# A None-Sparse Inference Accelerator that Distills and Reuses the Computation Redundancy in CNNs

Ying Wang, Shengwen Liang, Huawei Li, and Xiaowei Li
State Key Laboratory of Computer Architecture
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, P.R. China
{wangying2009, liangshengwen, lihuawei, lxw}@ict.ac.cn

**Abstract**
Prior research on energy-efficient Convolutional Neural Network (CNN) inference accelerators mostly focus on exploiting the model sparsity, i.e., zero patterns in weight and activations, to reduce the on-chip storage and computation overhead. In this work, we found in addition to zero patterns, a larger group of repetitive patterns and values exists in the working-set of CNN inference task, which is defined as computation redundancy and induces unnecessary performance and storage overhead in CNN accelerators. Based on this observation, we proposed a redundancy-free architecture that detects and eliminates the repetitive computation and storage patterns in CNN for more efficient network inference. The architecture consists of two parts: the off-line parameter analyzer that extracts the repetitive patterns in the 3D tensor of parameters, and the dataflow accelerator. The proposed accelerator at first preprocesses the weight patterns and the dynamically generated activations, and then cache these intermediate results in special $P^2$-cache banks for further usage in convolution or full-connection stage. It is evaluated in experiments that the proposed Cavoluche architecture removes up to 89% of the repetitive operations from the layer inference process and reduce 77% of on-chip storage space to store both redundancy-free weight and activations. It is seen in experiments that the implementation of Cavoluche outperforms the state-of-the-art mobile GPGPU in both performance and energy-efficiency. When compared to the latest sparsity base accelerators, Cavoluche also achieves better operation elimination effects.

**Keywords** CNN, Accelerator, Sparsity, Cache.

## I. INTRODUCTION

Deep Convolutional Neural Networks (CNNs) are making substantial progress in the field of computer vision, image processing and speech recognition. In addition to the traditional mainframe system workloads like data analytics and retrieval, shifting the power of deep learning to mobile and edge devices is becoming a trendy research topic. As a result, people are paying attention to more compact and energy-efficient deep learning ability in low-end devices. Conventionally, in software perspective, there are a lot of general data compression approaches [1] [2], proposed to decrease the execution overhead of CNN inference for mobile CPU or GPGPU devices. This software approaches either use general data compression algorithms based on sparse matrix and quantization [2], or leverage application-level observations to eliminate the structural or functional redundancy in CNN models [1]. These are all essentially algorithmic optimization techniques [2].

The compression-related optimization also goes to hardware design paradigm [3] [4], as the advent and popularity of energy-efficient specialized deep learning architecture (DLA). These specialized accelerators have different application scenarios, and also diverse architectures including systolic array, dot-production array, 2D coarse-grained reconfigurable array, etc. [3-5]. However, due to the numerous network parameters and activations, one of the common trendy design techniques in recent DLA research is leveraging the sparsity-compression and zero-removing architecture to shrink the on-chip storage overhead and decrease the intensive computation operations. For example, EIE, Cnvlutin, cambricon-x, and SCNN and are typical sparsity-based architectures that seek to skip the ineffectual computations with zero-operand of weight or activations in either convolutional or fully-connect layers [5] [6] [3] [9]. Combined with the principled sparsification techniques in the training stage of neural networks [1], these architectures receive multifold benefits:

- Sparsity coding enables the DLAs to work with smaller on-chip buffers, e.g., they have 100-200k buffers much smaller than early designs [7]. At the same time, the power and performance penalty caused by off-chip and on-chip data moving is reduced considerably.

- Sparsity-based acceleration architectures are exhibiting higher *Gops* and especially higher *Gops-per-watt* by eliminating the ineffectual operations with zero-value parameters/activation in *conv* and FC layers.

However, there are also some unsolved issues and potential limitations for sparsity architectures. First, model sparsity is limited by the network capacity. For complicated problems, the network must maintain sufficient capacity to achieve generalization accuracy or accurate loss estimation [8]. Also, in some scenarios of the application on lightweight devices, they employ compact but dense models [9] [10], on which the sparsity-based accelerators cannot achieve significant performance improvement. Besides, the other issue in prior sparsity accelerators is that none of them can exploit the parameter and activation sparsity in different types of layers at the same time, leaving a considerable improvement space to eliminate the ineffectual computation.

In contrast, this work seeks to eliminate a different type of computation redundancy in network inference. From the hardware aspect, sparsity can be viewed as a special operand patterns (Zero-Pattern) in the working-set of neural network inference, but not the sole data patterns that could be exploited for storage compression and computation speed-up. This work found the **repetitive data patterns** other than sparsity in deep networks and exploit this inherent redundancy for more efficient inference acceleration. This pattern-based speedup mechanism has some unique advantages:

-Compared to network sparsity (zero patterns), a pattern in network is deemed to be a vector including an arbitrary number of parameters. We found that there are abundant repetitive patterns in the massive space of network parameter set, so that the production result of repetitive patterns and the network activations can be cached and reused for many times in the entire procedure of neural layer inference. Another useful feature of

*DAC '19, June 2–6, 2019, Las Vegas, NV, USA*
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06...$15.00
DOI: 10.1145/3316781.3317749

pattern-based acceleration and compression is that they do not conflict with the sparsity acceleration. When we reduce the size of network working-set to a compact group of patterns, we could still exploit the sparsity in these patterns for computation and data elimination.

-The other advantage is that the redundancy can be reused across the layers. As the network models are becoming deeper and deeper, reusing the parameters and computation results across the layers are essential to the performance scalability of very deep networks.

Therefore, in this work, we present an efficient CNN inference accelerator, Cavoluche, which aims to distill the redundant patterns in the parameter tensor of deep networks, cache the intermediate computation result of such patterns, and allow them to be shared by the operations in the inference procedure of network layers. The proposed dataflow architecture can partition the computation of net layers, i.e., convolution and full connection layer, into two stages: repetitive patterns preprocessing, and intermediate result look-up in the proposed $P^2$-cache, so that only a small portion of patterns and their production result will be stored and computed in the accelerator.

In evaluation, it is shown that the implemented Cavoluche accelerator reduces the space of on-chip storage significantly, and removes up to 87% repetitive operations in the networks. When compared to state-of-the-art GPU and sparse DLA design, the implemented Cavoluche shows a more than 500*x* and 1.21 *x* Gops/watt growth respectively.

The paper is organized as follows. Section II introduces the background and preliminaries of CNN acceleration. Section III covers the detailed architecture of the proposed Cavoluche. Section IV presents the experimental set-up and results. Section V concludes the paper.

## II. Motivational Study: the implication of computation redundancy in CNN

A typical convolutional neural network (CNN) contains stacked components of convolutional layers, pooling and full-connection (FC) layers, as well as other operations like activation and normalization functions. For such CNN models, the convolutional layers using a set of 2D pre-trained filters to convolve feature data constitutes up to 30%-50% execution time in GPU acceleration [11]. For a convolutional neural network, the inference performance of convolutional layers is decided by the involved parameter size and the volume of operations despite the hardware platform types.

$$size\_of\_par = \sum_{l=1}^{d} IN_l \cdot OT_l \cdot K_l^2$$

$$vol\_of\_op = \sum_{l=1}^{d} IN_l \cdot OT_l \cdot K_l^2 \cdot X_l \cdot Y_l \qquad (1)$$

Wherein *d* is the depth (number of convolutional layers). $IN_l$, $OT_l$ , and $K_l$ are respectively the number of input, output channels, and kernel size in the *l*-th layer. $X_l$ and $Y_l$ are respectively the width and height of the output feature maps as in Fig. 1. According to equation-(1), *size_of_par* measures the size of network parameter set and decides the memory access performance, while *vol_of_op* determines the computation time when CNNs are invoked and executed on processors or accelerators.

For CNN accelerators, equation-(1) implies the key factors deciding the performance of CNN inference. Therefore, exploiting the repetitive patterns or sparse patterns can reduce *size_of_par*, and consequently the storage overhead through data re-encoding like CSR, and it can also directly eliminate the number of MAC operations with zero or repetitive operands by decreasing *vol_of_op*.

**The observation of redundancy**: Sparsity is one resource of overhead reduction in deep CNNs. However, from the dimension reduction theories, the gigantic space of parameters can be reduced to compact low-dimension space, more specifically, the myriad of sub-vectors in the tensor of parameters, can be repetitive, i.e., exactly the same or approximate sub-vector but in different positions of the 4D tensor of *Param(in, out, k, k)*, or indirectly repetitive, which means different sub-vectors of values could be the sum of multiple shared sub-vectors. Once the repetitive patterns are extracted from the networks, the large size of parameters can be dynamically derived from a compact pattern table, indicating a drastic drop of *size_of_par*. At the same time, the intermediate result of the operations related to these repetitive patterns can also be cached and reused without repetitively generating them from the Processing Elements (PE) of CNN accelerators, decreasing $vol\_of\_op$. If we can utilize these repetitive patterns in the parameter space of CNNs, we can significantly shrink the storage, and eliminate redundant operations contributed to the prolonged execution time of CNN inference. According to the data analysis results, there are both redundant parameters patterns inside one layer and across the layers of popular networks. In our experimental training with state-of-the-art networks, it is shown that 5% of patterns re-appear for more than 16-times in the parameter tensor throughout the layers of *NiN*. The source of repetitive patterns is due to the fact that the neural parameters can be trained to converge on a very compact value distribution range [2] [11] [12]. If we intentionally regularize the network to adopt similar parameters at the training stage [1], the proportionality of repetitive patterns will be even higher. For clearer view, $SV_{i,j,im}^{ot}$ are such patterns, which constitutes the parameter tensor in Fig. 1. The explanation and source of redundancy exploitation in DNN acceleration come from the dimensionality reduction theory of neural networks [12].

## III. Architecture of Cavoluche

### A. *Extracting the repetitive patterns in parameter tensors*

As mentioned above, the 3D tensor of convolutional filters in networks can be re-constructed with a smaller group of sub-vectors ($SV_{x,y,z}$) as in Fig. 1, which are deemed as patterns in this work, so that the gigantic weight space can be reduced to a lower dimension of 3D space,

In general, we exploit two different types of repetitive patterns, and the first one is direct redundancy. Inside the tensor, the direct redundancy is from the sub-vectors that are identical or have a very close distance from one another. The other type is indirect redundancy, resulted from the fact that many sub-vectors in the parameter tensor can be re-constructed from a limited set of shared short patterns.

For one typical convolutional layer-*l* as illustrated in Fig. 1, the input *IN* feature maps are convolved by the $IN \times OT \times K \times K$ filter, $\boldsymbol{T_l}$**:**

$$\boldsymbol{T_l} = \{\boldsymbol{W_{i,j}^{in,ot}}\} \qquad (2)$$

wherein, ***i=1,2,…,K; j=1,2,…,K; in=1,2,…,IN; ot=1,2,…,OT;*** However, when we look into the tensor of $T_l$, and re-deem the basic element in it as a pattern, i.e. a sub-vector of particular length. The basic element, **SV** represents the group of partitioned sub-vectors that compose the 4D tensor of convolutional filter. Shown in Fig. 1, there are *IM* vectors aligned in the channel-direction for each *position-i,j* in the tensor, and each $SV_{i,j,im}^{ot}$ contains $\frac{IN}{IM}$ units of neural weights. Therefore, the original filter $T_l$ can also be represented as:

$$T_l = \{SV_{i,j,im}^{ot}\}$$

wherein, ***i=1,2,…,k; j=1,2,…,k; im=1,2,…,IM*; *ot=1,2,…,OT***; Thus, there are $IM \times OT \times K^2$ sub-vectors, and we expects to find that many of these sub-vectors are repetitively distributed in the 4D tensor of parameters $T_l$. With these repetitive patterns, each $SV_{i,j,im}^{ot}$ belonging to SV$\in \mathbb{R}^{IM/IN \times K \times K}$, could be represented by the pattern in an compact codebook, $C_i \in \mathrm{C}, i = 0,1,2,\dots,P,$ the codebook could be a matrix C$\in \mathbb{R}^{(\frac{IN}{IM}) \times P}$, therefore , $\boldsymbol{T_l}$ could be reduced to a $(\frac{IN}{IM}) \times P$ matrix of codebook $\boldsymbol{C}$, accompanied by the index bit-map $B_l$. Thus, we have:

$$T_l = \{SV_{i,j,im}^{ot}\} = B_l \times C \qquad (3)$$

$B_l$ is the index-bitmap for layer-*l*. $B_l$ is a $K \times K \times IM \times OT$ matrix, and the elements $b_{i,j,im}^{ot}$ in this matrix $B_l$ are all binary vectors of $P$ length, indicating whether each of the $P$ codes is needed to construct the $SV_{i,j,im}^{ot}$ in the 4D filter, significantly shrinking the dimension of the entire parameter space. The procedure of converting the parameters tensors into a compact codebook and an index map is defined as pattern distillation and conducted at the off-line stage as described in Fig. 1.

Because Equation-(3) is the typical form of truncated Principal Component Analysis (PCA) [13], at the off-line stage of pattern distillation, we employ PCA to conduct the transformation of $T_l = B_l \times C$ so that the sub-vectors $SV_{i,j,im}^{ot}$ are mapped from the original space to a lower-dimension space of $(\frac{IN}{IM})$-size variables. In the new transformed new space of $(\frac{IN}{IM})$-size variables, only the first $P$ principal components are kept as patterns and the other less significant variables are truncated. These $P$ principal components are used to construct the codebook $C$.

In this way, each sub-vector in the space of $T_l$ can be represented by the sum of $t$ patterns from the codebook C.

$$SV_{i,j,im}^{ot} = C^1 + C^2 + \ldots + C^t \qquad (4)$$

Where 0<*t*<*P*. However, if the recovery of $SV_i$ are involving a lot of patterns, the additional operations will still be costly. Thus, to ensure that $SV_{i,j,im}^{ot}$ can be reconstructed with few patterns, we employ sparse PCA to have *t*<<*P* [14].
Likewise, extracting the repetitive patterns and decomposing the 2D matrix of weight for the fully-connected layer can also be conducted similarly as in Fig.1. The weight matrix for FC layer-*d* can also be represented by the index bit-map and the shared pattern book *C*'.

After we decompose $\boldsymbol{T_l}$ into a blob of SVs, a compact codebook is to be extracted from them. This stage of pattern distillation is very important to derive the shared pattern book that recovers the model parameters.

## *B. The hardware and dataflow description of Cavoluche*

The structure of Cavoluche accelerator is organized as in Fig. 2. The on-chip memory system includes one pattern book buffer that stores the sub-vector patterns learned through PCA at the off-line stage, and an activation buffer that stores the input pixels and the intermediate activations generated in each layer. In addition to the memory parts, there is a P-tile responsible for carrying out Multiply-and-Accumulate (MAC) operations with the pattern and activations. The P-tile is a set of Processing Elements (PE) organized into typical SIMD engine with an adder tree attached. The SIMD engine of P-tile can be instructed to conduct vector-production, MAC, average/max pooling, RELU activation and comparison operation. Another critical component is the Pattern-Production ($P^2$) cache bank that stores the production between the pattern code and the activation data. This buffer set is constantly looked-up to provide the reusable intermediate Code-Act production (CAP) results for convolution operations, so that they no longer need to be re-computed by the P-tile. The CAP results will be grouped and accumulated accordingly in A-tiles to generate the activations for the next layer. The working flow of this engine is divided into four stages (①-④), denoted in Fig. 2 and described as follows:

First, the convolutional filters and FC weights are quantized into short pattern codes according to the method described in the last chapter, and the pattern codes are preloaded into the pattern buffer before the computation starts in Cavoluche.

Second, the input feature map is dynamically loaded according to the addresses generated by the Address Generation Unit (AGU). For convolutional and FC layers, the activation addressing mechanism is different. For example, because a convolutional layer needs to apply the filters to the 2D plane of input channels, the activation/pixels are fetched along the channel-direction, and the data in each channel are tiled to let the activations belonging to a convolutional window continuously aligned in the activation buffer.

After that, the pattern codes in the buffer and the activation data in Activation Queue are serially sent to the P-Tile to generate the Code-Act production (CAP) results. The generated CAP results are interleaved into *n* cache banks for later usage. At this moment, the pattern index is also loaded into the Pattern Index FIFOs. These indexes are later used to fetch the correct CAP entries from the $P^2$-cache and sum them up into partial results for different feature-map channels of the next layer. In stage-④, these partial results are summed up accordingly into a complete activation, which will be stored in the activation queue again. They can be sub-sampled at the probable pooling layer as stage-⑤.

## *C. The data mapping and data replacement in* $P^2$-cache

The size of activation set is usually not as large as the parameter set. However, it will still lead to a huge volume of CAP entries when each activation sub-vectors are convolved with *P* pattern codes. Storing all the CAP entries in $P^2$-cache will require an enormous buffer. Therefore, the intermediate CAP results must be generated in proper order, carefully mapped to the cache banks and accompanied by an entry replacement strategy, so that small cache banks can be used and allow the CAP result to be reused to the maximum extent. The CAP result mapping in the $P^2$-cache is also shown in Fig. 2.

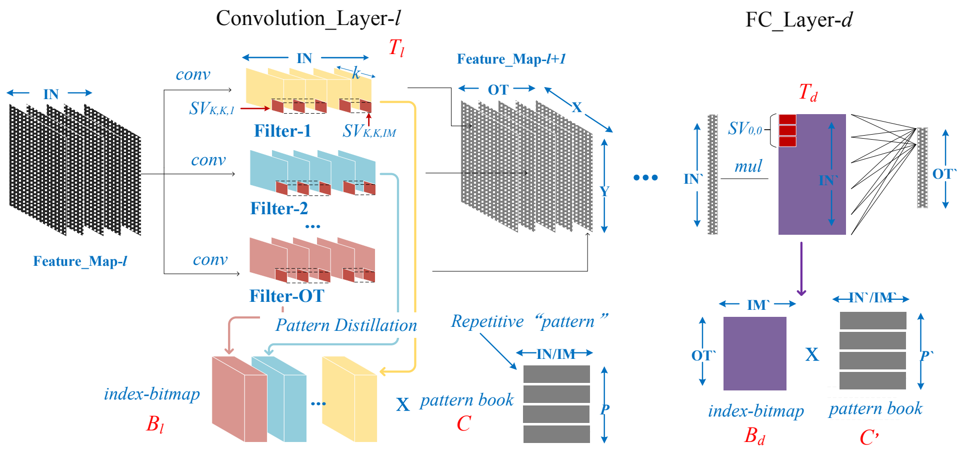
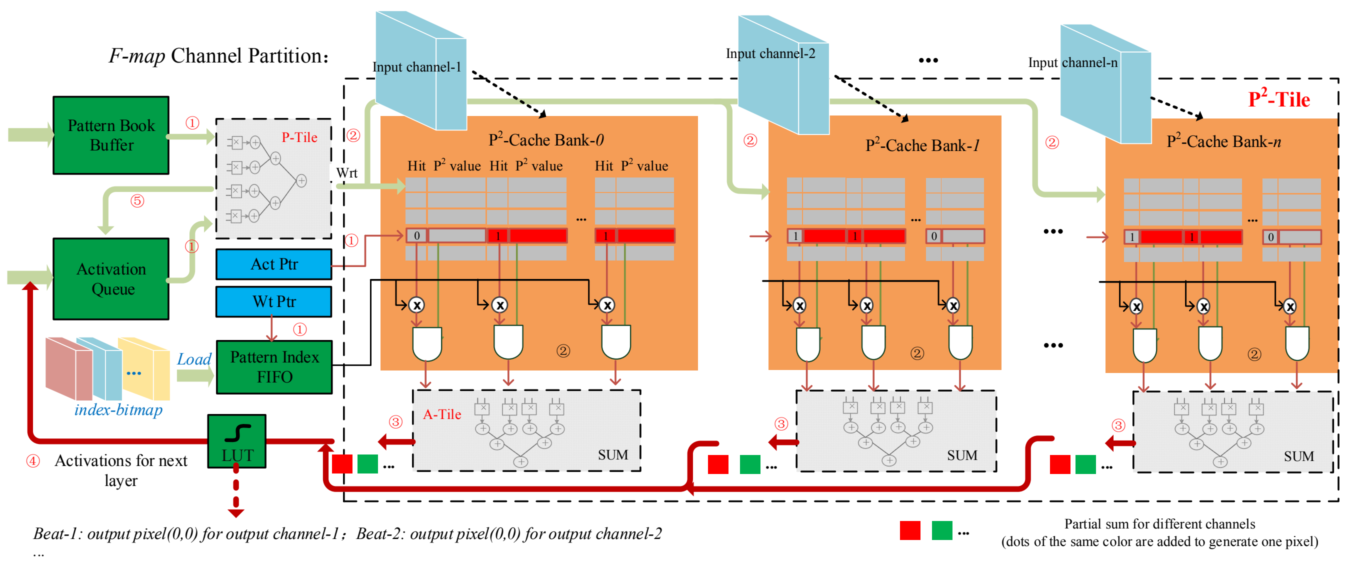


Fig. 1 Extracting the repetitive patterns for Cavoluche computation; the element in $B_l$ is a binary vector of $P$ length, and after multiplied with pattern book $C$, it becomes a IN/IM vector, which is the size of $SV_{i,j,im}^{ot}$



*Beat-1: output pixel(0,0) for output channel-1; Beat-2: output pixel(0,0) for output channel-2*
...
*Beat-IM: output pixel(0,0) for output channel-IM; Beat-IM+1: output pixel(0,1) for output channel-1; Beat-IM+2: output pixel(0,1) for output channel-2*
...

Fig. 2 Diagram of Cavoluche accelerator and the basic computation mapping in P-tiles and P$^2$-tiles

Because the activation sub-vector are serially loaded into P-tile in the channel direction, the CAP results associated with different input channels are interleaved into the *n* cache banks. Therefore, the output CAP entries from different banks must be added up to form one activation. However, the generation of activations for different output channels cannot be parallelized because the bank port can only be occupied to provide the CAP result for one output channel. For example, output channel-2 must wait for the completion of channel-1, and then reuse the same batch of CAP results cached in the banks for summing and activation operations. The batch of CAP entries that continuouslyaligned in one bank will be used to generate the associated activations in all output channels according to the *beat* shown in Fig. 2, and this data batch is deemed as a cone. A cone retires after all the associated activations have been generated, and it will be replaced by the next cone generated by the P-tile. Thus, the space of a P$^2$-cache needs only to be larger than the size of a cone.

## *D. The dataflow description of Cavoluche*

We illustrate the dataflow of the Cavoluche accelerator in Fig. 2. The design requires a dedicated compiler to generate the activation pointers (ActPtr), literally the address for P$^2$-cache indexing, at the off-line stage. At the same time, the compiler also statically generates the index bit-map and the index pointers (WtPtr). At the on-line inference stage, the ActPtr is continuously fetched and employed to obtain all the CAP entries associated to the corresponding activation sub-vector, while the WtPtr is used to load the associated code indexes that select the needed CAP entries from them to calculate the according pixel results for output channels. The static pointers are not directly stored in the memory as static addresses, and they are also indirectly generated by the address generator according to the pre-deterministic address patterns generated by the compiler as in [7]. In this way, the accelerator only stores the address patterns including the *base_addr, stride, granularity and ending_addr* instead of storing the entire address footprint on-chip [8].

Therefore, WtPtr and ActPtr in together decide the dataflow of the whole architecture when the compiler re-expresses the target network. The pseudo-code in Listing-1 outlines the inference computation loop executed on the P-tile and P$^2$-cache banks of the accelerator, and also the data mapping of input and output feature maps.

During execution, the four stages described previously are pipelined so that buffer accessing, CAP computation and activation generation are done in parallel. The codebook is fully loaded onto the pattern buffer for one layer, while the index bit-map and the intermediate activations are occasionally loaded from the off-chip DRAM memory and replaced to reuse the small buffers. To further reduce the size of cache banks, fused-layer technique is adopted [15].

Listing-1 Dataflow to infer one Conv layer in Cavoluche

***C***={0,1,…,*P*-1}: codebook

***S***: dimension of code in codebook

***Q***: bank count of P$^2$-cache

***A***: *X×Y× Z input feature map, add zero pad to make sure Z can be divided by S, so that A is X×Y× V blob of sub-vectors, Z =S × V*

***W***: *W is the conv-kernel size of this layer, the feature maps are partitioned and padded to make sure that the P$^2$-entry can be reused and the old P$^2$-entry in cache can be evicted and replaced after usage*

```
Procedure CAP Generation
For i=0,i<X, i+W
  For j=0, j<X, j+W
    For k=0, k<V, k+1
      For l=0, l< P, l+1
        For m=0, m<W, m+1
          For n=0, n<W,n+1
$CAP[i+m][j+n][k][m\cdot W+n][l]=A[i+m][j+n][k\cdot S: k\cdot S+S-1]\times C[l]$
```

***Mapping*** *The generated stream of 2-D blob* $\{CAP_{i,j}^{V,K}\}$ *is firstly partitioned into V tensors of* $\{CAP_{i,j}^{S,K}\}$*, and then interleaved the V tensors to Q banks, then each* $CAP_{i,j,s}^{,K}$ *is continuously stored into the bank in the order of i,j and s. Each of the P blocks in a cache row is a C-A product. The stale rows are evicted according to FIFO policy.*

```
Procedure Activation Generation (for one pixel in next layer)
Pattern_index_FIFO initializes the pointer to index each bank and each tile by popping out its header entry
For i=0, i<W×W, i++
  For k=0,k<V, k++
    For j=0, j<Q, j++
Act=Act+sum(Pop(bank-j)[0: P -1] × Weight_Index[i][k])
```

## E. Accelerator Implementation

**The length of sub-vector** has an essential impact on multiple aspects of Cavoluche accelerator:1. Parameter Compression ratio; 2. $P^2$-cache overhead and its access speed; 3. The utility of computation resources. Because the channel number of a *conv* filter is consistent with the input feature map, the size and number of weight sub-vectors is limited by the channel dimension. Particularly, suppose the channel number is $C$, the length of sub-vector is $m$, and the number of $P^2$-cache banks is $t$, if $c<m\times t$, there will be an underutilization of cache banks if each sub-vector production is mapped to one cache bank and all the outputs are summed together to form one activation in convolution operations. For example, the first and second *conv* layers of *Alexnet* receive 3-channel RGB input and double 48-channels respectively, while Vgg has a 3-channel input conv-1 layer and a 64-channel input conv-2 layer, which takes a considerable proportion of computation and parameter space in the entire inference procedure. Such layers cannot be partitioned into sufficiently long sub-vectors along the *z*-direction to utilize the multi-bank $P^2$-cache for higher performance.

**Compression ratio:** suppose the bit-width of weights and activations used by accelerator is 8-bit , and $w_i$ is the kernel size of layer-$i$, $l$ is the layer number, $D_i$ is the input channel number, $s_i$ is sub-vector (code) length of layer-$i$, $A_i$ indicates how many activations or pixels the input feature of layer-$i$ contains. The parameter compression ratio of Cavoluche is:

$$\frac{\sum_{i=0}^{l}\left(s_i\cdot P+\left(D_i\cdot D_{i+1}\cdot w_i^2 / s_i\right)\cdot\left(\frac{P}{8}\right)\right)}{\sum_{i=0}^{l} D_i\cdot D_{i+1}\cdot w_i^2} \quad (5)$$

If there are only $g$ code books shared by the layers ($g<l$), the ratio will be:

$$\frac{\sum_{j=0}^{g} s_j\cdot P+\sum_{i=0}^{l}\left(D_i\cdot D_{i+1}\cdot w_i^2 / s_i\right)\cdot\left(\frac{P}{8}\right)}{\sum_{i=0}^{l} D_i\cdot D_{i+1}\cdot w_i^2} \quad (6)$$

Likewise, the operation elimination factors, which indicates the ratio between the operation number in Cavoluche and that in normal accelerator for the same network, is represented as:

$$\frac{\sum_{i=0}^{l}\left(A_i\cdot s_i\cdot P\right)}{\sum_{i=0}^{l} D_i\cdot A_{i+1}\cdot w_i^2} \quad (7)$$

The above equation only contains Multiply-and-Accumulate operations and excludes the accumulation operations. One optimization trick to gain higher compression ratio and speed-up is to decrease the size of sub-vector (pattern) so that a compact pattern book can be used, at the cost of index table overhead. The other way is to re-partition the channels into more "fake channels" to make the more sub-vectors can be extracted along the channel direction (z-direction of filters) as in work [7].

## IV. Evaluation Result and Analysis

We implemented the synthesizable RTL model of Cavoluche accelerator for overhead evaluation and also built a cycle-accurate detailed simulator in C++ for fast performance and functional simulation. In this simulator, each hardware module, including P-tile, $P^2$-tile and other units, are faithfully abstracted, and the timing behavior of all components are co-verified with the synthesized RTL design. To measure the area, power and critical path delay, we synthesized the Cavoluche design using the Synopsys Design Compiler (DC) under the TSMC 65nm process technology.

**Comparison Baseline**. Because we focused on embedded machine learning application, we evaluate the network inference performance and energy-efficiency of NVIDIA Tegra X1 (TX1) with the *caffe* framework, and compare it with that of the Cavoluche accelerator. Sparsity-based accelerator like *EIE* is also compared in efficiency [9].

The total energy dissipated in GPU is measured with a power meter, while the power consumption of Cavoluche accelerator is simulated with PrimePower. The evaluated Cavoluche accelerator (Cav.) is configured according to Table.1. The basic overhead and parameters of Cav. are shown in Table.2, and it is compared with the latest sparse neural network inference accelerators. It is shown that Cav. has a relatively compact area overhead and power dissipation footprint. Though the **in-theory peak performance** of Cav. is not high as that of other designs due to fewer arithmetic units and lower clock rate, the practical inference speed of a particular CNN is much higher because of the reduced computation overhead due to redundancy elimination.

Table. 1 Configuration of baseline Cavoluche accelerator

| component | Num. | Size | Output bitwidth |
|---|---|---|---|
| PE in P-tile | 16∗4 | - | - |
| $P^2$-cache bank | 16 per tile | 4KB | 128∗16bit |
| $P^2$-tile | 8 | - | - |
| Pattern book | 1 | 4KB | 16∗16bit |
| Pattern size | - | 64B | - |
| Activation queue | 1 | 4KB | 8∗4∗16bit |

**Performance** We compare the performance of the proposed design to TX1 (GPU-cuBLAS). The inference speed of representative deep CNNs with Imagenet and Cifar-10 test-set are evaluated. It is shown in Fig. 3 that the inference speed of Cav. accelerator is 57x faster than TX1 on average, and 1.21x faster than that of state-of-the-art sparse accelerator [3]. *sparse* in Fig. 3 shows the speed-ups of *cavoluche* over our implementation of EIE simulator, which does not fully correspond to the evaluation results in [3]. It is because the performance of sparse accelerator depends on the network pruning results, literally the proportionality of zero parameters after training convergence. However, the proportionality of sparse connections is instable, often high for less complex dataset like *lenet* but not that satisfying for *Imagenet* dataset that requires higher representation power. We tried to prune the networks until it reaches at least 85% sparsity at a minor cost of accuracy losses to make the comparison look fair. The performance gains over sparse accelerator is two-fold: sparse accelerators like [3] is not that efficient for convolutional layers as for fully-connected layers, because they rely on *Img2col* operation as in *caffe* to transform convolution operations directly to matrix-vector multiplication, which induces unnecessary data duplication. The second reason is that their PE utilization is not that high because sometimes they issue sufficient non-zero weight/activation pair to drive the PEs after compression.

Table. 2 Comparison with recent sparsity-based accelerators

| | Cnv. [5] | Cam.-x [6] | EIE [3] | SCNN [4] | Cav. |
|---|---|---|---|---|---|
| Tech. (nm) | 65 | 65 | 45 | 16 | 65 |
| Freq.(MHz) | 606 | 1000 | 800 | 1000 | 800 |
| Peak perf. (GOP/s) | - | 544 | 102 | 2048 | 456 |
| Power (W) | - | 0.954 | 0.59 | - | 0.88 |
| Area (mm$^2$) | - | 6.38 | 40.8 | 7.9 | 5.85 |

*The Peak Perf does not translate to practical inference speed because of the redundancy elimination effect

Fig. 4 shows the energy efficiency of Cav. against TX1. In this figure, the energy spent by the TX1 for one-pass network inference, measured as *watt per invocation*, is normalized to that of Cav. accelerator. Our evaluation results show that GPU running the same networks consumes 600x more energy than the proposed Cavoluche on average.

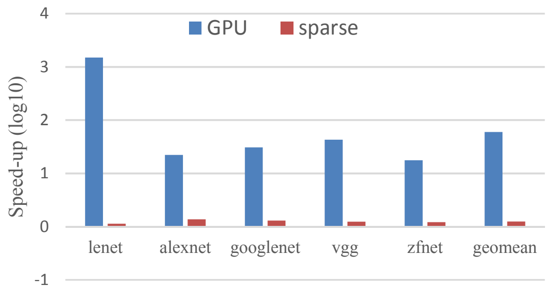


Fig. 3 Speedup of Cav. accelerator ratioed against TX1 and *sparse*

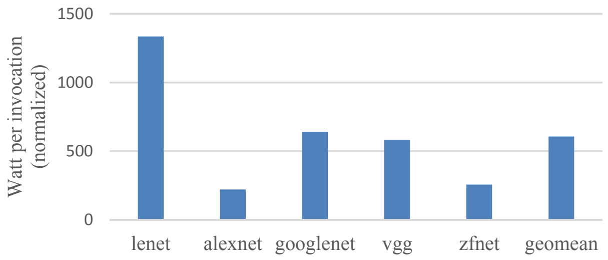


Fig. 4 Normalized Energy Efficiency of Cav.

Fig. 5 shows the mystery of performance speed-up and energy efficiency improvement brought by Cav. accelerator. It is shown in Fig. 5 that almost 82% of the add-operations and 95% of the multiply-operations are eliminated as redundancy in Cav. on average, compared to the standard implementation of network inference (GPU-cuBLAS). With far less computation load, Cav. accelerator can increase the inference efficiency significantly. We also evaluate the sparsity-based accelerator regarding the operation elimination effect, and count the volume of eliminable "ineffectual operations" with zero-operand in the work-set as in EIE [3] for comparison. When compared to the sparsity-based accelerator in our implementation, Cavoluche could eliminate 65% more redundant multiply-operations on average, and thus bring more performance improvement [3] [5].

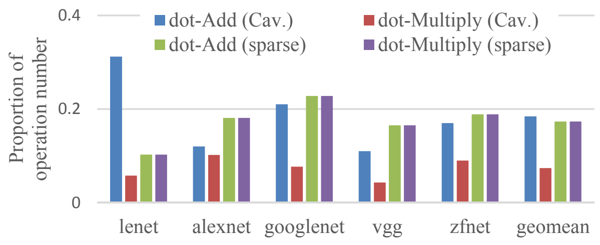


Fig. 5 Operation efficiency (ratio of operation numbers in Cavoluche/Sparse against that in baseline GPU implementation)

**Redundancy elimination** inevitably induces precision loss because not only the completely-identical patterns but also the approximate patterns are merged in computation, which is a side-effect of Cav. acceleration. However, this phenomenon can be compensated by two means: 1. Fine-tuning, repetitively distilling patterns and re-training can recover the accuracy loss thanks to the resilience of deep learning algorithm. 2. We can properly select the configuration like pattern length and code table size. It is shown in Fig. 6 that the implemented Cav. accelerator only induces 0.44% accuracy loss on average in the evaluated workloads.

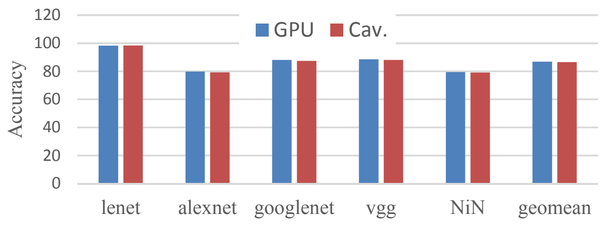


Fig. 6 Accuracy comparison

## V. Conclusion

Overall, we discovered the phenomenon of repetitive operand-patterns in the procedure of deep network inference and then proposed an energy-efficient Cavoluche accelerator to eliminate the repetitive operations in it. By functioning in a production-and-caching method, the implemented Cavoluche accelerator is proved to have significantly boosted the performance of network inference by 57x on average, and reduced over 99.87% of energy consumption on average when compared to state-of-the-art mobile GPGPU.

## VI. Acknowledgements

This work was supported in part by the National Natural Science Foundation of China under Grant 61874124, Grant 61432017, Grant 61876173, Grant 61532017, and YESS hip program No.YESS2016qnrc001. The corresponding authors are Huawei Li and Ying Wang. Thank the reviewers for their insightful comments.

## References

[1] W. Wen, et al.,"Learning Structured Sparsity in Deep Neural Networks," in Proc. NIPS, 2016.
[2] S. Han, et al., "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," arXiv preprint arXiv:1510.00149 (2015).
[3] S. Han, et al., "EIE: efficient inference engine on compressed deep neural network," arXiv preprint arXiv:160201528, 2016.
[4] A. Parashar, et al., "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks," in Proc. ISCA, 2017.
[5] J. Albericio, et al., "Cnvlutin: ineffectual-neuron-free deep neural network computing," in Proc. ISCA, 2016.
[6] S. Zhang, et al., "Cambricon-X: An accelerator for sparse neural networks," in Proc. MICRO, 2016.
[7] T. Chen, et al., "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in Proc. ASPLOS, 2014.
[8] R. Yazdani, et al., "The dark side of DNN pruning. in Proc. ISCA, 2018.
[9] S. Han, et al., "DSD: Dense-Sparse-Dense Training for Deep Neural Networks," in Proc. ICLR 2017.
[10] G. Andrew, et al.,"Mobilenets: Efficient convolutional neural networksfor mobile vision applications," In Proc. CVPR2017, 2016.
[11] A. Krizhevsky, et al., "ImageNet Classification with Deep Convolutional Neural Networks," in Proc. NIPS, 2012.
[12] Y. Gong, et al.," Compressing Deep Convolutional Networks using Vector Quantization," in Proc. ICLR 2015.
[13] R. Hadsell, et al., "Dimensionality Reduction by Learning an Invariant Mapping," in Proc. CVPR, 2006.
[14] A. d'Aspremont, et al., "Full regularization path for sparse principal component analysis," in Proc. ICML, 2007
[15] M. Alwani, et al.," Fused-layer CNN accelerators," in Proc. Micro, 2016.