SPRING: A Sparsity-Aware Reduced-Precision Monolithic 3D CNN Accelerator Architecture for Training and Inference

Ye Yu, and Niraj K. Jha, Fellow, IEEE

Abstract—Convolutional neural networks (CNNs) outperform traditional machine learning algorithms across a wide range of applications, such as object recognition, image segmentation, and autonomous driving. However, their ever-growing computational complexity makes it necessary to design efficient hardware accelerators. Most CNN accelerators focus on exploring various dataflow styles and designs that exploit computational parallelism. However, potential performance improvement from sparsity (in activations and weights) has not been adequately addressed. The computation and memory footprint of CNNs can be significantly reduced if sparsity is exploited in network evaluations. Therefore, different pruning methods have been proposed to increase sparsity. To take advantage of sparsity, some accelerator designs explore sparsity encoding and evaluation on CNN accelerators. However, sparsity encoding is just performed on activation data or CNN weights and only used in inference. It has been shown that activations and weights also have high sparsity levels during the network training phase. Hence, sparsity-aware computation should also be considered in the training phase. To further improve performance and energy efficiency, some accelerators evaluate CNNs with limited precision. However, this is limited to the inference phase since reduced precision sacrifices network accuracy if used in training. In addition, CNN evaluation is usually memory-intensive, especially during training. The performance bottleneck arises from the fact that the memory cannot feed the computational units enough data, resulting in idling of these computational units and thus low utilization ratios. A 3D memory interface has been used on high-end GPUs to alleviate memory bandwidth shortage. In this article, we propose SPRING, a SParsity-aware Reduced-precision Monolithic 3D CNN accelerator for training and inference. SPRING supports both CNN training and inference. It uses a binary mask scheme to encode sparsities in activations and weights. It uses the stochastic rounding algorithm to train CNNs with reduced precision without accuracy loss. To alleviate the memory bottleneck in CNN evaluation, especially during training, SPRING uses an efficient monolithic 3D nonvolatile memory interface to increase memory bandwidth. Compared to Nvidia GeForce GTX 1080 Ti, SPRING achieves 15.6×, 4.2×, and 66.0× improvements in performance, power reduction, and energy efficiency, respectively, for CNN training, and 15.5×, 4.5×, and 69.1× improvements in performance, power reduction, and energy efficiency, respectively, for inference.

Index Terms—Convolutional neural network, deep learning, hardware accelerator, inference, reduced precision, sparsity, stochastic rounding, training.

1 INTRODUCTION

Convolutional neural networks (CNNs) excel at various important applications, e.g., image classification, image segmentation, robotics control, and natural language processing. However, their high computational complexity necessitates specially-designed accelerators for efficient processing. Training of CNNs requires an enormous amount of computing power to automatically learn the weights based on a large training dataset. Few ASIC-based CNN training accelerators have been presented [1], [2], [3]. However, graphical processing units (GPUs) typically play a dominant role in the training phase when CNN computation essentially maps well to the single-instruction multiple-data (SIMD) units and the large number of SIMD units present in GPUs provide significant computational throughput for training CNNs [4], [5]. In addition, the higher clock speed, bandwidth, and power management capabilities of the Graphics Double Data Rate (GDDR) memory relative to the regular DDR memory make GPUs the de facto accelerator choice for CNN training. On the other hand, CNN inference is more latency- and power-sensitive as an increasing number of applications need real-time CNN evaluations on battery-constrained edge devices. Hence, ASIC- and FPGA-based accelerators have been widely explored for this purpose. However, they can only process low-level CNN operations, such as convolution and matrix multiplication, and lack the flexibility of a general-purpose processor. Although CNN models have evolved rapidly recently, their fundamental building blocks are common and long-lasting. Therefore, the ASIC- and FPGA-based accelerators can efficiently process new CNN models with their domain-specific architectures. FPGA-based accelerators achieve faster time-to-market and enable prototyping of new accelerator designs. Microsoft has used customized FPGA boards, called Catapult [6], in its data centers to accelerate Bing ranking by 2×. An FPGA-based CNN accelerator that uses on-chip memory has been proposed in [7], where a fixed-point representation is used to keep all the weights stored in on-chip memory thus avoiding the need to access external memory. To improve dynamic resource utilization of FPGA-based CNN accelerators, multiple accelerators, each specialized for a specific CNN layer, have been constructed using the same FPGA
2 BACKGROUND

In this section, we discuss the background material necessary for understanding our proposed sparsity-aware reduced-precision accelerator architecture. We first give a primer on CNNs. We then discuss existing sparsity-aware designs. Then, we discuss various CNN training algorithms that use low numerical precision. Finally, we describe an efficient on-chip memory interface that is used for CNN acceleration.

2.1 CNN overview

Although different CNNs have different hyperparameters, such as the number of layers and shapes, they share a
similar architecture, as shown in Fig. 1, CNNs are generally composed of five building blocks: CONV layers, activation (ACT) layers, pooling (POOL) layers, batch normalization layers (not shown in Fig. 1), and FC layers. Among these basic components, the CONV and FC layers are the most compute-intensive. We describe them next.

**CONV layers:** a batch of 3D input feature maps are convolved with a set of 3D filter weights to generate a batch of 3D output feature maps. The filter weights are usually fetched from external memory once and stored in on-chip memory as they are shared among multiple convolution windows. Therefore, CONV layers have relatively low memory bandwidth pressure and are usually compute-bound as they require a large number of convolution computations. Given the input feature map $I$ and filter weights $W$, the output feature map $O$ is computed as follows:

$$O[n][k][p][q] = \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} I[n][c][p+r][q+s] \times W[k][c][r][s]$$  \hspace{1cm} (1)

where $I \in \mathbb{R}^{NCHW}$, $W \in \mathbb{R}^{KCRS}$, and $O \in \mathbb{R}^{NKPS}$. $N$ is the number of images in a batch and $K$ is the total number of filters in the CONV layer. $C$ represents the number of channels in the input feature maps and filter weights. $H$ and $W$ denote the height and width of the input feature maps, respectively, whereas $R$ and $S$ denote the height and width of filter weights, respectively. The vertical and horizontal strides are given by $u$ and $v$, respectively. The height and width of the output feature maps are given by $P$ and $Q$, respectively.

**FC layers:** the neurons in an FC layer are fully connected with neurons in the previous layer with a specific weight associated with each connection. It is the most memory-intensive layer in CNNs [9], [32] since no weight is reused. The computation of the FC layer can be represented by a matrix-vector multiplication as follows:

$$y = Wx + b$$  \hspace{1cm} (2)

where $W \in \mathbb{R}^{m \times n}$, $y$, $b \in \mathbb{R}^{m}$, and $x \in \mathbb{R}^{n}$. The output and input neurons of the FC layer are represented in vector form as $y$ and $x$. $W$ represents the weight matrix and $b$ is the bias vector associated with the output neurons.

### 2.2 Exploiting sparsity in CNN accelerators

It is known that the sparsity levels of CNN weights typically range from 20% to 80% [33], [34], and when the rectified linear unit (ReLU) activation function is employed, the activations are clamped to zeros in the 50% to 70% range [23]. The combination of weight and activation sparsities can reduce computations and memory accesses significantly if the accelerator can support sparsity-aware operations. In order to speed up CNN evaluation by utilizing weight/activation sparsity, the first step is to encode the sparse data in a compressed format that can be efficiently processed by accelerators. EIE is an accelerator that encodes a sparse weight matrix in a compressed sparse column (CSC) format [35] and uses a second vector to encode the number of zeros between adjacent non-zero elements [20]. However, it is only used to speed up the FC layers and has no impact on the CONV layers. Hence, a majority of CNN computations do not benefit from sparsity-aware acceleration. A lightweight run-time output sparsity predictor has been developed in SparseNN, an architecture enhanced from EIE, to accelerate CNN inference [21]. Activations in the CSC format are first fed to the lightweight predictor to predict the non-zero elements in the output neurons. Then, the activations associated with non-zero outputs are sent to feedforward computations to bypass computations that lead to zero outputs. If the number of computations skipped is large enough, the overhead of output predictions can be offset. However, since the output sparsity predicted by the lightweight predictor is an approximation of the real sparsity value, it incurs an accuracy loss that makes it unsuitable for CNN training. SCNN is another accelerator that uses a zero-step format to encode weight/activation sparsity: an index vector is used to indicate the number of non-zero data points and the number of zeros before each non-zero data point. It multiplies activation and weight vectors in a manner similar to a Cartesian product using an input stationary dataflow [22]. However, the Cartesian product does not automatically align non-zero weights and activations in the FC layers since the FC layer weights are not reused as in the case of CONV layers. This leads to performance degradation for FC layers and makes SCNN unattractive for CNNs dominated by FC layers. Cnvlutin [24] enhances the DaDianNao architecture to support zero-skipping in activations using a zero-step offset vector that is similar to graphics processor proposals [36], [37], [38], [39]. The limitation of this architecture is that the length of offset vectors in different PEs may be different. Hence, they may require different numbers of cycles to process the data. Thus, the PE with the longest offset vector becomes the performance bottleneck while other PEs idle and wait for it. Cambricon-X is an accelerator that also employs the zero-step sparsity encoding method and uses a dedicated indexing module to select and transfer needed neurons to PEs, with a reduced memory bandwidth requirement [25]. The PEs run asynchronously to avoid the idling problem of Cnvlutin. UCNN is an accelerator that improves CNN inference performance by exploiting weight repetition in the CONV layers [23]. It uses a factorized dot product dataflow to reduce the number of multiplications and a memorization method to reduce weight memory access via weight repetition.

Both the CSC and zero-step encoding formats compress data by eliminating zero-elements and the accelerators discussed above efficiently process the compressed data. How-
ever, weight/activation sparsity can not only be exploited at the PE level but also at the bit level. Stripes, a bit-serial hardware accelerator, avoids the processing of zero prefix and suffix bits through serial-parallel multiplications on CNNs [40]. Each bit of a neuron is processed at every cycle and zero bits are skipped on the fly. Multiple neurons are processed in parallel to mitigate performance loss from bit-serial processing. Pragmatic, a CNN accelerator enhanced from Stripes, supports zero-bit skipping regardless of its position [41]. However, it needs to convert the input neuron representation into a zero-bit-only format on the fly, which leads to up to a 16-cycle latency.

2.3 Low-precision CNN training algorithms

The rapid evolution of CNNs in recent years has necessitated the deployment of large-scale distributed training using high-performance computing infrastructure [42, 43]. Even with such a powerful computing infrastructure, training a CNN to convergence usually takes several days, sometimes even a few weeks. Hence, to speed up the CNN training process, various training algorithms with low-precision computations have been proposed.

Single-precision floating-point (FP32) operation has mainly been used as the training standard on GPUs. Meanwhile, efforts have been made to train CNNs with half-precision floating-point (FP16) arithmetic since it can improve training throughput by 2×, in theory, on the same computing infrastructure. However, compared to FP32, FP16 involves rounding off gradient values and quantizing to a lower-precision representation. This introduces noise in the training process and defers CNN convergence. To maintain a balance between the convergence rate and training throughput, mixed-precision training algorithms that use a combination of FP32 and FP16 have been proposed [45]. [46]. The FP16 representation is used in the most compute-intensive multiplications and the results are accumulated into FP32 outputs. Dynamic scaling is required to prevent the vanishing gradient problem [47].

Compared to floating-point arithmetic, fixed-point operations are much faster and more energy-efficient on hardware accelerators, but have a lower dynamic range. To overcome the dynamic range limitation, the dynamic fixed-point format [48] is used in CNN training [49]. [50]. Unlike the regular fixed-point format, the dynamic fixed-point format uses multiple scaling factors that are updated during training to adjust the dynamic range of different groups of variables. The CNN training convergence rate is highly sensitive to the rounding scheme used in fixed-point arithmetic [30]. Instead of tuning the dynamic range used in the dynamic fixed-point format, a stochastic rounding method has been proposed to leverage the noise tolerance of CNN algorithms [30]. CNNs are trained in a manner that the rounding error is exposed to the network and weights are updated accordingly to mitigate this error, without impacting the convergence rate.

2.4 Efficient on-chip memory interface and emerging NVRAM technologies

CNN training involves feeding vast input feature maps and filter weights to the accelerator computing units to compute the error gradients used to update CNN weights in backpropagation. Besides the large memory size required to store all the CNN weights, a high memory bandwidth becomes indispensable to keep running the computing units at full throughput. Hence, through-silicon via (TSV)-based 3D memory interfaces have been used on high-end GPUs [5] and specialized CNN accelerators [3]. The most widely-used TSV-based 3D memory interface is HBM. In each HBM package, multiple DRAM dies and one memory controller die are first fabricated and tested individually. Then, these dies are aligned, thinned, and bonded using TSVs. The HBM package is connected to the processor using an interposer in a 2.5D manner. This shortens the interconnects within the memory system and between the memory and processor, thus reducing memory access latency and improving memory bandwidth. In addition, since more DRAM dies are integrated within the same footprint area, HBM enables smaller form factors: HBM-2 uses 94% less space relative to GDDR5 for a 1GB memory [51].

Apart from improving the DRAM interface, the industry has also been exploring various NVRAM technologies to replace DRAM, such as ferroelectric RAM (FeRAM), spin-transfer torque magnetic RAM (STT-MRAM), phase-change memory (PCM), nanotube RAM (NRAM), and resistive RAM (RRAM). It has been shown in [52], [53] that RRAM can be used in an efficient 3D memory interface to deliver high memory bandwidth and energy efficiency. Information is represented by different resistance levels in an RRAM cell. Compared to a DRAM, an RRAM cell needs a higher current to change its resistance level. Therefore, the access transistors of an RRAM are larger than those of a DRAM [54]. However, DRAM is expected to reach the scaling limit at 16nm [55] whereas RRAM is believed to be suitable for sub-10nm nodes [56]. Hence, the smaller technology node of an RRAM should offset the access transistor overhead. Besides, the nonvolatility of RRAM eliminates the need for a periodic refresh that a DRAM requires. This not only saves energy and reduces latency, but also gets rid of the refresh circuitry used in DRAM.

3 Sparsity-aware reduced-precision accelerator architecture

In this section, we present the proposed architecture, SPRING: a sparsity-aware reduced-precision CNN accelerator for both training and inference. We first discuss accelerator architecture design and then dive into sparsity-aware acceleration, reduced-precision processing, and the monolithic 3D NVRAM interface.

Fig. 2 shows the high-level view of the architecture. SPRING uses monolithic 3D integration to connect the accelerator tier with an RRAM interface. Unlike TSV-based 3D integration, monolithic 3D integration only has one substrate wafer, where devices are fabricated tier over tier. Hence, the alignment, thinning, and bonding steps of TSV-based 3D integration can be eliminated. In addition, tiers are connected through monolithic inter-tier vias (MIVs), whose diameter is the same as that of local vias and one-to-two orders of magnitude smaller than that of TSVs. This enables a much higher MIV density (10^3/mm^2 at 14nm [57]), thus leaving much more space for logic. The accelerator tier is put
Memory controllers
Accelerator tier
MIVs
RRAM tiers

Fig. 2. The SPRING architecture

at the bottom, on top of which is the memory controller tier. Above the memory controller tier lie the multiple RRAM tiers.

Fig. 3 shows the organization of the accelerator tier. The control block handles the CNN configuration sent from the CPU. It fetches the instruction stream and controls the rest of the accelerator to perform acceleration. The activations and filter weights are brought on-chip from the RRAM system by a direct memory access (DMA) controller. Activations and weights are stored in the activation buffer and weight buffer, respectively, in a compressed format. Data compression relies on binary masks that are stored in a dedicated mask buffer. The compression scheme is discussed in Section 3.1. The compressed data and the associated masks are used in the PEs for CNN evaluation. The PEs are designed to operate in parallel to maximize overall throughput.

Fig. 4 shows the main components of a PE. The compressed data are buffered by the activation FIFO and weight FIFO. Then, they enter the pre-compute sparsity module along with the binary masks. Multiple multiplier-accumulator (MAC) lanes are used to compute convolutions or matrix-vector multiplications using zero-free activations and weights after they are preprocessed by the pre-compute sparsity module. The output results go through a post-compute sparsity module to maintain the zero-free format. Batch normalization operations [58] are used in modern CNNs to reduce the covariance shift. They are executed in the batch normalization module that supports both the forward pass and backward pass of batch normalization. Three pooling methods are supported by the pooling module: max pooling, min pooling, and mean pooling. The reshape module deals with matrix transpose and data reshaping. Element-wise arithmetic, such as element-wise add and subtract, is handled by the scalar module. Lastly, a dedicated loss module is used to process various loss functions, such as L1 loss, L2 loss, softmax, etc.

3.1 Sparsity-aware acceleration

Traditional accelerator designs can only process dense data and do not support sparse-encoded computation. They treat zero elements in the same manner as regular data and thus perform operations that have no impact on the CNN evaluation results. In this context, weight/activation sparsity cannot be used to speed up computation and reduce the memory footprint. In order to utilize sparsity to skip ineffectual activations and weights, and reduce the memory footprint, SPRING uses a binary-mask scheme to encode the sparse data and performs computations directly in the encoded format.

Compared to the regular dense format, SPRING compresses data vectors by removing all the zero-elements. In order to retain the shape of the uncompressed data, an extra binary mask is used. The binary mask has the same shape as that of the uncompressed data where each binary bit in the mask is associated with one element in the original data vector. Fig. 5 shows an example of the binary-mask scheme that SPRING uses to compress activations and weights. The original uncompressed data vector has 16 elements, and if each element is represented using 16 bits, the total data...
We implement the binary mask scheme using a low overhead pre-compute sparsity module that preprocesses the sparse-encoded activations and weights and provides zero-free data to the MAC lanes. After output data traverse the MAC lanes, another post-compute sparsity module is used to remove all the zero-elements generated by the activation function before storing them back to on-chip memory. Fig. 6 shows the pre-compute sparsity module that takes the zero-free data vectors and binary mask vectors as inputs, and generates an output mask as well as zero-free activations/weights for the MAC lanes. The output binary mask indicates the common indexes of non-zero elements in both the activation and weight vectors. After being preprocessed by the pre-compute sparsity module, the “dangling” non-zero elements in the activation and weight data vectors are removed. The dangling non-zero activations refer to the non-zero elements in the activation data vector where their corresponding weights at the same index are zeros, and vice versa.

Fig. 5(a) shows the mask generation process used by the pre-compute sparsity module. The output mask is the AND of the activation and weight masks. The output mask, together with the activation and weight masks, is used by two more XOR gates for filter mask generation. Fig. 5(b) shows the XOR gates data filtering process using the three masks obtained in the previous step. The sequential scanning and filtering mechanism for one type of data used in the filtering step is shown in Algorithm 1. The data vector, as well as the two mask vectors, is scanned in sequence. At each step, a 1 in the output mask implies a common non-zero index. Hence, the corresponding element in the data vector passes through the filter. On the other hand, if a 0 appears in the mask filter and the corresponding mask bit in the filter mask is 1, then a dangling non-zero element is detected in the data vector and is blocked by the filter. If both the output mask bit and filter mask bit are zeros, it means that the data elements at this index in both the activation and weight vectors are zeros and thus already skipped. After filtering out the dangling elements in activations and weights, a zero-collapsing shifter is used to remove the zeros and keep the data vectors zero-free in a similar sequential scanning manner, as shown in Fig. 5(c). These zero-free activations and weights are then fed to the MAC lanes for computation. Since only zero-free data are used in the MAC lanes, ineffectual computations are completely skipped, thus improving throughput and saving energy.

Algorithm 1 Sequential scanning and filtering mechanism

1: **Inputs:** in_data, output_mask, filter_mask
2: **output:** out_data
3: data_pointer ← 0, mask_pointer ← 0
4: while mask_pointer < mask_length do
5: if output_mask[mask_pointer] == 1 then
6:   out_data[data_pointer] = in_data[data_pointer]
7:   data_pointer++
8: else if filter_mask[mask_pointer] == 1 then
9:   out_data[data_pointer] = 0
10: data_pointer++
11: mask_pointer++

3.2 Reduced-precision processing using stochastic rounding

SPRING processes CNNs using fixed-point numbers with reduced precision. Every time a new result is generated by the CNN, it has to be first rounded to the nearest discrete number, either in a floating-point representation or a fixed-point representation. Since the gap between adjacent numbers in the fixed-point representation is much larger than in the floating-point representation, the resulting quantization error in the former is much more pronounced. This prevents the fixed-point representation from being used in error-sensitive CNN training. In order to utilize the faster and more energy-efficient fixed-point arithmetic units, we adopt the stochastic rounding method proposed in [30]. The traditional deterministic rounding scheme always rounds a real number to its nearest discrete number, as shown in Eq. 9. We follow the definitions used in [30], where \( \epsilon \) denotes the smallest positive discrete number supported in the fixed-point format and \( \lfloor x \rfloor \) is defined as the largest integer multiple of \( \epsilon \) less than or equal to \( x \).
The zero-free activations and weights from the pre-compute sparsity module are subject to multiplications in the MAC lanes, where the products are represented with $2 \times \text{IL}$ integer bits and $2 \times \text{FL}$ fractional bits to prevent overflow. Accumulations over products are also performed using $2 \times (\text{IL} + \text{FL})$ bits. Then, a stochastic rounding module is used to reduce the numerical precision before applying the activation function or storing the result back to on-chip memory. We use a linear-feedback shift register to generate pseudo-random numbers for stochastic rounding.

### 3.3 Monolithic 3D NVRAM Interface

SPRING uses a monolithic 3D NVRAM interface previously proposed in [52] and adapts it to its 3D architecture to provide the accelerator tier with significant memory bandwidth. As shown in Fig. 2, SPRING uses two memory channels where each channel has its own memory controller to control the associated two RRAM ranks. An ultra-wide memory bus (1KB wide) is used in each channel, since the interconnects between SPRING and memory controllers, and between memory controllers and RRAM ranks, are implemented using vertical MIVs. This on-chip memory bus not only reduces the access latency relative to the conventional off-chip memory bus, but also makes row-wide granular memory accesses possible to enable energy savings. In addition, the column decoder can be removed to reduce the access latency and power dissipation in this row-wide access granularity scheme. To reduce repeated accesses to the same row, especially the energy-consuming write accesses of RRAM, the row buffer is reused as the write buffer. A dirty bit is used to indicate if the corresponding row entry in the row buffer needs to be written back to the RRAM array when flushed out. The read and write accesses are decoupled by adding another set of vertical interconnects, as shown in Fig. 9 [53]. Hence, the slower write access does not block the faster read access and thus a higher memory bandwidth is achieved. In addition, RRAM nonvolatility not only enables the elimination of bulky periodic refresh circuitry, but also allows the RRAM arrays to be powered down in the idle intervals to reduce leakage power. A rank-level adaptive power-down policy
is used to maintain a balance between performance and energy saving: the power-down threshold for each RRAM rank is adapted to its idling pattern so that a rank is only powered down if it is expected to be idle for a long time.

## 4 SIMULATION METHODOLOGY

In this section, we present the simulation flow for SPRING and the experimental setup.

Fig. 10 shows the simulation flow used to evaluate the proposed SPRING accelerator architecture. We implement components of SPRING at the register-transfer level (RTL) with SystemVerilog to estimate delay, power, and area. The RTL design is synthesized by Design Compiler [59] using a 14nm FinFET technology library [60]. Floorplanning is done by Capo [61], an open-source floorplacer. On-chip buffers are modeled using FinCACTI [62], a cache modeling tool enhanced from CACTI [63], to support deeply-scaled FinFETs at the 14nm technology node. The monolithic 3D RRAM system is modeled by NVSim [64], a circuit-level memory simulator for emerging NVRAMs, and NVMain [65], an emerging NVRAM architecture simulator. The synthesized results, together with buffer and RRAM estimations, are then plugged into a customized cycle-accurate Python simulator. This accelerator simulator takes CNNs in the TensorFlow [66] Protocol Buffers format and estimates the computation latency, power dissipation, energy consumption, and area. SPRING treats the TensorFlow operations like complex instruction set computer instructions where each operation involves many low-level operations.

We compare our design with the Nvidia GeForce GTX 1080 Ti GPU, which uses the Pascal microarchitecture [4] in a 16nm technology node. The die size of GTX 1080 Ti is $471 \text{mm}^2$ and the base operating frequency is $1.48 \text{GHz}$, which can be boosted to $1.58 \text{GHz}$. GTX 1080 Ti uses an 11 GB GDDR5X memory with 484 GB/s memory bandwidth to provide 10.16 TFLOPS peak single precision performance.

We evaluate SPRING and GTX 1080 Ti on seven well-known CNNs: Inception-Resnet V2 [67], Inception V3 [68], MobileNet V2 [69], NASNet-mobile [70], PNASNet-mobile [71], ResNet-152 V2 [72], and VGG-19 [73]. We evaluate both the training and inference phases of these CNNs on the ImageNet dataset [74]. We use the default batch sizes defined in the TensorFlow-Slim library [75]: 32 for training and 100 for inference.

## 5 EXPERIMENTAL RESULTS

In this section, we present experimental results for SPRING and compare them with those for GTX 1080 Ti.

Table 1 shows the values of various design parameters used in SPRING. They are obtained through the accelerator design space exploration methodology proposed in [76]. It is shown in [30] that with 16 FL bits, training CNNs using the stochastic rounding scheme can converge in a similar amount of time with a negligible accuracy loss relative to when single-precision floating-point arithmetic is used. Hence, we use 4 IL bits and 16 FL bits in the fixed-point representation. The convolution loop order refers to the execution order of the multiple for-loops in the CONV layer.
SPRING executes convolutions by first unrolling the for-loops across multiple inputs in the batch. Then, it unrolls the for-loops within the filter weights, followed by unrolling in the activation channel dimension. In the next step, it unrolls the for-loops with activation feature maps. Finally, it unrolls for-loops across the output channels. At a similar technology node (14nm vs. 16nm), SPRING reduces chip area by 68% relative to GTX 1080 Ti.

Fig. 11 and Fig. 12 show the normalized performance of SPRING and GTX 1080 Ti over the seven CNNs in the training and inference phases, respectively. All results are normalized to those of GTX 1080 Ti. In the training phase, SPRING achieves speedups ranging from $5.5 \times$ to $53.1 \times$ with a geometric mean of $15.6 \times$ on the seven CNNs. In the inference phase, SPRING is faster than GTX 1080 Ti by $5.1 \times$ to $67.9 \times$ with a geometric mean of $15.5 \times$. In both cases, SPRING has better performance speedups on relatively light-weight CNNs, i.e., MobileNet V2, NASNet-mobile, and PNASNet-mobile. This is because these light-weight CNNs do not require large volumes of activations and weights to be transferred between the external memory and on-chip buffers. Therefore, the memory bandwidth bottleneck is alleviated and the speedup from sparsity-aware computation becomes more noteworthy. On the other hand, on large CNNs, such as Inception-Resnet V2 and VGG-19, the sparsity-aware MAC lanes of SPRING idle and wait for data fetch from the RRAM system, lowering the performance speedup relative to GTX 1080 Ti.

Fig. 13 and Fig. 14 show the normalized reciprocal of power of SPRING and GTX 1080 Ti in training and inference, respectively. All results are normalized to those of GTX 1080 Ti. On an average, SPRING reduces power dissipation by $4.2 \times$ and $4.5 \times$ for training and inference, respectively.

Fig. 15 and Fig. 16 show the normalized energy efficiency of SPRING and GTX 1080 Ti for training and inference, respectively. All results are normalized to those of GTX 1080 Ti. Compared to the GTX 1080 Ti, SPRING achieves an average of $66.0 \times$ and $69.1 \times$ energy efficiency improvements in training and inference, respectively. This makes the GTX 1080 Ti columns invisible. We observe that, among the seven CNNs, SPRING achieves the best normalized energy efficiency on MobileNet V2, both in the training and inference phases. Since MobileNet V2 has a much smaller network size (97.6% parameter reduction compared to VGG-19 [69]), most of the network weights can be retained in on-chip buffers without accessing the external memory. Hence, SPRING can reduce energy consumption significantly through our sparsity-aware acceleration scheme. On the other hand, energy reduction from sparsity-aware computation is offset by energy-consuming memory accesses on large CNNs, such as Inception-Resnet V2 and VGG-19. This is consistent with the results reported in [25] that show that over 80% of the total energy consumption is from memory access.

### 6 Discussions and Limitations

In this section, we discuss the assumptions we made in this work and the limitations of the SPRING architecture.

The performance speedup, power reduction ratio, and energy efficiency improvement reported in Section 5 are obtained at the batch level. We use batch-level training results since the CNN training results are based on the assumption that with sufficient precision bits, fixed-point training using stochastic rounding scheme can lead to convergence with no worse number of cycles than the training process based
on single-precision floating-point arithmetic, as suggested in [30], where 16 FL bits are used for fixed-point training with stochastic rounding and the convergent epoch number is similar to that of single-precision floating-point training.

A major limitation of the SPRING accelerator architecture is that the sequential scanning and filtering mechanism shown in Algorithm 1 needs multiple cycles to filter out dangling non-zero elements and collapse the resulting zeros. This may incur a long latency in data preprocessing, which makes SPRING unsuitable for latency-sensitive edge inference applications. However, since this sequential scanning and filtering scheme is pipelined, the overall throughput is unaffected and therefore the total latency for one batch is independent of the sequential scanning steps used by the pre-compute sparsity module.

Our binary mask encoding method is similar to the dual indexing encoding proposed in [77]. Although we both use a binary mask to point to the index of non-zero elements in the data vector, our binary mask encoding scheme has several advantages. First, the index masks are kept in binary form throughout the entire sparsity encoding and decoding process. Hence, the storage overhead of the binary mask is at most 5%, assuming 4 IL bits and 16 FL bits. The real storage overhead is much lower than this value since most of activations and weights are zeros. However, the binary masks are converted to decimal masks in [77] to serve as select signals of a MUX. This not only increases the storage overhead of the masks, but also increases the computation complexity of mask manipulation. Besides, their binary-to-decimal mask transfer process is sequential, which incurs a long processing latency that increases as the size of the mask vector increases.

7 Conclusion

In this article, we proposed a sparsity-aware reduced-precision CNN accelerator, named SPRING. A binary mask scheme is used to encode weight/activation sparsity. It is efficiently processed through a sequential scanning and filtering mechanism. SPRING adopts the stochastic rounding algorithm to train CNNs using reduced-precision fixed-point numerical representation. An efficient monolithic 3D
NVRAM interface is used to provide significant memory bandwidth for CNN evaluation. Compared to Nvidia GeForce GTX 1080 Ti, SPRING achieves 15.6×, 4.2×, and 66.0× improvements in performance, power reduction, and energy efficiency, respectively, in the training phase, and 15.5×, 4.5×, and 69.1× improvements in performance, power reduction, and energy efficiency, respectively, in the inference phase.

**References**

[1] 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 Proc. IEEE/ACM Int. Symp. Microarchitecture, Dec. 2014, pp. 609–622.

[2] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, “Scaledeep: A scalable computer architecture for learning and evaluating deep networks,” in Proc. Int. Symp. Computer Architecture, June 2017, pp. 13–26.

[3] C. Ying, S. Kumar, D. Chen, T. Wang, and Y. Cheng, “Image classification at supercomputer scale,” arXiv preprint arXiv:1811.06992, Dec. 2018.

[4] D. Foley and J. Danskin, “Ultra-performance Pascal GPU and NVLink interconnect,” IEEE Micro, vol. 37, no. 2, pp. 7–17, Mar. 2017.

[5] J. Choquette, O. Giroux, and D. Foley, “Volta: Performance and programmability,” IEEE Micro, vol. 38, no. 2, pp. 42–52, Mar. 2018.

[6] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chioiu, K. Constantines, J. Demme, H. Esmailiezhad, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, “A reconfigurable fabric for accelerating large-scale datacenter services,” in Proc. Int. Symp. Computer Architecture, 2014, pp. 13–24.

[7] J. Park and W. Sung, “FPGA based implementation of deep neural networks using on-chip memory only,” in Proc. IEEE Int. Conf. Acoustics, Speech, Signal Processing, Mar. 2016, pp. 1011–1015.

[8] Y. Shen, M. Ferdman, and P. Milder, “Overcoming resource underutilization in spatial CNN accelerators,” in Proc. Int. Conf. Field Programmable Logic Applications, Aug. 2016, pp. 1–4.

[9] J. Jiu, J. Wang, S. Yoo, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, “Going deeper with embedded FPGA platform for convolutional neural network,” in Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays, 2016, pp. 26–35.

[10] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, “ShiDianNao: Shifting vision processing closer to the sensor,” in Proc. ACM/IEEE Int. Symp. Computer Architecture, June 2015, pp. 92–104.

[11] S. Wang, D. Zhou, X. Han, and T. Yoshimura, “Chain-NN: An energy-efficient 1D chain architecture for accelerating deep convolutional neural networks,” in Proc. Design, Automation Test Europe Conf. Exhibition, Mar. 2018, pp. 674–687.

[12] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, “Cnvltin: Inefficient-neuron-free deep neural network computing,” in Proc. ACM/IEEE Int. Symp. Computer Architecture, June 2016, pp. 1–13.

[13] 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 Proc. IEEE/ACM Int. Symp. Microarchitecture, Oct. 2016, pp. 1–12.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep neural networks,” in Proc. Conf. Neural Information Processing Syist., Dec. 2012, pp. 1097–1105.

[15] M. A. Horowitz, “Fused-layer neural networks,” in Proc. ACM/IEEE Int. Symp. Computer Architecture, June 2016, pp. 737–739.

[16] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, “Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks,” in Proc. IEEE Int. Symp. High Performance Computer Architecture, Feb. 2017, pp. 553–564.

[17] Y. Chen, J. Emer, and V. Sze, “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” in Proc. ACM/IEEE Int. Symp. Computer Architecture, June 2016, pp. 737–739.

[18] H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-datacenter performance analysis of a tensor processing unit,” in Proc. Int. Symp. Computer Architecture, June 2017, pp. 1–12.

[19] M. Alwani, H. Chen, M. Ferdman, and P. Milder, “Fused-layer CNN accelerators,” in Proc. IEEE/ACM Int. Symp. Microarchitecture, Oct. 2016, pp. 1–12.

[20] T. Xiang, Y. Feng, X. Ye, X. Tan, W. Li, Y. Zhu, M. Wu, H. Zhang, and D. Fan, “Accelerating CNN algorithm with fine-grained dataflow architectures,” in Proc. IEEE Int. Conf. High Performance Computing Communications; IEEE Int. Conf. Smart City; IEEE Int. Conf. Data Science Syst., June 2018, pp. 243–251.

[21] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, “DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” in Proc. Int. Conf. Architectural Support Programming Languages Operating Syst., Mar. 2014, pp. 269–284.

[22] 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 Proc. Symp. Computer Architecture, June 2016, pp. 243–254.

[23] J. Zhu, J. Jiang, X. Chen, and C. Tsiu, “SparseNN: An energy-efficient neural network accelerator exploiting input and output sparsity,” in Proc. Design, Automation Test Europe Conf. Exhibition, Mar. 2018, pp. 241–244.

[24] 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 Proc. ACM/IEEE Int. Symp. Computer Architecture, June 2017, pp. 486–498.

[25] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. W. Fletcher, “UCNN: Exploiting computational reuse in deep neural networks via weight repetition,” in Proc. Int. Symp. Computer Architecture, June 2018, pp. 674–687.

[26] J. Albericco, P. Judd, T. Hetherington, J. Aamodt, N. E. Jerger, and A. Moshovos, “Cnvltin: Inefficient-neuron-free deep neural network computing,” in Proc. ACM/IEEE Int. Symp. Computer Architecture, June 2016, pp. 1–13.

[27] 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 Proc. IEEE/ACM Int. Symp. Microarchitecture, Oct. 2016, pp. 1–12.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in Proc. Conf. Neural Information Processing Syist., Dec. 2012, pp. 1097–1105.

[29] M. Rhu, M. O’Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, “Compressing DMA engine: Leveraging activation sparsity for training deep neural networks,” in Proc. IEEE Int. Symp. High Performance Computer Architecture, Feb. 2018, pp. 78–91.

[30] HMC Consortium. (2014) Hybrid Memory Cube specification 2.1. [Online]. Available: http://hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSK_FHMC_Specification_Rev21_20151105.pdf

[31] Samsung Newsroom. (2016) Samsung begins mass producing world’s fastest DRAM - based on newest High Bandwidth Memory (HBM) interface. [Online]. Available: https://www.samsung.com/semiconductor/news/news-events/samsung-begins-mass-producing-worlds-fastest-dram-based-on-newest-high-bandwidth-memory-hbm/

[32] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanam, “Deep learning with limited numerical precision,” in Proc. Int. Conf. Machine Learning, July 2015, pp. 1737–1746.

[33] Z. Lu, S. Rallapalli, K. Chan, and T. La Porta, “Modeling the resource requirements of convolutional neural networks on mobile devices,” in Proc. ACM/IEEE Int. Conf. Multimedia, 2017, pp. 1663–1671.

[34] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, “Cafeine: Towards uniformed representation and acceleration for deep convolutional neural networks,” in Proc. IEEE/ACM Int. Conf. Computer-Aided Design, Nov. 2016, pp. 1–8.
Ye Yu received the B.Eng. degree in Electronic and Computer Engineering from The Hong Kong University of Science and Technology, Hong Kong, China, in 2014, and the M.A. degree in Electrical Engineering from Princeton University, NJ, USA, in 2016. He is currently a Ph.D. candidate in the Electrical Engineering Department at Princeton University, NJ, USA.

His current research interests include low-power/high-performance VLSI design, monolithic 3D IC design, computer architecture for machine learning, deep learning acceleration, and computer vision.

Niraj K. Jha (S’85-M’85-SM’93-F’98) received his B.Tech. degree in Electronics and Electrical Communication Engineering from Indian Institute of Technology, Kharagpur, India in 1981, M.S. degree in Electrical Engineering from S.U.N.Y. at Stony Brook, NY in 1982, and Ph.D. degree in Electrical Engineering from University of Illinois at Urbana-Champaign, IL in 1985. He is a Professor of Electrical Engineering at Princeton University.

He has served as the Editor-in-Chief of IEEE Transactions on VLSI Systems and an Associate Editor of IEEE Transactions on Circuits and Systems I and II, IEEE Transactions on VLSI Systems, IEEE Transactions on Computer-Aided Design, IEEE Transactions on Computers, IEEE Transactions on Multi-Scale Computing System, Journal of Electronic Testing: Theory and Applications, and Journal of Nanotechnology. He is currently serving as an Associate Editor of Journal of Low Power Electronics. He has served as the Program Chairman of several conferences, the Director of the Center for Embedded System-on-a-chip Design funded by New Jersey Commission on Science and Technology, and the Associate Director of the Andlinger Center for Energy and the Environment.

He is the recipient of the AT&T Foundation Award and NEC Preceptorship Award for research excellence, NCR Award for teaching excellence, Princeton University Graduate Mentoring Award, and six Dean’s Teaching Commendations from the School of Engineering and Applied Sciences. He is a Fellow of IEEE and ACM. He received the Distinguished Alumnus Award from I.I.T., Kharagpur in 2014.

He has co-authored five books, 15 book chapters, and more than 440 technical papers, 14 of which have won various awards and six more that have been nominated for best paper awards. He has received 17 U.S. patents.

His research interests include monolithic 3D IC design, low power hardware/software design, computer-aided design of integrated circuits and systems, machine learning, smart healthcare, and secure computing. He has given several keynote speeches in the area of nanoelectronic design/test and smart healthcare.