration.

3) *Platforms:* We employ Vivado HLS (v2016.2) to conduct C simulation and C/RTL hybrid simulation to extract the inference time of various DCNN models. For area evaluation, we compile our design using Synopsys Design Compiler [30] with TSMC 65 nm technology library. For power/energy measurement, we use PrimeTime tool [31] after HLS for the intrinsic components of Tetris and the baselines.

4) *Design Specifics:* For the design parameter KS, we choose 16 weights to be kneaded in Tetris-KN, in other words, the input splitter could reference 16 activations at one time according to the input kneaded weight. This design parameter is also evaluated to explore the sensitivity of Tetris-KN to the inference speedup. We use 16 PEs, and for Tetris-KN and the baselines, each PE is clocked at the same frequency referenced in HLS simulation for Xilinx Z7020 FPGA that fp16 multiplication could be accomplished within one cycle. The practical frequency for our proposed Tetris-KN is possible to be tuned higher because the replacement of multipliers with multioperand adders (SAC) provides abundant timing intervals which could be utilized to boost the inference frequency, but in this article, we keep the frequency constant for fair comparison in evaluating the actual inference time. However for Tetris-CW, it is a little bit tricky because different window sizes will influence the main frequency, due to the critical path delay introduced by the Ω module. Therefore, we set the frequency to cover the worst-case scenario, that is, the time spent on traversing the *entire* CW for the essential bit and the next starting point, and explore its performance under different window size and frequency configurations.

A. Parameter Scaling Analysis

KS: In Tetris-KN, as the major design parameter, it affects the runtime inference speedup because the performance primarily depends on the amount of weight/activation pairs computed. Intuitively, the more weights kneaded, the more speedup obtained. That is because more kneaded weights will lead to more opportunities to fill up zero slacks that further increases the effectual computations and hence the throughput. Besides, more kneaded weights reduce the operations of computing the partial sum, so it shortens the inference time that further contributes to more energy savings. Our evaluation proves this notion as shown in Fig. 10. We scale the KS from small (10 weights) to large (32 weights) and evaluate the inference time reduction at each step. The total time is saved to 75.1% for AlexNet at KS = 10, and further reduced to 64.2% at KS = 32. Other DNN models exhibit similar behaviors at fp16 mode. For INT8 mode, the data are ranged randomly between ∼48% and ∼50%, and it shows an obvious saturation across the employed DNNs. The reason is that the slacks are fewer compared with fp16 mode because INT8 has fewer bits in nature and enlarging KS will not significantly improve the inference time. Even so, INT8 mode is also much faster because Tetris accelerates this mode by halving its splitter in Fig. 9.

On the other hand, we cannot naively set KS as large as possible because more interleaved essential bits also lead to wider decoders in the splitter, and will inevitably increase the storage intensity due to the larger indices, i.e., KS = 32 requires 5 bits for each <p> in Fig. 4. In order to acquire a balanced design complexity and inference speedup, we choose KS = 16 in conducting subsequent evaluations, but note that this design parameter could be dynamically configured considering different speedup, storage, and power consumption constraints.

CW Size: In Tetris-CW, the length of CW predominates efficiency. Proper setting should make the window frame as many essential bits as possible and make the slack output as minimum as possible in each cycle. At the same time, we cannot blindly pursue this objective by enlarging the window size at all costs, because the combinatorial circuits in Ω also scale larger along with the window size, introducing additional critical path latency overhead. Therefore, we evaluate this parameter from two aspects.

1) The first one only considers the *increment* of inference cycles, without taking the path delay of Ω into account. Since Tetris-KN ensures the output at each time for SAC is an essential bit, it could be regarded as the theoretical upper bound, so the increment of inference cycles under different window size settings of Tetris-CW over Tetris-KN hence reflects the magnitude of the slack output. Obviously, the less the increment, the better the setting.

2) The second one involves the critical path delay on top of the inference cycles to explore the actual inference

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on November 29,2021 at 06:37:09 UTC from IEEE Xplore. Restrictions apply.

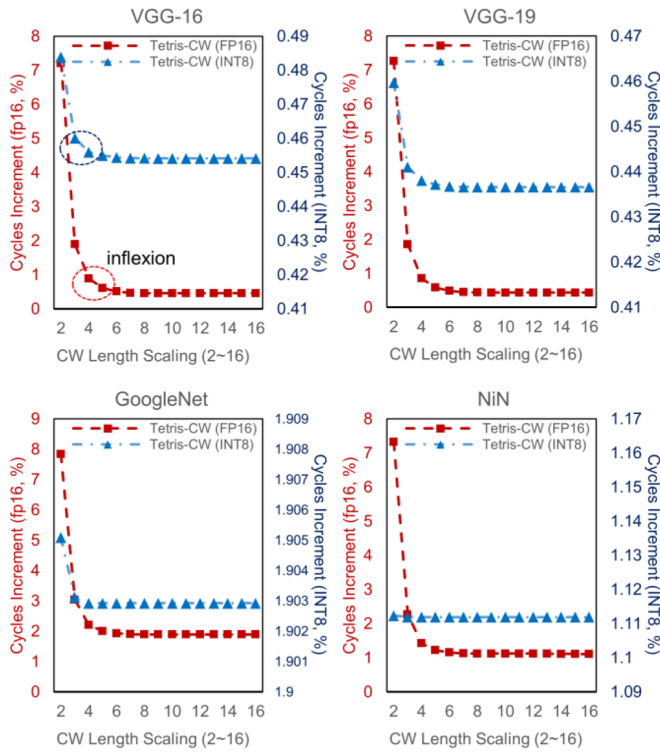

Fig. 11. Window size scaling and its impact on inference cycles. We stat the cycle increment proportion over Tetris-KN with a KS of 16 under fp16 and INT8 precision.

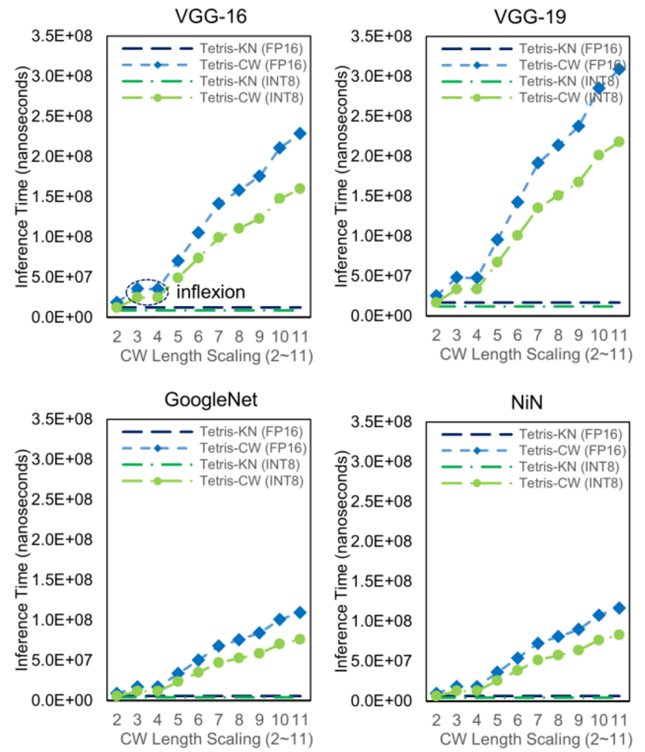

Fig. 12. Actual inference time with window size scaling. The latency of Ω module is scaled with the window size, and further influences the frequency of Tetris-CW. The larger the window size, the lower the frequency.

time. The proportion of increased inference cycles is just one metric reflecting the slack output induced by the mechanism. However, the actual inference time is also contingent on the hardware latency introduced by Ω.

In Fig. 11, we plot the cycles increment (in percentage) over Tetris-KN, with the left y-axis for fp16 precision, and the right y-axis for INT8. For various DNN models, we observe an extremely similar behavior. All the models demonstrate a "hockey" curve, with the inflexion point nearly identical on the x-axis. For example, VGG-16 and VGG-19 exhibit 7.21% and 7.27% increment for fp16 at CW size = 2, while 0.89% and 0.85% at CW size = 4, respectively. For size scaling from 4∼16, the data become pretty stable, less than 0.5% increment for each case. Similar for GoogleNet and Nin, the inflexion also emerges at CW size = 4 with 2.21% and 1.43% increment, after which the data also become stable at nearly 1.8% and 1.2% increment. For INT8, all the evaluated DNN models demonstrate tiny increment with CW size scaling, i.e., around 0.02% for VGG series, even 0.002% for GoogleNet. We can draw two conclusions from the above-observed results: 1) small window size does result in increased inference cycles, and the inflexion point is uniformly at CW size = 4 and 2) large window size settings do not lead to minimized inference cycles, which further proves that there exists an upper limit in configuring the window size. Exceeding this limit does not bring more headroom in accelerating model inference for Tetris-CW.

As the second step, we scale the window size and stat the overall inference time in each case. As illustrated in Fig. 12, an interesting observation is that the evaluated DCNN models still exhibit the hockey curve, but this time the bottom values show up at small scale, i.e., CW size = 2∼4, and climb rapidly when it scales up from 5 to 11, in sharp contrast to Fig. 11. Also two conclusions here: 1) although small CW size brings performance degradation in the absolute inference cycles, the actual inference time, however, is definitively overwhelmed by the hardware latency overhead and 2) selecting appropriate CW size should bilaterally consider inference cycles increment in Fig. 11 and the actual inference time in Fig. 12. Small size setting has shortened inference time but undermines the energy efficiency because of the increased slack output, while a large size setting demonstrates optimal inference cycles but victimized by the critical path delay. Comparatively, CW size = 4 is an appropriate option that brings both less cycle increment and actual inference time, and we will use this configuration in the rest of the experiments. The conclusion is also in correspondence with Fig. 6, in which the window size was suggested to be slightly larger than the continuity of slacks, so size 4 is supposed to be the best option as well.

B. Performance

Fig. 13 compares the inference speed of two Tetris implementations and the baselines, using real-world execution time instead of the number of cycles in Vivado HLS simulation. Here, we observe that by kneading weights Tetris-KN could achieve 3.97× speedup for fp16 and 6.96× for its INT8 mode over DaDN on average. For Tetris-CW with CW size = 4, the speedup is 3.11× and 5.26×, slightly lower than Tetris-KN. Comparatively, the other baseline PRA could achieve nearly 2.6× speedup. The benefits stem from enforcing effectual computations by weight kneading or CW mechanism. By deploying SAC instead of MAC in Tetris, these techniques

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on November 29,2021 at 06:37:09 UTC from IEEE Xplore. Restrictions apply.

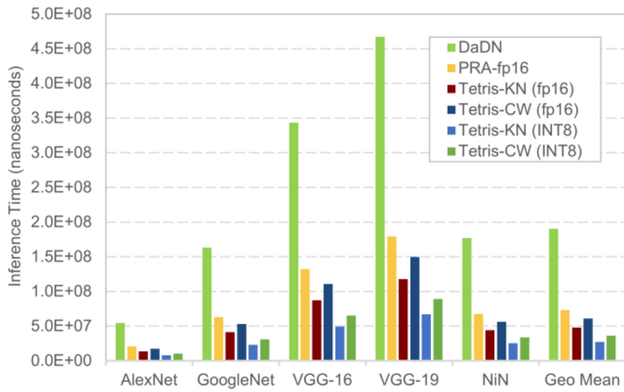

Fig. 13. Performance comparison. We use absolute inference time consumed as the representative. Lower is better.

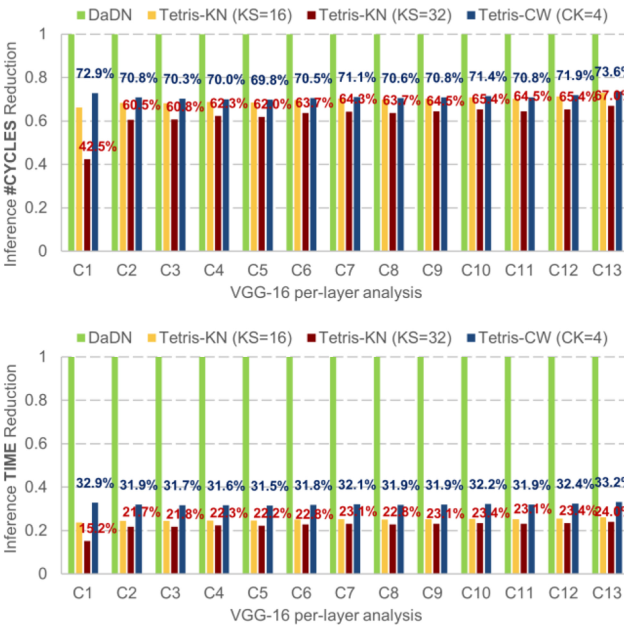

Fig. 14. Speedup analysis for each Conv layer (normalized to DaDN) under two KS configurations of Tetris-KN (fp16) and Tetris-CW ($CK = 4$).

are properly supported at the hardware level. Partial sum is not computed at pair-wise pattern like in DaDN, and dealing with activation data in batches significantly improves the inference throughput. PRA-fp16 is a bit-serial computing scheme and it must traverse the entire weight to probe the essential bit, and leverage large shifters via multiple synchronization stages to accumulate the partial sum that significantly harms the throughput. Since it also accounts for the essential bits, model inference is also accelerated but only obtains limited improvement. Tetris does not incur any of these problems. Combined with the specialized INT8 acceleration mode, Tetris further provides 1.75× and 1.68× speedup compared with fp16 mode. Fig. 14 shows per-layer speedup analysis of VGG-16 as a case study, in which we also present absolute number of cycles (upper) as well as actual inference time (lower), normalized to DaDN.

C. Energy and Efficiency

Furthermore, we evaluate the energy consumption as well as the efficiency of Tetris and the baselines. For the energy

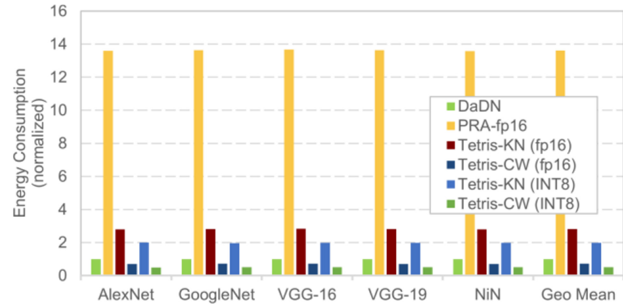

Fig. 15. Energy consumption, normalized to DaDN. Lower is better.

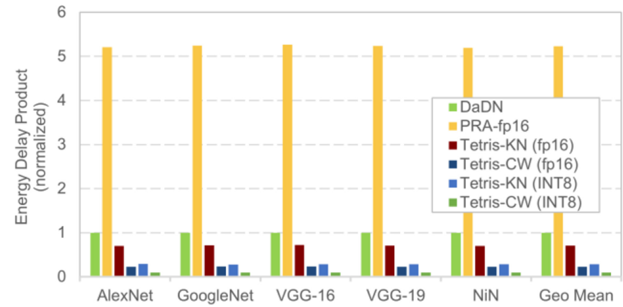

Fig. 16. Energy efficiency comparision. Also lower is better.

efficiency metric, we use the energy delay product (EDP) as the representative. After DNN has finished the inference for a single image, we record the total inference time together with its energy consumption after synthesis to compute the EDP values. To facilitate direct comparisons across all DNN models and the baselines, we normalize the results to DaDN as reported in Fig. 15. The average energy consumption is 2.81× and 1.98× higher than DaDN for the Tetris-KN designs, the increase in energy stems from the enlarged eDRAM for storing the indices. For the Tetris-CW designs, the energy consumption is 0.71× and 0.49× lower than DaDN. Therefore, we can conclude that the indices consume a large fraction of energy.

From the energy efficiency perspective as reported in Fig. 16, things are totally different: the improvement for Tetris-KN is 1.41× for fp16 and 3.51× for INT8 over DaDN. For Tetris-CW, the improvement even attains 4.37× and 10.52×. The benefit of performance enhancement does not bring with the cost of energy for Tetris. Tetris does not introduce complex hardware architecture to fulfill SAC and on the contrary, it simplifies the multiplier to pure adders. Tetris-CW does not incur the storage overhead due to indices, so its efficiency is much higher than Tetris-KN. For the bit-serial baseline, it must enroll 16× more weight buffers to compensate the throughput loss compared with DaDN, so its energy consumption increases enormously and degrades the efficiency to only 19.1% of DaDN. This is consistent with the data reported in its paper [10]. Tetris-CW outperforms PRA as 22.9× and 55.1× for fp16 and INT8 mode, respectively.

Discussion: From experiments B and C, we find that the two Tetris implementations have their own pros and cons. If inference time or performance is the highest priority in accelerating machine learning applications, i.e., in the cloud datacenters, Tetris-KN is the first choice; in other scenarios, i.e., in battery-powered edge devices, energy efficiency is the major consideration, so Tetris-CW might be the better option.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on November 29,2021 at 06:37:09 UTC from IEEE Xplore. Restrictions apply.

TABLE II
AREA OVERHEAD COMPARISON. WE EVALUATE TOTAL AREA OF TETRIS AS WELL AS THE BASELINES, TOGETHER WITH THE AREA BREAKDOWN OF 1 TETRIS PE

Area (16 PEs) in mm^2			**Area Breakdown of 1 Tetris PE (mm^2)**						
DaDN		79.36	**ITEM**	*I/O RAM	Throttle Buffer	Splitter Array	Activation Function	Segment Adders	Rear Adder Tree
PRA		153.65	**Size**	20KB/PE	5KB	16x 16 SACs	ReLu	16x 16SACs	1x 16SACs
Tetris-KN		89.76	**Area**	3.828	0.957	0.544	0.143	0.129	0.008
Tetris-CW		90.07		3.828	0.957	0.849 **(w/ Ω)**	0.143	0.129	0.008
Over head	T-KN	1.131x	**Percentage**	68.24%	17.06%	9.70%	2.55%	2.30%	0.14%
	T-CW	1.134x		67.99%	16.99%	15.08%	2.54%	2.29%	0.14%

*The I/O RAM is the on-chip eDRAM storage allocated per PE.

That also confirms the flexibility of our design discussed in Section III-B.

D. Area Overhead

Table II lists the area of Tetris and two baselines. The overall area overhead is 1.13× compared with DaDN, but Tetris has a relatively smaller area compared to PRA. PRA suffers from large weight FIFOs, because the bit serialization scheme cannot match the throughput of bit-parallel schemes like DaDN and Tetris, and large buffers must be enrolled to provide more bit-level operations simultaneously. The overhead of 16 PEs could reach 1.93× over DaDN. For the area breakdown, the major contributor for Tetris is I/O activation/weight RAM (68.24%) allocated per PE from the on-chip eDRAM, and the throttle buffer (17.06%). For functional components, splitter array (9.7%, 15.08%) dominates the chip area. Segment adders and rear adder tree do not occupy a large space with only 0.1293 mm^2 and 0.008 mm^2 each.

VII. CONCLUSION

In this article, we proposed a novel computing paradigm (SAC) as well as the associate accelerator family with two practical implementations, targeting the ineffectual computation problem in machine learning accelerators. Differed from preconceived design philosophy, Tetris does not resort to MAC operation to obtain the partial sums at the cost of tedious multiplying and shifting, and most critically oblivious to the zero bit slacks that undermine the performance and energy. The two detection mechanisms assure the essential bits involved in model inference and SAC is designed to fulfill this purpose at the architectural level. The Tetris family provide the user flexibility in selecting the accelerator according to the application scenarios. We believe that the techniques proposed in this article will motivate a reconsideration of DCNN accelerator design, or could even be applied to general-purpose computing engines like GPGPUs in the future.

REFERENCES

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556v6*.

[2] M. Lin, Q. Chen, and S. Yan, "Network in network," 2013, *arXiv:1312.4400v3*.

[3] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. Comput. Vision Pattern Recognit.*, Las Vegas, NV, USA, 2016, pp. 2818–2826.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vision Pattern Recognit.*, Las Vegas, NV, USA, 2016, pp. 770–778.

[6] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. CVPR*, Honolulu, HI, USA, 2017, pp. 2261–2269.

[7] M. Riera, J.-M. Arnau, and A. Gonzalez, "Computation reuse in DNNs by exploiting input similarity," presented at the 45th Annu. Int. Symp. Comput. Archit., Los Angeles, CA, USA, 2018, pp. 57–68.

[8] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," presented at the 44th Annu. Int. Symp. Comput. Archit., Toronto, ON, Canada, 2017, pp. 1–12.

[9] W. Lu *et al.*, "Promoting the harmony between sparsity and regularity: A relaxed synchronous architecture for convolutional neural networks," *IEEE Trans. Comput.*, vol. 68, no. 6, pp. 867–881, Jun. 2019.

[10] J. Albericio *et al.*, "Bit-pragmatic deep neural network computing," presented at the 50th Annu. IEEE/ACM Int. Symp. Microarchit., Cambridge, MA, USA, 2017, pp. 382–394.

[11] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," presented at the 43rd Int. Symp. Comput. Archit., Seoul, South Korea, 2016, pp. 1–13.

[12] A. Parashar *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," presented at the 44th Annu. Int. Symp. Comput. Archit., Toronto, ON, Canada, 2017, pp. 27–40.

[13] S. Zhang *et al.*, "Cambricon-X: An accelerator for sparse neural networks," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, Taipei, Taiwan, 2016, pp. 1–12.

[14] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Seoul, South Korea, 2016, pp. 243–254.

[15] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proc. Int. Conf. Comput. Vision (ICCV)*, 2017, pp. 1398–1406.

[16] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015, *arXiv preprint arXiv:1510.00149*.

[17] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," 2016, *arXiv preprint arXiv:1607.03250*.

[18] J.-H. Luo, J. Wu, and W. Lin, "ThiNet: A filter level pruning method for deep neural network compression," 2017, *arXiv preprint arXiv:1707.06342*.

[19] Y. Chen *et al.*, "DaDianNao: A machine-learning supercomputer," presented at the 47th Annu. IEEE/ACM Int. Symp. Microarchit., Cambridge, U.K., 2014, pp. 609–622.

[20] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," presented at the 19th Int. Conf. Archit. Support Program. Lang. Oper. Syst., Salt Lake City, UT, USA, 2014, pp. 269–284.

[21] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *IEEE Micro*, to be published.

[22] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," presented at the 45th Annu. Int. Symp. Comput. Archit., Los Angeles, CA, USA, 2018, pp. 688–698.

[23] H. Sharma *et al.*, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks," presented at the 45th Annu. Int. Symp. Comput. Archit., Los Angeles, CA, USA, 2018, pp. 764–775.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on November 29,2021 at 06:37:09 UTC from IEEE Xplore. Restrictions apply.

[24] B. Moons and M. Verhelst, "An energy-efficient precision-scalable ConvNet processor in 40-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 52, no. 4, pp. 903–914, Apr. 2017.
[25] H. Sim, S. Kenzhegulov, and J. Lee, "DPS: Dynamic precision scaling for stochastic computing-based deep neural networks," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, San Francisco, CA, USA, 2018, pp. 1–6.
[26] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "UNPU: A 50.6TOPS/W unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, San Francisco, CA, USA, 2018, pp. 218–220.
[27] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," presented at the 49th Annu. IEEE/ACM Int. Symp. Microarchit., Taipei, Taiwan, 2016, pp. 1–12.
[28] Nvidia. *TensorRT, Programmable Inference Accelerator*. Accessed: Sep. 22, 2017. [Online]. Available: https://developer.nvidia.com/tensorrt
[29] Y. Jia. *Caffe Deep Learning Framework*. Accessed: Dec. 10, 2017. [Online]. Available: https://github.com/BVLC/caffe/
[30] Synopsys. *Design Compiler*. Accessed: Nov. 4, 2015. [Online]. Available: http://www.synopsys.com/Tools/Implementation/RTL Synthesis/DesignCompiler/Pages/default.aspx
[31] Synopsys. *PrimeTime Static Timing Analysis*. Accessed: May 2, 2018. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html

Hang Lu received the Ph.D. degree from the University of Chinese Academy of Sciences, Beijing, China, in 2015.

He is currently an Associate Professor with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His current research interests include high performance and power efficient networks-on-chip architectures, many-core processors, and domain-specific accelerators.

Mingzhe Zhang (M'18) received the Ph.D. degree in computer architecture from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2018.

He is currently an Assistant Professor with the Institute of Computing Technology, Chinese Academy of Sciences. His current research interests include nonvolatile memory, near-data processing, domain-specific accelerator, and emerging technology.

Yinhe Han (SM'06) received the B.Eng. degree from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2001, and the Ph.D. degree from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 2006.

He is currently a Professor with the State Key Laboratory of Computer Architecture, ICT, CAS. His current research interests include computer architecture and chip design for intelligent robot. He has published over 110 papers in the above areas, including ISCA, HPCA, DAC, TC, and other top conferences and journals.

Prof. Han was the Program Chair of ATS'2014, the Finance/Publicity Chair of HPCA'2013/17, the Program Co-Chair of WRTLT 2009, and served on the Technical Program Committees of several IEEE and ACM conferences, including DAC'17, PACT'14, and HPCA'13. He was the Chair of Young Computer Scientists and Engineers Forum, China Computer Federation from 2016 to 2017. He is the Secretary General of CCF Technical Committee on Fault-Tolerant Computing from 2016 to 2019.

Qi Wang received the Ph.D. degree in computer science from the University of Chinese Academy of Sciences, Beijing, China, in 2015.

She is an Associate Professor with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing. In 2010, she received a 1-year fellowship from INRIA under the joint program with the Chinese Academy of Sciences to pursue her research within the SWING team of INRIA, CITI Laboratory, INSA, Lyon, France. Her current research interest includes evaluation of wireless networks for delay sensitive applications.

Dr. Wang was a recipient of the 2012 EIFFEL Doctoral Fellowship from the French Ministry of Foreign Affairs.

Huawei Li (M'00–SM'09) received the B.S. degree in computer science from Xiangtan University, Xiangtan, China, in 1996, and the M.S. and Ph.D. degrees from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 1999 and 2001, respectively.

She has been a Professor with ICT, CAS since 2008. She visited the University of California at Santa Barbara, Santa Barbara, CA, USA, from 2009 to 2010. She has published over 180 technical papers, and holds 27 Chinese Patents. Her current research interests include testing of VLSI/SOC circuits, design verification, design for reliability, fault-tolerance, and approximate computing.

Prof. Li was a recipient of the 2012 National Technology Invention Award of China. She has been served as an Associate Editor for the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS since 2015. She was the Technical Program Co-Chair of IEEE Asian Test Symposium (ATS) in 2007 and 2018, and the General Co-Chair in 2014. She was the Technical Program Co-Chair of IEEE International Test Conference in Asia in 2018. She has served as the Steering Committee Chair of IEEE Workshop on RTL and High Level Testing from 2014 to 2016, the Steering Committee Vice Chair of ATS from 2017 to 2019, the Secretary General from 2008 to 2015, and has been the Chair of the China Computer Federation Technical Committee on Fault-Tolerant Computing, since 2016. She has served on the technical program committees for several IEEE conferences.

Xiaowei Li (SM'04) received the B.Eng. and M.Eng. degrees in computer science from the Hefei University of Technology, Hefei, China, in 1985 and 1988, respectively, and the Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 1991.

He was an Associate Professor with the Department of Computer Science and Technology, Peking University, Beijing, from 1991 to 2000. In 2000, he joined ICT, CAS, as a Professor, where he is currently the Deputy Director of the State Key Laboratory of Computer Architecture. He has coauthored over 280 papers in journals and international conferences, and he holds 60 patents and 30 software copyrights. His current research interests include VLSI testing, design for testability, design verification, dependable computing, and wireless sensor networks.

Prof. Li has been the Vice Chair of the IEEE Asia and Pacific Regional Test Technology Technical Council since 2004. He was the Chair of the Technical Committee on Fault-Tolerant Computing, China Computer Federation from 2008 to 2012, and the Steering Committee Chair of IEEE Asian Test Symposium from 2011 to 2013. He was the Steering Committee Chair of IEEE Workshop on RTL and High Level Testing from 2007 to 2010. He services as an Associate Editor for the *Journal of Computer Science and Technology*, the *Journal of Low Power Electronics*, the *Journal of Electronic Testing: Theory and Applications*, and the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on November 29,2021 at 06:37:09 UTC from IEEE Xplore. Restrictions apply.