2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)

Bit Prudent In-Cache Acceleration of Deep Convolutional Neural Networks

Xiaowei Wang, Jiecao Yu, Charles Augustine†, Ravi Iyer†, Reetuparna Das
University of Michigan †*Intel Corporation*
{*xiaoweiw, jiecaoyu, reetudas*}*@umich.edu,* {*charles.augustine, ravishankar.iyer*}*@intel.com*

***Abstract*—We propose Bit Prudent In-Cache Acceleration of Deep Convolutional Neural Networks - an in-SRAM architecture for accelerating Convolutional Neural Network (CNN) inference by leveraging network redundancy and massive parallelism. The network redundancy is exploited in two ways. First, we prune and fine-tune the trained network model and develop two distinct methods - coalescing and overlapping - to run inferences efficiently with sparse models. Second, we propose an architecture for network models with a reduced bit width by leveraging bit-serial computation. Our proposed architecture achieves a 17.7×/3.7× speedup over server class CPU/GPU, and a 1.6× speedup compared to the relevant in-cache accelerator, with 2% area overhead each processor die, and no loss on top-1 accuracy for AlexNet. With a relaxed accuracy limit, our tunable architecture achieves higher speedups.**

***Keywords*-In-Memory Computing; Cache; Neural Network Pruning; Low Precision Neural Network.**

I. Introduction

Convolutional Neural Networks, or CNNs, have become increasingly popular during the past two decades. Numerous network models have been designed for a wide variety of tasks. Behind the proliferation of neural network development is the increased compute capability of modern hardware. In addition to commodity CPU and GPU, customized accelerators [1], [2], [3] have been developed to achieve better latency, energy efficiency, and throughput for CNN workloads.

Another acceleration paradigm is based on in-memory architectures [4], [5], [6] that both reduces data movement and leverages massive parallelism. For instance, Neural Cache [6] is a promising design that repurposes SRAM arrays in the last-level cache of general purpose processors to create over a million bit-serial ALUs and leverages them to accelerate CNN computation. In contrast to custom accelerators, a general purpose processor cache based solution improves performance of many other workloads when not functioning as a CNN accelerator.

Unfortunately, Neural Cache architecture is *unaware* of the *redundancy* in neural networks. Neural network compression has been developed in the recent years, and it provides promising opportunities to reduce the redundancy significantly. Specifically, *weight pruning* [7], [8] and *low-precision* [9], [10], [11] are two popular approaches for neural network compression.

This paper poses the question: how can memory-centric architectures, such as Neural Cache, benefit from removing the redundancy in CNN computation? Further, we compare the two redundancy elimination approaches of weight pruning and low-precision weights, and analyze the performance/accuracy trade-offs of these two related approaches.

There are several challenges for exploiting the sparsity of pruned neural networks in Neural Cache. The vector parallelism in SRAM arrays requires that computation cannot be skipped even if only one of the vector elements needs it. This is the case for pruned networks where multiple channels are calculated in parallel and sparsity is distributed across channels. We solve this problem by developing techniques which create dense computation by coalescing non-zero filter channels. Thus, a reduced but dense compute structure amenable for vector processing is created. Filter channels are gathered into a dense format using a new offline pruning and retraining process, while input channels are gathered dynamically at runtime using a new hardware coalescing unit.

Unfortunately, coalescing can increase input loading time. In CNN computation, one input activation map is reused across several different filters to create several output channels. Neural Cache broadcasts (using in-cache intra-slice bus) the same input data to several filters, keeping input loading time minimal. After pruning, we encounter a situation that different filters have been heterogeneously pruned such that the same input data can no longer be broadcasted to all filters, leading to a multi-fold increase in input loading time. We tackle this problem by input-loading aware pruning and exploring a filter channel overlapping technique which does not change input data mapping from original unpruned network models.

Finally, we develop efficient compute techniques for low-precision neural networks. In-cache bit-serial compute paradigm naturally takes advantage of low-precision input activations, since compute cycles of bit-serial algorithms scale down with bit-width. However, we observe that bit-serial multiply-accumulate is inefficient when weights are ultra-low precision such as ternary or binary, due to the extra starting rounds for multiplication. We redesign the process of multiply-accumulate for ternary/binary weights to combine the multiplication and accumulation in fewer cycles using logical operations.

In summary, this paper makes the following contributions:

- We develop techniques to leverage redundancy preva-

2378-203X/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCA.2019.00029

lent in DNNs for memory-centric vector architectures such as Neural Cache. Redundancy can be eliminated by pruning weights or using lower precision.

- We analyze inefficiencies of convolution with pruned sparse models, and propose novel techniques to enable dense computation amenable for vector processing with sparse DNN models. Specifically, we explore a pruning method to coalesce non-zero filter channels and develop a hardware block that dynamically coalesces the input activations to align them with pruned weights. We also develop a pruning method for overlapping multiple filters together to effectively utilize the vector units. This method does not require input coalescing at runtime.
- We design new bit-serial in-SRAM compute algorithms to exploit the ultra-low precision binary and ternary network models. Convolutions with ternary/binary weights are converted to logical operations and additions.
- We compare the proposed sparsity-aware architecture with pruned models to the proposed low-precision architecture. The related performance/accuracy tradeoffs are analyzed.
- The evaluated sparsity-aware architecture achieves a 17.7×, 3.7×, 1.6× speedup over server-class CPU, GPU, and Neural Cache baselines without accuracy loss; in terms of energy efficiency, the design is 34.0×, 13.8×, 1.6× better than CPU, GPU, and Neural Cache. We further show that with accuracy loss allowed, the low-precision architecture has a latency 43.6×, 9.1×, 3.9× better than CPU, GPU, and Neural Cache.

II. Background

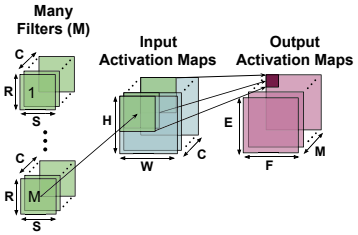

Figure 1. Computation of A Convolutional Layer.

A. Convolutional Neural Networks

Convolutional Neural Network (CNN) is a machine learning method that takes a 3D-array as input, performs convolution and non-linear functions on the array iteratively, and extracts the array information for further tasks such as classification. A typical CNN consists of many convolutional layers, a few pooling and fully connected layers, and non-linear functions between the layers. The overall procedure of a convolutional layer is shown in Figure 1. A convolutional layer takes in C channels of 2D activation maps as input, with activation of each channel having H pixels in height and W pixels in width. A batch of in total M 3D-filters are the network parameters of the layer. Each 3D-filter has C channels, each channel with $R \times S$ parameters called weights, where R is the filter height and S is the filter width. At convolution, the C channels of $R \times S$ filters are overlaid on the C channels of inputs. The $R \times S \times C$ input pixels are element-wise multiplied with their corresponding filter weights, and then the products are summed up across all C channels to produce one output pixel as shown in Figure 1. The $R \times S$ filter slides over the $H \times W$-sized input activation with stride U, generating $E \times F$ output pixels per 3D filter. Therefore the M filters create a total of $M \times E \times F$ output pixels for one convolutional layer and these pixels become inputs of the following convolutional layer. It has been shown that among the end-to-end CNN computation, most (90%) of the time is spent on convolutional layers [1]. Therefore this paper focuses on the acceleration of convolutional layers.

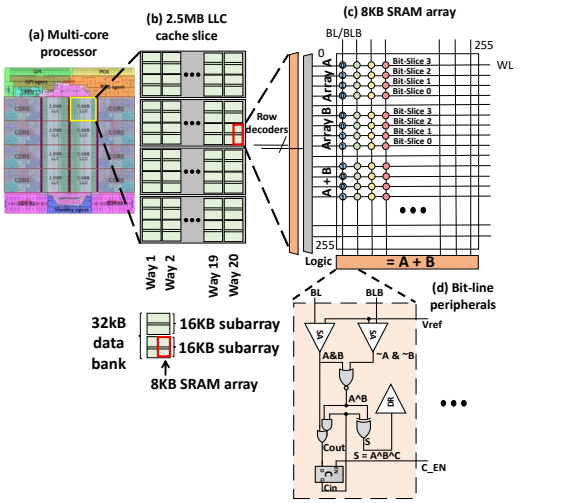

Figure 2. Neural Cache Architecture Overview.

B. Neural Cache

Neural Cache [6] is an architecture which repurposes the last level cache in general purpose processors to perform massively parallel in-SRAM computing for neural networks. Figure 2 shows the overall architecture of Neural Cache. The building block of Neural Cache is bitline computing enabled by SRAM array peripherals. In bitline computing, two wordlines of an SRAM array are activated at the same time and by sensing the shared bitline pairs (existing in Xeon LLC), logical operations (`and` and `nor`) on the cell data of the two wordlines can be performed [12], [13].

To perform arithmetic operations such as addition and multiplication, a bit-serial architecture is utilized (Figure 2 (c)): data are mapped to a transposed layout where different bitlines hold data from different elements in the operand vector. Each n-bit element is stored across n wordlines, and thus each wordline holds one *bit-slice* from all the vector elements. The bits in each bit-slice are of the same bit position. The arithmetic operation is done bit-slice by bit-slice – for example, to compute $A + B$, bit-slice 0 of vector A and bit-slice 0 of vector B are first activated and added

by the peripheral logic (generating sum and carry), then the same operation is done to bit-slice 1, 2, ..., $n-1$ of the two operand vectors (carry taken from the previous bit-slice step). For n-bit integer addition, bit-serial computation takes n cycles; similarly, n-bit integer multiplication can be done in n^2+3n-2 cycles. The weights and activations are linearly quantized to 8-bit integers for bit-serial computation.

In the above architecture, 256 bitlines in one 8 kB SRAM array are turned into 256 ALUs in a vector unit. Xeon's 35 MB LLC can accommodate 4480 such 8 kB arrays. Thus up to 1,146,880 elements can be processed in parallel, while operating at frequency of 2.5 GHz at runtime. By repurposing memory arrays, the above throughput is achieved for a cost of 7.5% area increase of an SRAM array and less than 2% area overhead for the entire Xeon processor die. Note, while a 35 MB LLC cache access from core takes 20-30 ns, the smaller 8 kB SRAM arrays can themselves operate at a frequency up to 4 GHz [14], [15].

Neural Cache uses the cache structure of 2.5 MB Xeon LLC slice [14], [15], [16] for demonstrating its idea. For fairness, we build our architecture upon the same cache structure as described below. Multiple slices on a processor die are connected by a ring. Each slice has 20 columns which serve as 20 different ways of the set-associative LLC. Each way consists of 4 banks and each bank has 4 SRAM arrays of 8 kB. The 20 ways are connected by a 256-bit bus within the cache slice. The ways and banks in a slice are shown in the left hand side of Figure 3.

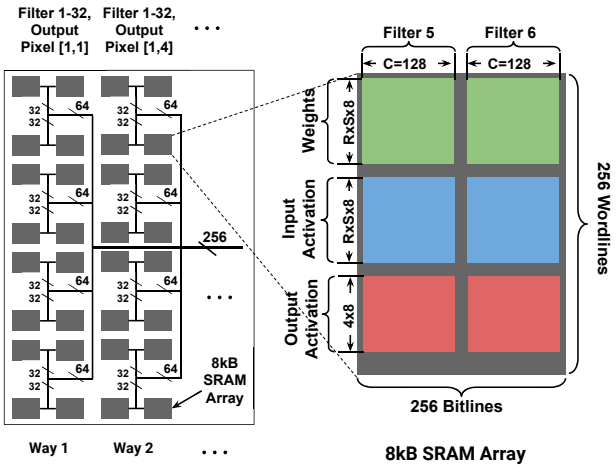

Figure 3. Neural Cache Data Mapping of a 2.5MB Slice.

Figure 3 shows a typical data mapping scheme of one convolutional layer. In each 256×256 array, the filter weights are stored in $R\times S\times8$ wordlines, and the input activations to be multiplied with weights are loaded to another $R\times S\times8$ wordlines; each bitline corresponds to one input channel. The M filters span multiple arrays in the same or neighboring ways. One way of each slice is reserved for storing the outputs of the previous layer and another way reserved for system background processes. The other 18 ways hold the replicated weights for in-cache computation.

The in-cache convolution is performed in the five successive stages as described below:

1. Weight Loading: At the start of each layer, the filter weights are loaded from DRAM to the cache. Mapping a 2D filter to each bitline does not result in full utilization of all bit-serial compute units. Thus filters are replicated throughout all the ways and then slices to exploit parallelism. For example, for AlexNet layer conv3, without replication, only 98,304 bitlines are used for computation, resulting in 10% utilization. With replication, the utilization increases to 95%. The inter-slice and intra-slice interconnect structures allow low-cost replication of weights using broadcasts. The weights from all the M filters are broadcasted to all the slices via the inter-slice ring, and then to all the ways via the intra-slice bus. After filter replication, the output pixels that still cannot be computed in parallel are computed in serial.

2. Input Loading: Each slice computes a tile (subset) of $E\times F$ output pixel positions across all M channels. The pixel positions with neighboring heights and widths are mapped to the same slice. Therefore, the output activations generated in one slice can be used as the input activation of the next layer. In input loading, the input activations are broadcasted from the reserved way to all the compute ways via the intra-slice bus. A small portion of the activations (at the border of tiles) are transferred through the inter-slice ring.

3. MAC (Multiply-ACcumulation): After data loading, at each array, the $R\times S$ weights multiply with the $R\times S$ inputs sequentially; after each multiplication, the product is accumulated into the partial sum in the reserved wordlines in the array.

4. Reduction: To sum up all the input channels of each filter, the partial sums at different bitlines are added up in the reduction stage. In reduction, at each array, partial sums of half the channels to be reduced are copied to another set of wordlines and aligned channel-wise with the other half of channels. Then the second half of partial sums are added into the first half. The copying and addition are called one round of reduction; the reduction rounds are conducted iteratively until the final reduction result is calculated.

5. Output Transfer: After reduction, the output activation maps at the compute arrays are transferred to the reserved array in the stage of output transfer. Then the inputs at a different height or width are loaded in, and the MAC, reduction, and output transfer repeat.

C. Sparsity and Reduced Precision in CNNs

Neural networks demand a large amount of compute resources. The number of operations required per convolutional layer is proportional to $R\times S\times C\times E\times F\times M$. Previous literature proposed two methods to reduce the computation requirements: 1) model pruning to reduce parameter size [7]; 2) low-precision computing [9], [11], [10].

Sparsity in CNNs: Pruning is based on the observation that

a typical network has many weights whose values are close to zero [7]. If these near-zero values are collapsed to zero, then their multiplication and addition with inputs can be skipped so that the total operations for a convolution and data loading time will decrease.

A typical procedure of pruning [7] is briefly described as follows. First, the weights to be pruned are determined - weights from the pre-trained model are evaluated by a type of regularization (for example, absolute value), and the weights with regularization less than a threshold are chosen to be pruned. Second, the pruned model is retrained to learn the values of the remaining weights in the network. The steps of weight pruning and retraining can be iteratively performed to increase model accuracy. After pruning, the network becomes sparse, and the percentage of the weights pruned is the *pruning rate*. Han *et al.* [7] report an overall 63% pruning rate on convolutional layers of AlexNet.

However, if pruning happens at the granularity of each individual weight, previous literature [8], [17] suggests that the network sparsity cannot be easily exploited due to either the extra decoding stage of compressed weights or the low utilization rate of SIMD functional units. Many works therefore propose structured pruning at multiple levels [8], [18], where the weights with one or more shared coordinate index in the $R \times S \times C \times M$ array are grouped together, and *each group either gets pruned entirely or remains entirely*.

Reduced Precision CNNs: It is known that on general-purpose hardware, integer operations are cheaper than floating point operations. Therefore, it would be beneficial to convert floating point values of weights and activations into discrete integers and compute with the integer representations. Such conversion is called quantization. The quantization of filter weights can be done statically before the inference, and the quantization of input activations is done dynamically. In practice, 8-bit quantization of weights and activations usually has negligible effects on accuracy. For bit widths less than 8, researchers still achieve a reasonable accuracy with proper quantization schemes [9], [10], [11].

The bit-serial compute diagram of Neural Cache makes low bit-width arithmetic extremely efficient. The cycles for addition scale down linearly with bit width, and the multiplication cycles are proportional to the product of the two operands – thus scaling down quadruply with bit width. Our proposed architecture fully leverages the benefits of low bit-width operands.

III. Sparsity-Aware Architecture

The sparsity in filter channels can be leveraged to eliminate the energy spent on computation with zero-valued filters, and speed up the convolution by filling up the in-SRAM compute slots with *effective* computation. Neural Cache architecture does not leverage sparsity. Each SRAM array in Neural Cache is repurposed to function as a vector unit with 256 SIMD slots. Sparsity in filter channels, when

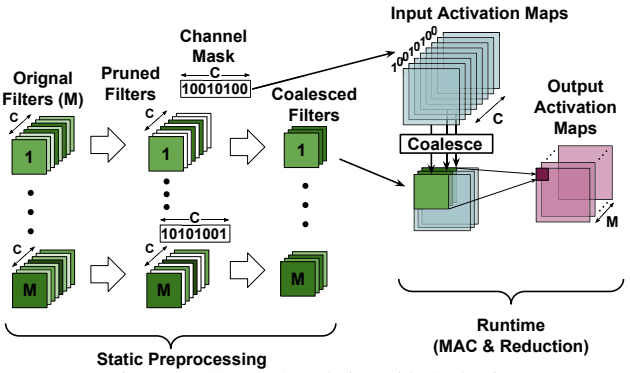

Figure 4. Sparse Convolution with Coalescing.

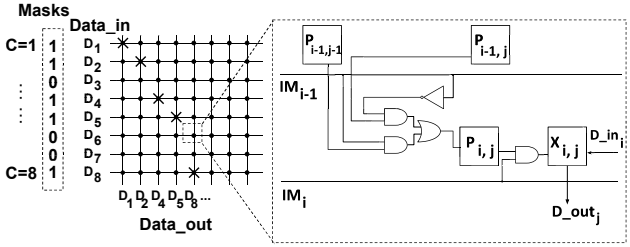

Figure 5. Architecture of Coalescing Unit (CU). Left: Crossbar and example data before coalescing (eight channels - D_1 to D_8) and after coalescing (D_1, D_2, D_4, D_5, D_8). Right: Reconfiguring Peripheral for one switch connectivity. $X_{i,j}$: Output of RP to Crossbar, whether the switch at (i, j) is connected. $P_{i,j}$: Boolean value of whether the position (i, j) is on the "path." IM_i: Channel mask for the i-th channel.

stored in a dense format, leads to wasteful computation over bitlines because some slots in the vector unit are being utilized to compute zero. We propose two techniques to avoid this, as discussed below.

First, we propose to coalesce all non-zero filter channels into consecutive SIMD slots (i.e., bitlines). This data-mapping creates a dense structure from sparse filters. The filters can be coalesced statically apriori during the offline pruning/training phase. However, input channels or activations are not known until runtime and need to be coalesced dynamically. We propose an area efficient coalescing unit to dynamically coalesce the input channels. *Second*, we explore the idea of overlapping non-conflicting filter channels in different 3D filters. There are M possible 3D filters. The advantage of this technique is that input channels do not require coalescing or any additional encoding during runtime. The disadvantage is that the pruning rate achieved is lower due to additional restrictions imposed by the overlapping requirements.

In this section, we describe the details of the above techniques, namely sparsity with coalescing and sparsity with overlapping.

A. Sparsity with Coalescing

Figure 4 provides an overview of the procedure of sparsity-aware computation by coalescing channels. We follow the structured sparsity approach [8] for pruning and use the granularity of a filter channel for pruning. Our pruning algorithm described in Section III-A3 selectively prunes

unimportant $R \times S$ 2D-filters. The choice of 2D filter as a granularity of pruning finds the desired balance between achieving a high pruning rate, and keeping the encoding of the pruned filters simple. Each compressed 3D filter is equipped with a channel mask which distinguishes the non-zero channels from zero channels. Different 3D filters are permitted to have different sparsity patterns (and different channel masks) as shown in the figure.

We propose a dynamic coalescing process that helps to align input activations to heterogeneously pruned filters (conceptually shown in right hand side of Figure 4). Before coalescing, the input data has C channels with each of them being an $H \times W$ activation map. There are M sets of different $R \times S \times C$ filters for one convolutional layer, and the inputs are coalesced according to the M filter pruning patterns. After pruning, each of the M filters is equipped with one channel mask vector of C bits, where each bit indicates whether the 2D $R \times S$ filter at the corresponding channel is pruned (0 for pruned channels, 1 for unpruned channels). During coalescing for the m-th filter, if Mask(c,m) = 1, then the c-th input channel is copied to the coalesced input data; otherwise the c-th input channel will not appear in the coalesced input data. Therefore, after coalescing, only the input channels that correspond to unpruned filter channels are preserved; such input channels and filter channels are automatically aligned. Section III-A1 provides details of the coalescing process.

After coalescing, convolution is performed only on the dense channels of the filter and activation as described in Section III-A2.

*1) **Dynamic Input Coalescing***

To implement dynamic input coalescing, we propose the design of *Coalescing Unit* (CU) and place the CUs in the reserved way. Each CU is connected to one 16 kB subarray, and is placed between the sense-amplifier and the intra-slice data bus connecting multiple ways in a slice. In total 8 CUs are required per L3 slice. Note, input activations are broadcasted from the reserved way to all the compute ways via the data bus in Neural Cache. Thus placing CUs only in the reserved way is sufficient. The activation data are coalesced right before being transferred on the bus, making the incoming input data at compute arrays already aligned with the coalesced filters, with a minimal hardware overhead of CUs.

The Coalescing Unit consists of a $256\ bit \times 256\ bit$ crossbar, reconfiguring peripheral (RP) which determines which inputs of the crossbar are connected to which outputs, and a FIFO buffer.

Crossbar: At the input interface, 256 bits from 256 input channels are fed to 256 data_in ports. The 256×256 switch connectivity configuration is initialized once per layer by RP. With the configured switch, the output of crossbar is an ordered gather of the input data bits with corresponding mask value 1. With the crossbar, the output can be any combination and arrangement of the 256 bits, so the inputs can be coalesced under any sparsity pattern. Figure 5 shows the structure of a Coalescing Unit with a smaller 8-bit input/output vector size, as well as the circuits of RP for one switch value output. In this example, among the 8 input channels, the channels 3, 6, 7 are pruned, so the outputs are data from the channels 1, 2, 4, 5, 8.

Reconfiguring Peripheral (RP): The RP can be implemented in combinational logic. The right side of Figure 5 shows how one switch value produced by RP is related to the neighboring switch values; such circuit is repeated at each cross-point to generate the 256×256 switch value array. The key idea is: if one switch is set, then the switch to its *right-down* may be set; else, the switch to its *down* may be set. We call such a dependency chain a *path*, and the switches on the path have the potential to be set. Whether a switch on the path is set or not depends on whether the corresponding channel mask is 1. In Figure 5, $P_{i,j}$ denotes whether a switch is on the path; $X_{i,j}$ is the actual switch value of the position (i, j) produced by RP. IM_i is the input mask bit of the channel i, representing whether the input channel i should be passed to the crossbar output.

Formally, the logic of $P_{i,j}$ and $X_{i,j}$ is:

$$P_{i,j} = (P_{i-1,j} \wedge \overline{IM_{i-1}}) \vee (P_{i-1,j-1} \wedge IM_{i-1})$$

$$X_{i,j} = P_{i,j} \wedge IM_i$$

$P_{0,j}$ (the first row) do not depend on any other switches and will be initialized with preset value. $P_{i,0}$ (the first column) can also be overwritten with initial value besides depending on the switch $P_{i-1,0}$.

When there are multiple 3D-filters mapped to the same array after pruning, the input masks of different filters are passed to the RP sequentially and the switch values are updated sequentially for each filter. Also, the starting point of the path is set according to the actual starting position of the current filter in the output. For example, if the first filter has 32 unpruned channels, then when the RP calculates switch values for the second filter, $P_{0,32}$ is set to 1 as the first row initialization. The number of unpruned channels for each filter can be stored alongside the filters (8 bits for a 256-channel filter). The upper triangle of the crossbar is used in this case, to coalesce the activations starting from the second filter mapped to the array.

Note that this whole RP configuration process is done once per layer. This is because even with multiple output pixels to compute in serial, only the input pixel position changes; the indices of the effective input channels remain the same.

FIFO buffer: Due to the column muxing for the bitlines, 32 bits of data are read out of the SRAM array, and the data bus for each array is also 32-bit wide. However, the interface of the crossbar is 256-bit wide. Therefore, the FIFO buffer serves as a bridge for balancing the different bandwidths

between the data bus and the crossbar, and between the crossbar and the SRAM array. It collects every 32 bits from array sense-amps and sends in 256-bit data to the crossbar. At the output of the crossbar, the FIFO buffer transmits the 256-bit data to the bus in 32-bit chunks.

The crossbar size of 256×256 can handle filters with up to 256 channels. To accommodate filters with more than 256 channels, we simply split the filters into smaller ones, each with only 256 channels, and treat them as multiple sequential filters. The normal convolution steps can proceed until the reduction ends. The reduction results of each 256 input channels can be summed up in the end.

*2) **Convolution Operation***

With the coalesced input channels, a few modifications to the convolution procedure are proposed regarding the irregularity in data mapping. We describe these changes below.

(a) Weight Loading: The filter weights of the pruned model can be coalesced offline after training, before the inference process starts. All the weights in the pruned channels are eliminated so they are no longer loaded from DRAM to the cache. The weight loading time is therefore reduced proportionally to the percentage of channels pruned.

(b) Input Loading: With the input-loading aware structured pruning design (details in Section III-A3), to compute output pixels in one position, it is sufficient to load the coalesced inputs *once* from the reserved way to all the compute ways via the intra-slice bus. The total time for input loading will remain the same as baseline, mainly because the total amount of transferred data does not change. The incurred overhead to input loading includes CU latency and the time to load the masks for coalescing. The CU latency is short when compared to bus transfer; the data size of masks ($C \times M$ bits are required per layer) is small compared to weights.

(c) MAC: The total number of output pixels computed serially is reduced because fewer input channels need computing. Hence, more output pixels can fit in the same number of bitline computing slots, reducing the output pixels computed in serial. Thus the total time on MAC is reduced. Note, the standard MAC algorithm is unaffected by coalescing, since the filters are pruned at the granularity of each channel, instead of individual weight.

(d) Reduction: The reduction stage needs modification because the boundaries for filters are irregular now. We design a "preparing round" to tackle this difficulty. The core idea of preparing round is to perform initial reductions within each filter until there is enough space to fit in all the filters. Two sets of fresh wordlines are allocated for performing copying and addition. The eight 32-bit segments in a 256-bit wordline are sequentially copied to the newly allocated wordlines, where they match up for addition. The 32-bit segment size is a result of 8-way column multiplexing of SRAM arrays. When copying a segment consisting of weights from more than one filter, the 32-bit segment is first AND-ed with a mask to selectively copy the weights only from the desired filters. After the preparing round, the filters are lined-up ready for normal reduction. The reduction is done in parallel for all the filters within an array. For filters that span more than one array, the reduction within each array is first done, before intra-array results are added up finally.

For example, in AlexNet layer conv3, C=256. In one array, there are 3 different filters for reduction, and the corresponding bitlines are BL1-BL96, BL97-BL160, BL161-BL256. Each partial sum takes up wordlines WL1-WL32. In the preparing round, 64 new wordlines (WL33-WL96) are allocated, and the partial sums of the n-th filter are copied to these 64 wordlines of the n-th bitline quarter. For instance, for the first filter, BL1-BL64 are copied to WL33-WL64 of BL1-BL64; BL65-BL96 are copied to WL65-WL96 of BL1-BL32. Then the second half of new wordlines are added into the first half. After such starting round, each filter takes up 64 bitlines and the remaining reduction can be done as in Neural Cache.

*3) **Weight Pruning Method***

Similar to the previous pruning techniques [7], [8], we prune the weights that would have the least impact on final inference results. We use the L2-norm (sum of squares of all elements) of each 2D filter to measure its importance. All the 2D-filters with an L2-norm lower than a given threshold are pruned. This criterion is straightforward and also effective in our experiments. Other criteria for weight importance measurement can also be directly applied to prune networks at the same granularity.

The pruned models will be fine-tuned to regain the accuracy. A $[C \times M]$-bit mask is generated for each convolutional layer to indicate the pruned 2D filters. All masked weights are fixed to 0 during the retraining. The pruning and retraining steps will be performed layer by layer to achieve the best accuracy. Also, the pruning rate will be gradually increased until the targeting accuracy cannot be met.

To avoid the irregularity in input activation loading, we propose a different, more structured, criterion for pruning. To minimize the number of on-bus data transfers, it is desired to have all the SRAM arrays connected to the same data bus to get the exact same channels after pruning. Note that in each 16 kB sub-array, the two 8 kB arrays share the same sense-amplifiers, and thus using the same pruning pattern for such two arrays is beneficial. We denote the eight 8 kB arrays within the same way and at the orthogonal dimension to the dimension of the shared sense-amps as a *half-way*, and the channel masks need to be the same for all the half-ways.

The procedure for the more structured pruning is as following: **(1)** According to the pruning rate, calculate the number of channels in all filters after pruning. Dividing this total channel count by the bitlines (channels) in a half-way,

we can get the number of half-ways, Y, required for one output pixel position. **(2)** Divide the M filters equally into Y half-ways. Group the channels that have the same position within a half-way together. Then there are $C \times M/Y$ such channel groups. **(3)** Calculate the L2-norm of each channel group and order them. The groups with the lowest L2-norm will be pruned until the pruning rate is met.

The above pruning procedure is necessary for efficient dynamic coalescing. Without the customized pruning approach, the weight sparsity rate is so low that there is little opportunity to coalesce. The pruning and retraining process is offline, so the inference performance is not affected.

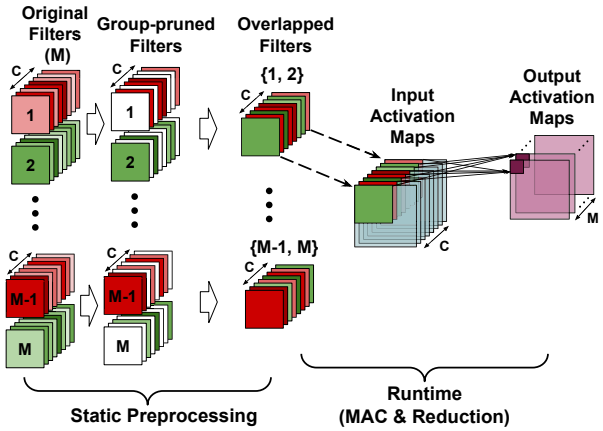

Figure 6. Sparse Convolution with Overlapping.

B. Sparsity with Overlapping

Overlapping is another technique we design to leverage sparsity. The key insight is that, if different filters are sparse at different channels as a result of customized pruning, then they can be overlapped as one filter when doing MAC operations. Figure 6 provides an overview of the convolution process with filter overlapping. During offline training, the filters are grouped and pruned so that multiple filters can be combined into one. At runtime, the compressed filters directly perform a 2D-MAC with the input activation maps – channels are naturally aligned; at reduction, the partial sums from different filters are gathered correctly according to the channel mask vector.

1) *Convolution Operation*

After the filters are overlapped in preprocessing, the weight loading, input loading, and MAC stages for convolution are the same as the baseline. The reduction stage is different from baseline and specified as follows.

Similar to reduction with coalescing, an additional preparing round is required before the normal reduction operation, due to the interleaving of MAC results computed with multiple 3D-filters. In the preparing round, two sets of fresh wordlines are allocated (the number of wordlines of each set equals the bit width of partial sum). For an array with N filters, the bitlines are partitioned into groups of $256/N$ bitlines, and the partial sum from each filter is copied to its own group of $256/N$ bitlines to perform reduction in the preparing round. Copying of partial sums is done selectively with channel masks. After the preparing round, reduction for the N filters in the same array can be done in parallel. For example, in AlexNet layer conv3, C=256. In the first 8kB array, there are 32 wordlines (WL1-WL32) storing the partial sums of the overlapped filters (M=1,2). In the preparing round for reduction, the BL1-BL128 of WL1-WL32 are selectively copied to another set of wordlines, WL33-WL64, based on channel-mask for C1-C128 of M1. Then, BL129-BL256 of WL1-WL32 are selectively copied to BL1-BL128 of WL65-WL96, based on channel-mask for C129-C256 of M1. Next, the partial sums for M2 are similarly copied to BL129-BL256 of WL33-WL96. Then, addition is done between WL33-WL64, and WL65-WL96. After the above starting round, the partial sums of filter M1 are in BL1-BL128, and those of M2 in BL129-BL256. So the remaining reduction can proceed as in original Neural Cache.

The total latency for reduction will drop thanks to the reduced number of output pixels computed in serial. The extra overhead includes the time for loading the channel masks, consisting of $C \times M$ bits per layer, as well as the extra preparing round.

2) *Weight Pruning Method*

Similar with coalescing, the filter weights are pruned at the granularity of each $R \times S$ 2D-filter.

With the overlapping scheme, it is desired that the M different filters are pruned at different channels. Such pruning generates remaining filters that can overlap with each other. To achieve this, we can explicitly control which channels are pruned by carefully designing the weight masks. All the M 3D filters are divided equally into "overlappable groups," where each group has N filters. The N filters in each group are masked such that at each channel, only weights from one filter are unpruned, while others being pruned. The remaining filter must have the highest L2-norm of its weights over other filters at the channel. For example, if N=2, then we take the 1st and 2nd filters, compare the L2-norm of the 2D-filter weights of the two filters iteratively from the 1st channel to the last (C-th) one. At the channel c, if L2-norm$(c,1)$>L2-norm$(c,2)$, then Mask$(c,1)$=1, Mask$(c,2)$=0. Otherwise, Mask$(c,1)$=0, Mask$(c,2)$=1. The masks for the 3rd and 4th, ..., (M-1)-th and M-th filters are generated with the same rule. With this pruning procedure, the pruning rate is $1-\frac{1}{N}$. It is possible to prune the filters in other patterns for overlapping, which might yield better accuracy. Such exploration is left for future work.

After the masks are generated, the network is fine-tuned layer by layer as described in Section III-A3. To adjust the target pruning rate, the parameter N can be tuned on a layer-by-layer basis.

IV. Low-Precision Architecture

Reducing the bit width of filter weights and activation maps, while maintaining a reasonable inference accuracy, is well studied in literature. Due to the bit-serial characteristic of the in-SRAM computation, it is natural to leverage the reduced bit width to accelerate computation. Specifically, using ternary and binary weights can further replace the requirement of multiplication simply with addition.

With fewer bits in the weights and activations, it is possible to fit more filter weights along a bitline and avoid unnecessary filter splitting (where one 2D filter is split and mapped to multiple bitlines), thus reducing the total number of reduction needed. Also, the maximum possible bit width for reduction result decreases, saving the cycles for copying and addition in reduction. Finally, less time and energy will be spent on data movement for loading weights and input activations - the data size to load is proportional to the bit width.

The following subsections describe the architectural support for ternary and binary networks.

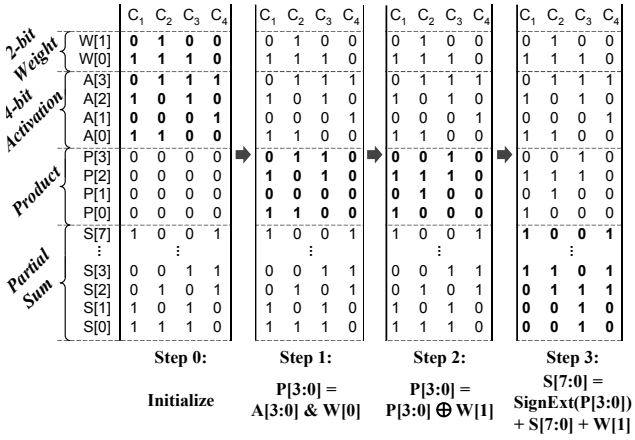

Figure 7. MAC step of Ternary Network.

A. Ternary Networks

Ternary networks have weight values from the set {0, 1, -1}. Each weight can be encoded with 2 bits – one for sign (bit 0 for positive; 1 for negative), the other for magnitude (bit 0 for zero; 1 for one). To simplify computation, we require a zero be encoded as positive zero. The multiplication of ternary weights with quantized activations can be achieved with supported arithmetic operations. First, each bit-slice of activation is AND-ed with the magnitude bit to perform the multiplication with zero-valued weights. The AND-ed results are written to a new set of wordlines for the product. Second, each bit-slice of the product is XOR-ed with the sign bit, to compute the 1's complement of the product for those channels with weight values -1. Finally, the products are added to the partial sums for accumulating the products from the same channel, after being sign-extended to match the partial sum bit width. The sign bit of weight is used as carry in the addition, and therefore the 1's complement generated in the previous step will become 2's complement here. Collocating the conversion to 2's complement with the partial sum accumulation saves one addition operation.

Figure 7 demonstrates an example of 4-bit activations convolving with ternary weights. The figure shows one multiplication and addition in the MAC step, where 4 channels are computed in parallel. W[1] is the sign bit and W[0] is the magnitude bit. In Step 1, A[3:0] is bitwise AND-ed with W[0] and the results are written to P[3:0]. In Step 2, P[3:0] is bit-wise XOR-ed with W[1] and the results update P[3:0]. In Step 3, P[3:0] is accumulated into partial sum S[7:0], while being sign-extended with the weight sign bit W[1] and also using W[1] as the carry. In this example, 8 cycles are spent on multiplication (Step 1-2), and another 8 cycles are spent on addition (Step 3).

Table I
Baseline CPU & GPU Configuration

CPU	Intel Xeon E5-2697 v3
Base Frequency	2.6 GHz
Cores/Threads	14/28
Process	22 nm
TDP	145 W
Cache	32 kB i-L1 per core, 32 kB d-L1 per core, 256 kB L2 per core, 35 MB shared L3
System Memory	64 GB DRAM, DDR4

GPU	Nvidia Titan Xp
Frequency	1.6 GHz
CUDA Cores	3840
Process	16 nm
TDP	250 W
Cache	3MB shared L2
Graphics Memory	12 GB DRAM, GDDR5X

B. Binary Networks

Binary networks have weight values of either 1 or -1. Each weight can be encoded with only one sign bit. MACs of weights and activations can therefore be converted to addition and subtraction of activations. For the in-cache compute, the algorithm is similar to the one for ternary networks, but without the first step of AND-ing with magnitude bit. For 4-bit activations, it takes 4 cycles to do one multiplication, and a number of partial sum bit width cycles for one accumulation.

V. Evaluation Methodology

CPU/GPU Baseline: The dual-socket Intel Xeon E5-2697 v3 serves as the CPU baseline platform; Nvidia Titan Xp as the GPU baseline. The hardware specifications are summarized in Table I. We use TensorFlow v1.9 and its profiler as the software framework and the tool for measuring the latency of CPU/GPU baseline. For CPU energy and power measurement, we use RAPL tools [19]. Nvidia-SMI is used for measuring GPU power and energy. Note that the baseline CPU chip has the exact same L3 cache structure as the model for the Neural Cache architecture.

Neural Cache Baseline: We use an in-house cycle-

accurate simulator for estimating the time and energy for Neural Cache arithmetic. For fair comparison, we use the same cycle latency and energy model as reported in the paper [6]. The in-SRAM compute frequency is at 2.5 GHz. The energy per cycle is 15.4 pJ and 8.6 pJ for compute and read/write cycles in SRAM, respectively.

The time of data loading (filter weights and input activations) depends on the latency of transferring data between DRAM and LLC, as well as on the LLC inter-slice ring. To model this, we develop a micro-benchmark in C measuring the latency of loading data from DRAM to the CPU. The micro-benchmark loads data to the exact cache sets as the data loading for in-cache computation. Note that for input loading, the data are loaded from one reserved LLC slice, so we exclude the DRAM bounded time reported by Intel VTune. We use RAPL tools [19] to measure the CPU energy of the micro-benchmark, for estimating the data loading energy.

Proposed architecture: For sparsity-aware pruning, we develop a tool-chain in TensorFlow. The cycle-accurate simulator is used to model our proposed sparsity and low-precision architecture with proper modifications. The latency, energy and area overheads of the proposed circuit are modeled by synthesis. For the Coalescing Unit, we use IBM 45 nm soi12s0 cell library and Synopsys Design Compiler for modeling the Reconfiguring Peripheral. With scaling to 22 nm, the area of each RP is 0.05 mm^2. We conservatively do not scale for the power and latency estimation. The estimated power of each RP, including leakage power and dynamic power, is 325 mW. The RP has a latency of 29.2 ns, which is equivalent to 73 cycles at 2.5 GHz. The 256×256 crossbar is modeled conservatively at 28 nm. Each crossbar is 0.032 mm^2, and it transmits 256 input bits in a latency of 163 ps and an energy of 49 pJ. In total, the 8 Coalescing Units in the reserved way translate to less than 7% area overhead to a cache slice and 1.9% area overhead to the processor die.

CNN models: We use AlexNet [20] and Inception v3 [21] as the network model for evaluating performance and accuracy. The dataset is the ImageNet 2012 1K dataset for image classification [22] (training set for training and fine-tuning, validation set for evaluating accuracy). For evaluating the performance of CPU and GPU baseline, the network weights are floating point numbers (single-precision for CPU, half-precision for GPU) because the quantized network has worse performance on CPU and GPU platforms due to lack of optimized software libraries. To fairly compare with CPU/GPU baselines, we apply state-of-art pruning techniques of Scalpel [8] - a pruning method for CPUs and GPUs. For evaluating the accuracy of Neural Cache baseline and sparse architecture, we perform 8-bit linear quantization on weights and inputs, based on the dynamic range of all the weights/inputs in one layer. Note that for the sparsity architectures, the quantization happens after pruning and fine-tuning the network. Quantization for ternary network is done with the approach described in WRPN [23], where the weights are first clipped by the range [-1, 1] and then linearly quantized. For binary network quantization, we follow the approach in DoReFa-Net [10].

VI. Results

In this section, we show the results of latency, energy, for all convolutional layers of the CNNs. We also show the inference accuracy and model size after pruning. The configurations we evaluated are: CPU and GPU baseline (CPU-base, GPU-base), Neural Cache baseline (N$-base), sparsity-aware architecture with coalescing (SPR-coal)/ overlapping (SPR-olap), and low-precision architecture with 4-bit activation, ternary (LPR-2b)/ binary (LPR-1b) weights.

A. Latency

Figure 8 shows the latency of all the convolutional layers of the AlexNet with corresponding configurations. The CPU and GPU baseline have significantly higher latencies of 6.89 ms and 1.43 ms. All the accelerator architectures benefit from in-situ computation and a large number of SIMD slots. The Neural Cache baseline is 0.619 ms. The LPR configurations achieve the best latency at the cost of accuracy loss, 0.199 ms and 0.158 ms for ternary and binary models (3.1× and 3.9× speedup over N$-base). This is because bit-serial computation scales with bit-width intrinsically. The SPR-coal and SPR-olap achieve 0.375 ms and 0.390 ms of latency.

Figure 9 compares the layer latency (break down into 5 stages) of SPR/LPR configurations with Neural Cache. For SPR configurations, we see a significant drop in weight loading time, thanks to the reduced size of total weights. The MAC time goes down because the number of serial computation required is smaller. The gain of input loading time savings is reduced by the large input size of the first layer, where pruning is not applied for maintaining accuracy. The reduction time goes down slightly as expected, where the benefits of fewer channels are offset by the overheads of the preparing round computation and mask loading.

The performance is also affected by the parameters of layers. The layers with larger filter sizes, more operations and smaller input data sizes enjoy a better speedup since the weight loading and MACs can be better accelerated. Also, a higher pruning rate leads to a higher speedup for SPR configurations.

Figure 10 further breaks down the overall latency of SPR-coal. The computation (MAC and reduction) consists about half of the latency, with 28% for MAC and 20% for reduction. The weight loading also takes up a significant portion (32%), which is in part due to the long latency of DRAM communication. The latency for all the convolutional layers and the relative speedup over CPU baseline, for Inception v3, are shown in Table IV.

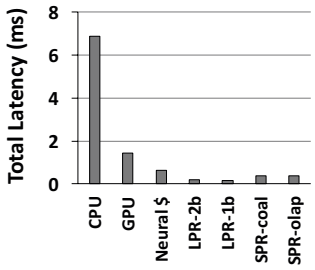

Figure 8. Total Convolution Latency Across Layers.

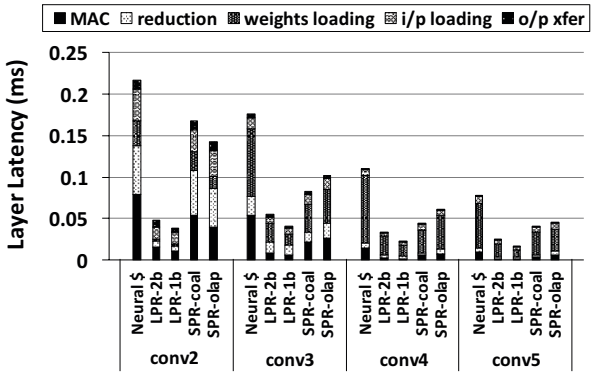

Figure 9. Latency of Each Convolutional Layer.

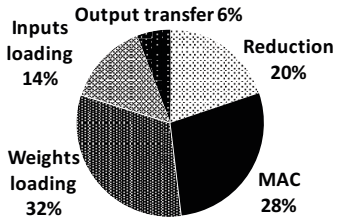

Figure 10. Latency Breakdown of SPR-Coal Configuration.

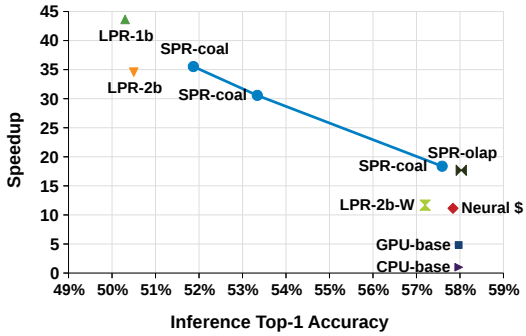

Figure 11. Accuracy vs Performance for AlexNet.

B. Accuracy vs Performance Trade-off

An overview of accuracy vs. performance for all configurations is presented in Figure 11. The top-1 accuracy on the validation set is chosen as the accuracy metric, and performance is quantified as the speedup over baseline CPU. The CPU baseline has an accuracy of 57.97% and N$-base is 57.84% with *8-bit* quantized weights and activations. The SPR configurations may achieve zero (SPR-olap, 58.03%) or less than 0.5% accuracy loss (SPR-coal, 57.59%). There are three connected data points for SPR-coal on the graph, which is achieved by varying the sparsity at structured pruning: the two points with higher sparsity have the accuracy of 53.34% and 51.87%. Note that the SPR configurations are quantized to 8-bit weights and activations as described in experiment methodology.

Overall, SPR methods can maintain minimal accuracy loss while offering up to 18.4× speedup over CPU baseline. SPR-coal also enables flexible trade-off between accuracy and performance. The point SPR-coal (51.87%, 35.5×) has better speedup than LPR-2b, also with a 1.37% higher accuracy. LPR-1b achieves the highest speedup (43.6×), at the cost of a 7.67% accuracy loss to CPU-base. If future research improves the accuracy for LPR models, the LPR data points will shift right in the figure, making them a more competitive option. The LPR-2b-W data point is experimented with the same configuration with LPR-2b except that the channel number is doubled for the all the layers to improve accuracy, as described in WRPN [23]. Although the accuracy is close to the baseline, the speedup is lower than SPR configurations because doubling channels incurs overheads to all stages. For LPR-2b-W, the widening of input channels C by 2× also requires widening of output channels M by 2×. The number of weights is proportional to $C \times M$, so it increases by 4× with widening, offsetting the 4× reduction from 8-bit weight baseline to 2-bit LPR configuration. The time for ternary weight MAC does improve but in reduction stage the total number of channels increases by 4×. Overall the speedup of LPR-2b-W is worse than SPR and 5% better than Neural Cache baseline.

The top-1 accuracy results with Inception v3 are summarized in Table IV.

C. Energy

Figure 12 plots the estimated energy consumption of convolutional layers of AlexNet for all the configurations. Baseline CPU has the highest energy at 0.512 J. N$-base consumes 0.024 J. LPR-1b consumes the least energy at 0.007 J. SPR-coal and SPR-olap have the energy consumption of 0.0151 J and 0.0150 J, respectively. The layer-by-layer energy consumption is plotted in Figure 13. Leakage indicates the background leakage energy on CPU and DRAM. In N$-base the most significant portion is MAC, and the LPR and SPR both successfully reduce MAC cycles and thus energy. LPR-1b is 3.4× more energy efficient than N$-base while SPR-olap achieves 1.59× improvement. Figure 14 shows the energy breakdown of SPR-coal, where MAC dominates energy consumption at 42%. Less than 40% of energy is spent on data movement. Table IV shows the energy consumption of all the convolutional layers for Inception v3 and the energy efficiency improvements over the CPU baseline.

The savings in energy can be attributed to the reduced amount of necessary computation and data movement, with the removal of redundant filters. The energy trend is similar to the latency, since at computation the SIMD slots are highly utilized.

We further propose the following techniques to reduce

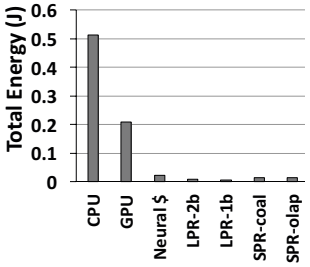

Figure 12. Total Energy for Convolution.

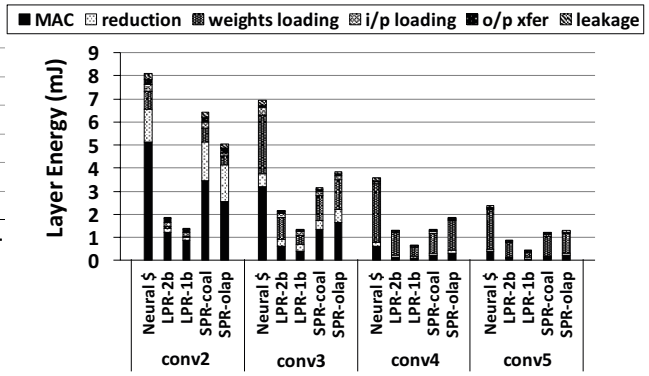

Figure 13. Energy of Each Convolutional Layer.

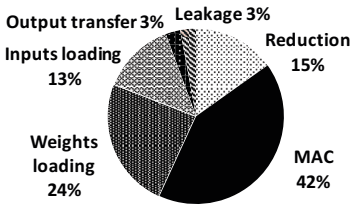

Figure 14. Energy Breakdown of SPR-Coal Configuration.

Table II
PRUNING RATE OF SPR-COAL AND SPR-OLAP

Layer	conv2	conv3	conv4	conv5	Overall
SPR-coal	27%	60%	55%	42%	50%
SRP-olap	50%	50%	50%	50%	49%

Table III
MODEL SIZE COMPARISON

Neural $	LPR-2b	LPR-1b	SPR-coal	SPR-olap
2285 kB	602 kB	321 kB	1147 kB	1163 kB

energy. First, based on the observation that most of the unpruned 8-bit quantized weights are centered around the value 128, we can use fewer bits to encode the weight values that are close to 128. In MAC, for those encoded weights, it takes fewer cycles since the bit width is reduced (for an n-bit weight, the multiplication needs 11n-2 cycles). Then, the input activations corresponding to encoded weights are added up and then shifted to correct the offset in weight encoding. Second, during reduction, the column peripherals of the bitlines that do not have active partial sum data can be turned off. For example, in the addition phase of the first reduction round, half of the total bitlines do not need activating. Third, we can power-gate the unused CPU cores to save static energy. It is sufficient to leave one core running to control the Neural Cache operations, while other cores are power-gated.

With zeros in the activations, there can be additional savings by selectively disabling the bitlines at the write-back cycle of the in-SRAM compute. The zeros in the activations can be detected by sensing the shared bitline: for an 8-bit activation, for instance, it is cheap to sense in analog whether the AND of the 8 bitlines is zero. If so, then this activation must be zero and a mask bit is set to skip the computation on this bitline at the MAC stage. This can lead to an additional 8% of the total energy savings on average, which are not included in the reported results.

D. Model Size

In Table III, the total sizes of the pruned/quantized models for AlexNet are compared. For low-precision models, the model size decreases linearly with the bit width. For pruned models, the model size is determined by the pruning rate (summarized in Table II), with a small overhead of mask vectors. The SPR-coal and SPR-olap achieve 49.8% and 49.1% pruning rate, respectively. The pruning rate falls in the medium level, and is exploited by our architecture with high efficiency. The model size of LPR-2b is reduced by 4× and LRP-1b by 8×, which is the direct result of using low numerical precision of weights. For Inception v3, in both SPR-coal and SPR-olap, 50% of weights are pruned for the layers we prune.

Table IV
EFFICIENCY OF INCEPTION V3 MODEL

Architecture	latency (ms)	energy (J)	top-1 accuracy	speedup	relative energy efficiency
CPU	56.8	6.14	78.5%	1.0	1.0
GPU	19.9	2.73	78.5%	2.9	2.2
Neural $	4.66	0.18	78.3%	12.2	35.2
SPR-coal	3.43	0.12	76.7%	16.6	50.1
SRP-olap	3.64	0.13	77.2%	15.6	45.8
LPR-1b	1.00	0.03	61.0%	56.7	183.6

Table V
COMPARISON WITH ASICS. AREA SCALED TO 22 NM.

Architecture	Area (mm^2)	Latency (ms)	Inferences /($s \cdot mm^2$)	Inferences /($s \cdot mm^2 \cdot J$) - with DRAM energy
SPR-coal	9.8	0.38	272	17,977
SPR-coal (w/o RP)	4.2	0.47	511	29,242
Eyeriss [24]	1.4	115.3	24	1,511
Cambricon-X [25]	0.7	4.97	275	23,226
SCNN [17]	14.9	0.76	88	N/A

E. Comparing to ASICs

To demonstrate the efficiency of this work over other accelerators, in Table V, we perform a comparison with recent works on sparsity-aware DNN accelerators. Eyeriss [24] is a low-power DNN accelerator. Cambricon-X [25] optimizes for sparse weights. SCNN [17] optimizes for both sparse weights and activations. We use the SPR-coal configuration in comparison. The area overhead includes the modifications required by Neural Cache and the additions for this work. To optimize for area efficiency, we propose another design, where the crossbar values of the Coalescing Unit are directly loaded from DRAM instead of being generated from the Reconfiguring Peripheral. This design incurs a higher latency, as shown in the table. We do not synthesize or layout

other ASICs. Instead, the area overheads are scaled to 22 nm process node, according to the relationship between area and technology node for recent processors [26]. The energy and frequency are as reported in the original papers and not scaled. We compare the throughput under the unit area and energy. The results are shown include energy consumed by DRAM. The latencies of Cambricon-X and SCNN are not directly provided, so they are estimated based on the cycle count and frequency. The accelerator energy is estimated based on the reported average power and the estimated latency. The energy of SCNN is not estimated because the average power for SCNN was not reported in the paper. The DRAM energy of Eyeriss and Cambricon-X is estimated based on the DDR4 power model of Micron [27].

Compared to the other ASICs, our design has the shortest latency thanks to the high parallelism. The area-optimized design has a small increase in latency because the coalescing configuration is read from DRAM, but it still outperforms other accelerators in terms of throughput per area. Comparing the throughput per area per energy (not including DRAM), our design is within 1.8× of Cambricon-X. However, when DRAM energy is included, our design is better in this metric than Cambricon-X. In contrast to the in-memory architecture of this work, Cambricon-X has a small on-chip buffer, and hence requires a lot of off-chip data transfer between DRAM and the accelerator chip. Please note that our architecture achieves the stated improvements while being a *general purpose processor*. Further, the in-cache compute capabilities can be leveraged to accelerate several other application domains, including CNNs.

VII. Related Work

To the best of our knowledge, this is the first work that leverages sparsity and low-precision for in-SRAM CNN inference acceleration. We describe the related work below.

In-situ SRAM Computation: A few previous works demonstrate the feasibility of in-situ computation in SRAM arrays [12], [13]. Compute Caches [12] performs logical operations and binary matrix multiplication. Cache Automaton [28] targets non-deterministic finite automata processing. Neural Cache [6] accelerates CNN and is the compared baseline of this paper. There are ASIC [2], [3], [24] and FPGA [29] accelerators built for CNN acceleration. While the ASIC solutions achieve high efficiency, they incur extra design cost. ASICs lack flexibility in that they cannot be repurposed for other domains. In contrast, cache improves the performance of many other workloads when not functioning as a CNN accelerator. FPGA solutions provide flexibility but require additional hardware.

Sparsity-aware NN accelerators: EIE [30] is an ASIC for DNN inference which leverages sparsity in weights and activations. It uses a compressed model to fit the network in the on-chip SRAM. However, it focuses on the fully connected layers while our work accelerates convolutional layers, which take up the majority of CNN inference time. SCNN [17] proposes an accelerator for sparse CNNs with a dataflow that multiplies the non-zero weights and activations of the same channel while keeping them in a compressed format. Eyeriss [1] uses clock-gating to skip unnecessary MAC operations and save energy; it does not improve convolution latency. Cnvlutin [31] detects zeros in activations with hardware and skips ineffectual computation. Stripes [32] and Pragmatic [33] apply variations of bit-serial computation and save computation time proportional to bit-width, where the necessary bit-width is determined by the network redundancy in each layer. Cambricon-X [25] is an accelerator with an indexing module for efficiently selecting the non-sparse weights and activations. The above works build customized ASIC, while this paper is based on the cache structure of commodity general purpose processors.

VIII. Conclusion

Our proposed in-cache architecture improves latency and energy efficiency by skipping sparse weights with two schemes – coalescing and overlapping. In coalescing, the effective input channels are coalesced from the original activation map efficiently by the Coalescing Unit. Therefore the vector slots will not be wasted on ineffective computation. In overlapping, the filters are overlapped with each other where the non-zero channels do not conflict. Hence, the utilization of vector units is increased due to more filters being convolved with inputs at the same time. We also develop in-cache compute techniques for low-precision CNNs with binary and ternary weights.

Acknowledgment

We thank the members of M-bits research group and the anonymous reviewers for their suggestions, which helped to improve the paper. This work was supported in part by the NSF CAREER-1652294 award, NSF-1763918 award and Intel gift award.

References

[1] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Computer Architecture (ISCA), ACM/IEEE 43rd International Symposium on*. IEEE, 2016, pp. 367–379.

[2] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[3] N. P. Jouppi, C. Young, N. Patil, D. Patterson *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 1–12.

[4] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the*

43rd International Symposium on Computer Architecture. IEEE Press, 2016, pp. 14–26.

[5] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, pp. 27–39.

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

[7] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[8] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 2017, pp. 548–560.

[9] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 525–542.

[10] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[11] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *arXiv preprint arXiv:1612.01064*, 2016.

[12] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 481–492.

[13] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw, "A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 4, pp. 1009–1021, 2016.

[14] M. Huang, M. Mehalel, R. Arvapalli, and S. He, "An energy efficient 32-nm 20-mb shared on-die L3 cache for intel® xeon® processor E5 family," *J. Solid-State Circuits*, vol. 48, no. 8, pp. 1954–1962, 2013.

[15] W. Chen, S.-L. Chen, S. Chiu, R. Ganesan, V. Lukka, W. W. Mar, and S. Rusu, "A 22nm 2.5 mb slice on-die l3 cache for the next generation xeon® processor," in *VLSI Technology (VLSIT), 2013 Symposium on*. IEEE, 2013, pp. C132–C133.

[16] W. J. Bowhill, B. A. Stackhouse, N. Nassif, Z. Yang, A. Raghavan, O. Mendoza, C. Morganti *et al.*, "The xeon® processor E5-2600 v3: a 22 nm 18-core product family," *J. Solid-State Circuits*, vol. 51, no. 1, pp. 92–104, 2016.

[17] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of 44th International Symposium on Computer Architecture*. ACM, 2017, pp. 27–40.

[18] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in neural information processing systems*, 2016, pp. 2074–2082.

[19] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: memory power estimation and capping," in *Low-Power Electronics and Design, 2010 ACM/IEEE International Symposium on*. IEEE, 2010, pp. 189–194.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[21] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2818–2826.

[22] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[23] A. Mishra, E. Nurvitadhi, J. J. Cook, and D. Marr, "WRPN: Wide reduced-precision networks," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=B1ZvaaeAZ

[24] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[25] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *Microarchitecture (MICRO), 49th International Symposium on*. IEEE, 2016, pp. 1–12.

[26] W. Huang, K. Rajamani, M. R. Stan, and K. Skadron, "Scaling with design constraints: Predicting the future of big chips," *IEEE Micro*, vol. 31, no. 4, pp. 16–29, 2011.

[27] Micron Technology Inc., "Calculating memory power for ddr4 sdram," Tech. Rep. TN-40-07, 2017.

[28] A. Subramaniyan, J. Wang, E. R. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, "Cache automaton," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 259–272.

[29] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 1–14.

[30] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, pp. 243–254.

[31] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 1–13.

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

[33] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 382–394.