SAVE: Sparsity-Aware Vector Engine for Accelerating DNN Training and Inference on CPUs

Zhangxiaowen Gong*†, Houxiang Ji*, Christopher W. Fletcher*,
Christopher J. Hughes†, Sara Baghsorkhi†, Josep Torrellas*

*University of Illinois at Urbana-Champaign   †Intel Labs
{gong15, hj14, cwfletch}@illinois.edu, {christopher.j.hughes, sara.s.baghsorkhi}@intel.com, torrella@illinois.edu

***Abstract*—General Matrix Multiplication (GEMM) is the key operation in Deep Neural Networks (DNNs). While dense GEMM uses SIMD CPUs efficiently, sparse GEMM is much less efficient, especially at the modest levels of unstructured sparsity common in DNN inference/training. Thus, most DNNs use dense GEMM.**

In this paper, we propose *SAVE*, a novel vector engine for CPUs that efficiently skips ineffectual computation due to sparsity in dense DNN implementations. SAVE's hardware extensions to the vector pipeline are transparent to software. SAVE accelerates FP32 and mixed-precision kernels with unstructured sparsity from both weights and activations. Further, SAVE is not DNN-specific and can potentially speed-up any vector workload with sparsity. To evaluate SAVE, we use simulations of a 28-core machine and run VGG16, ResNet-50, and GNMT, with and without pruning. With realistic sparsity, SAVE accelerates inference by 1.37x-1.68x and end-to-end training by 1.28x-1.64x.

I. Introduction

Deep Neural Networks (DNNs) have attained state-of-the-art results in tasks such as image recognition [40], speech recognition [4], scene generation [49], and game playing [55]. For DNN inference, CPUs are generally favored due to their flexibility, high availability, and low latency, especially when tight integration between DNN and non-DNN tasks is desired [25]. For DNN training, GPUs and accelerators provide higher raw compute power. However, the high memory capacity on CPU platforms makes training with large datasets and/or models easier [65]. Also, the high availability of datacenter CPUs encourages companies to distributedly train DNNs on CPUs during off-peak periods [61]. For example, Facebook trains their *Sigma* and *Facer* frameworks either entirely or partially on CPUs [25]. Other examples of training on CPUs include Intel's assembly and test factory [68], deepsense.ai's reinforcement learning [1], Kyoto University's drug design [21], Clemson University's natural language processing [9], GE Healthcare's medical imaging [33], and more [52]. Further, CPU makers have recently introduced features to accelerate training, such as BFloat-16 [65] and Intel Advanced Matrix Extensions [36]. Therefore, accelerating both DNN training and inference on CPUs is an important yet undervalued area.

Given this fact, our goal is to improve CPU performance on DNNs. We are inspired by literature that exploits *sparsity* in DNNs [3], [22], [24], [27], [38], [47], [48], [62], [66], [67]. Sparsity offers work-skipping opportunities due to the axiom that $x \cdot 0 = 0$. If a multiply operand is zero, the result is zero, so the multiplication is *ineffectual* and can be skipped.

In DNNs, sparsity is often *unstructured*. On one hand, the popular ReLU [26], [29], [30], [40], [43], [56], [60] activation function outputs zero between 40-90% of the time in an unpredictable way [47], [51]—creating dynamic, unstructured sparsity. On the other hand, weight pruning can drive weights to zero in training and/or inference [24], [41], [45].

To align the sparsity pattern with the underlying hardware, structured pruning has been proposed [5], [42], [62]. However, it often lowers the accuracy of the model more than unstructured pruning does at the same pruning rate. Luckily, while unstructured sparsity is difficult to exploit, prior work has shown significant performance gains from it by designing 'sparse' accelerators from the ground-up [3], [22], [47], [67].

Yet, exploiting unstructured sparsity on CPUs is notably more challenging. DNN workloads on CPUs consist of General Matrix Multiplications (GEMMs) that, for high performance, are vectorized. The goal is to maximize the utilization of Vector Processing Units (VPUs) performing Vector Fused Multiply-Add (VFMA) operations. To exploit unstructured sparsity, consider a naïve scheme that dynamically checks if vector lane operands are zero and, if so, skips the corresponding multiplications for those lanes. This approach can seldom improve performance because the vector instruction still has to wait for the other lanes to compute their results.

To address this challenge, we observe that VPUs are typically under-provisioned relative to other features in the core, to meet the needs of common-case applications. For example, Intel's Sunny Cove and AMD's Zen micro-architectures both support 2 VPUs (for 2 VFMA ops per cycle), yet have allocation/dispatch bandwidths of up to 5 and 6 micro-ops per cycle, respectively [15], [63]. This implies that in DNN codes, which are dominated by VFMAs, the VPU reservation stations fill quickly, and are bottlenecked waiting for the VPUs.

Based on this, our key idea is to add hardware that *searches* through the operands pending in reservation stations, to find and dynamically schedule effectual operations from different instructions to available VPU lanes. We call our new vector pipeline *Sparsity-Aware Vector Engine (SAVE)*. If a VFMA lane can be skipped because it is ineffectual, SAVE tries to find a pending lane from another VFMA to schedule there.

This work was funded in part by NSF under grants CNS 1956007, CNS 1763658, and CCF 1725734.

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00070

The result is fewer VPU operations, leading to speed-ups.

On top of this base idea, we introduce a number of additional optimizations. First, we design hardware-efficient mechanisms to load-balance VPU lanes. Second, because GEMM frequently issues vector broadcasts, we add a small high-bandwidth cache to exploit locality and improve broadcast throughput. Third, we devise techniques to handle the complications from mixed-precision VFMAs. Finally, we disable one VPU when it is idle due to high sparsity, and boost the core frequency.

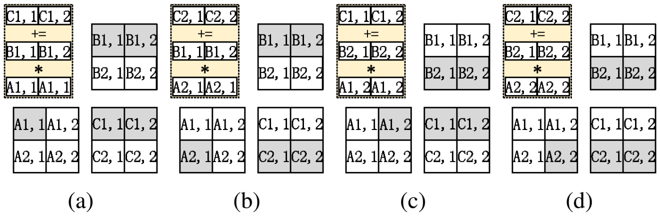
To the best of our knowledge, SAVE is the first CPU vector pipeline that exploits unstructured sparsity. SAVE is transparent to software and can thus benefit any legacy vector code. We evaluate SAVE using simulations of a 28-core machine. At realistic sparsity, SAVE speeds up the convolutional layers and LSTM cells in inference with dense VGG16, dense ResNet-50, pruned ResNet-50, and pruned GNMT by 1.68x, 1.37x, 1.59x, and 1.39x, respectively. Further, SAVE accelerates their end-to-end training by 1.64x, 1.29x, 1.42x, and 1.28x, respectively.

II. Background

A. Matrix Multiplication

GEMM is the core operation in DNNs. LSTMs [19] and batched MLPs use GEMM as a building block. Convolution can be computed either through (un)folding a big GEMM [11] or directly with a series of small GEMMs [18].

Fig. 1 illustrates a vectorized GEMM on a 2×2 tile with 2-lane vectors. In each step, the vector operation is shown at the top-left. The first step broadcasts the scalar $A_{1,1}$ to a vector, then element-wise multiplies the vector by $B_{1,[1:2]}$ and finally accumulates the product into $C_{1,[1:2]}$. The next three steps similarly multiply different broadcasted scalars from A and vectors from B, and accumulate into vectors from C.



(a) (b) (c) (d)

Fig. 1: Vectorized GEMM with two lanes on a 2×2 tile.

The GEMM reuses registers to reduce memory traffic. $C_{1,[1:2]}$ and $C_{2,[1:2]}$ are kept in accumulator registers throughout the computation. Further, (a) and (b) reuse $B_{1,[1:2]}$, while (c) and (d) reuse $B_{2,[1:2]}$. Although the broadcasted scalars from A are not reused in the example, accesses to A exploit spatial locality, e.g., (c) reads $A_{1,2}$ after (a) reads $A_{1,1}$. For larger matrices, tiling may create reuse of A.

B. Instruction Set Architecture for GEMM

We evaluate our ideas in an x86 environment with AVX-512 extensions [34]. However, the ideas we present are generalizable to other ISAs.

The core of GEMM is multiply-accumulate (MAC). Modern SIMD ISAs, such as ARM SVE [59] and Intel AVX-512 [34], include VFMA instructions. With AVX-512, a VFMA takes three operands of up to 512 bits in length each. A single VFMA can operate on 16 single-precision floating-point (FP32) lanes, where each lane i computes:

$$C'_i = C_i + A_i B_i \tag{1}$$

The accumulator and the two multiplicands can all be vector registers, or one multiplicand can be from memory. The memory operand can be a full vector or a scalar broadcasted to all vector lanes, supporting the use case in Fig. 1.

When a broadcasted scalar has high reuse, the software may use an explicit broadcast instruction to fill a vector register with the broadcasted scalar, and then reuse the register to reduce memory traffic. We call this the *explicit broadcast* pattern. On the other hand, if a broadcasted scalar has low reuse, the software may employ VFMA memory operands to minimize register pressure and increase code density. We call this the *embedded broadcast* pattern.

Reduced precision [39] and quantization [8] improve DNN performance. One can use lower precision multiplicands, but the accumulator often keeps a higher precision [28]. Vendors are adding mixed-precision MACs, such as Intel's AVX512VNNI (fixed-point) and AVX512_BF16 (Bfloat-16, or BF16), and ARM's BF16 extensions [6]. BF16 is a 16-bit FP format. BF16 and FP32 have the same dynamic range. Training in BF16 yields an accuracy comparable to that of using FP32, without tuning hyperparameters [39], [65].

A BF16/FP32 mixed-precision VFMA instruction, such as Intel's VDPBF16PS and ARM's BFDOT, operates on two multiplicand vectors with BF16 elements, and an accumulator vector with half as many FP32 elements. Two adjacent BF16 lanes map to one FP32 lane, forming a group. In each group, the instruction computes the dot product of the two-lane BF16 sub-vectors and then accumulates onto the FP32 accumulator:

$$C'_i = C_i + A_{[2i:2i+1]} \bullet B_{[2i:2i+1]} \tag{2}$$

We use • to denote vector dot product. VDPBF16PS computes the dot product in hardware by performing two consecutive MAC operations [34], shown in Fig. 2.

$$C'_i = C_i + A_{2i} B_{2i}$$
$$C''_i = C'_i + A_{2i+1} B_{2i+1}$$

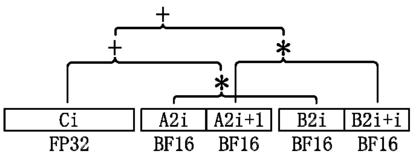


Fig. 2: Mixed-precision operation in VDPBF16PS.

C. Performance Bottleneck of DNN Kernels

For compute-bound DNN operations such as convolution, the main bottleneck is the throughput of the VPU. For example, Intel's Sunny Cove micro-architecture has 5-wide allocation but only 2 VPUs [63]. Similarly, AMD's Zen has 6-wide dispatch but only 2 VPUs [15]. As a result, their front-end bandwidth is heavily underutilized in DNN computations.

D. Sparsity in DNN Workloads and Ineffectual Operations

Sparsity has long been exploited in linear algebra codes. In DNN training and inference, where we operate on matrices, sparsity may arise in the activations and/or weights [23]. Networks that use the rectified linear unit (ReLU) [46] see 40-90% sparsity in activations. Dropout [58] also sparsifies activations, often to 50%. Sparsity in activations is innately unstructured. Weights sparsity comes from pruning, where an increasing fraction of weights is zeroed out during training. Pruned networks may be stored in a compressed representation for inference but are often in dense form during training, and masks are used for identifying dropped weights [69].

Pruning may be structured or unstructured. Structured pruning partitions weights into blocks of non-zero values to be efficiently processed by parallel hardware [5]. This requires considerable software effort and network tuning. Unstructured pruning [24], [69] is much easier to implement, but results in a pseudo-random pattern of non-zeros. Hardware for dense computations usually sees limited speedup from it.

When either multiplicand in a MAC is zero, the accumulator value does not change. We can skip such an ineffectual operation to increase the compute throughput. This is simple for scalar MACs but challenging for VFMAs. Unless all vector lanes are ineffectual, we cannot skip a VFMA.

III. Sparsity-Aware Vector Engine

We propose SAVE, a sparsity-aware vector engine for CPUs that skips ineffectual MAC operations. SAVE is transparent to software and speeds up DNN training and inference by exploiting *unstructured sparsity* in weights and activation.

Sparsity in a VFMA is either *non-broadcasted (NBS)* or *broadcasted (BS)*. NBS occurs when some elements of a vector are zero; BS occurs when a zero scalar is broadcasted to a vector. For NBS, SAVE combines the non-zero lanes from multiple VFMAs before issuing the VPU operation. This is feasible because the front-end bandwidth of server CPUs is higher than the VFMA throughput. Hence, DNN kernels quickly fill up the reservation stations (RS) with VFMAs. For BS, SAVE skips the entire VFMA, since it is ineffectual.

Fig. 3 shows the execution back-end and memory subsystem of a processor with SAVE. We show the added logic blocks in gray and the storage in black. In the rest of this section, we describe the basic SAVE architecture that skips ineffectual computation. In Section IV, we present advanced techniques that increase SAVE's performance. Finally, Section V introduces additional SAVE support for mixed-precision VFMAs.

To exploit NBS, we combine effectual lanes from multiple ready VFMAs. We call the set of ready VFMAs at a given time the *Combination Window (CW)*. Since modern CPUs have deep RS and ROB, we can have large CWs. However, VFMAs with the same accumulator have a true dependence, and only the oldest can be ready for execution. Hence, the number of VFMAs in the CW cannot exceed the number of accumulator registers. We observe that, for a large enough GEMM, with 32 ISA vector registers, the CW is often 24-28.

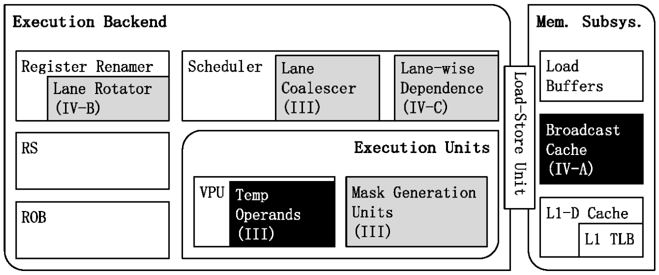


Fig. 3: SAVE adds logic units (gray) and storage units (black) to the execution backend and the memory subsystem. Each block is discussed in the section in the parentheses.

A VFMA's lane *i* is effectual when both multiplicands at lane *i* are non-zero. However, AVX-512 VFMAs can use write masks (WM) for predication, e.g., for dropped weights when pruning DNNs. The masked-out lanes are ineffectual.

SAVE generates an *Effectual Lane Mask (ELM)* for each VFMA, with one bit per lane. We allocate the ELMs from the AVX-512 mask physical register file (RF) to avoid additional storage cost [34]. In the studied GEMM kernels, one multiplicand is non-broadcasted while the other is broadcasted; however, we support NBS in both multiplicands *A* and *B* for generality. When *A* and *B* (and the WM, if used) of a VFMA are ready, SAVE schedules them to a *Mask Generation Unit (MGU)*. For each lane, the MGU checks the corresponding elements from *A* and *B*. If both are non-zero and the lane's WM bit (if present) is set, the hardware sets the lane's ELM bit. Fig. 4 shows this simple logic.

Because the MGU is simple, SAVE replicates it to process instructions in parallel. By matching the number of MGUs to the issue-width, the MGU throughput is never a bottleneck. We do not need the accumulator *C* of a VFMA to be ready before generating the ELM for the VFMA. A VFMA enters the CW once all of its operands and its ELM are available.

SAVE merges effectual lanes from multiple VFMAs in the CW into a temporary accumulator and two temporary multiplicands, collectively referred to as *temp*. To simplify the logic, SAVE keeps all the elements in their original vector lanes — an approach we refer to as *Vertical Coalescing*. Then, it computes a vector MAC in the VPU with the temp.

Algorithm 1 describes the scheduling algorithm of vertical coalescing. In each cycle, the scheduler first clears the temp (Line 1). Then, for each lane position (Line 2), it tests all entries in the RS *simultaneously* and finds the first VFMA with

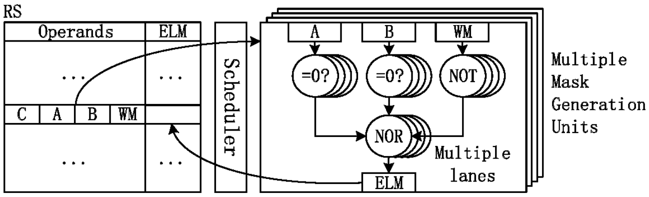
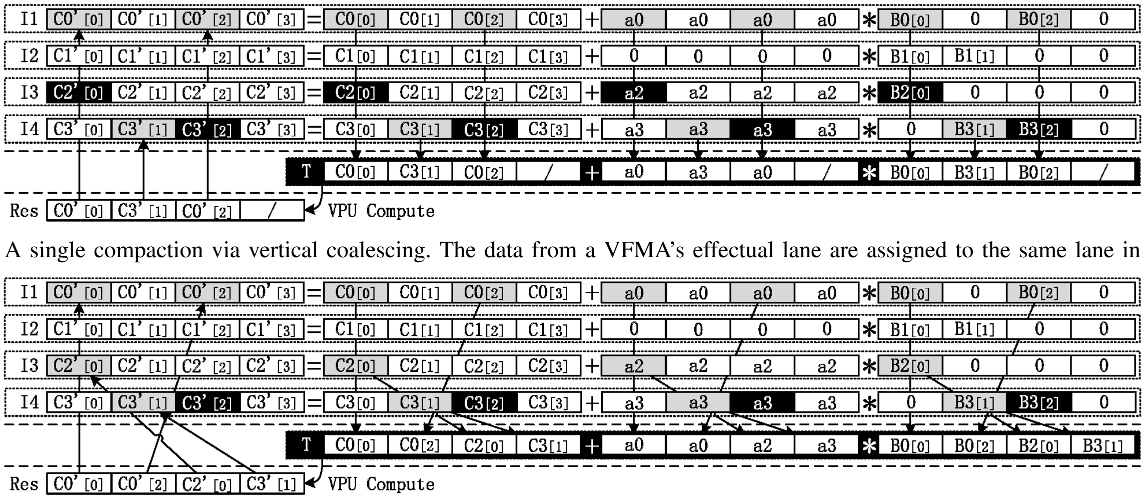


Fig. 4: Multiple Mask Generation Units (MGUs) producing the Effectual Lane Masks (ELMs) of multiple VFMAs in parallel. The RS shows a single VFMA being worked on.


I1 C0' [0] C0' [1] C0' [2] C0' [3] = C0[0] C0[1] C0[2] C0[3] + a0 a0 a0 a0 * B0[0] 0 B0[2] 0
I2 C1' [0] C1' [1] C1' [2] C1' [3] = C1[0] C1[1] C1[2] C1[3] + 0 0 0 0 * B1[0] B1[1] 0 0
I3 C2' [0] C2' [1] C2' [2] C2' [3] = C2[0] C2[1] C2[2] C2[3] + a2 a2 a2 a2 * B2[0] 0 0 0
I4 C3' [0] C3' [1] C3' [2] C3' [3] = C3[0] C3[1] C3[2] C3[3] + a3 a3 a3 a3 * 0 B3[1] B3[2] 0
T C0[0] C3[1] C0[2] / + a0 a3 a0 / * B0[0] B3[1] B0[2] /
Res C0' [0] C3' [1] C0' [2] /
VPU Compute

(a) A single compaction via vertical coalescing. The data from a VFMA's effectual lane are assigned to the same lane in T.

I1 C0' [0] C0' [1] C0' [2] C0' [3] = C0[0] C0[1] C0[2] C0[3] + a0 a0 a0 a0 * B0[0] 0 B0[2] 0
I2 C1' [0] C1' [1] C1' [2] C1' [3] = C1[0] C1[1] C1[2] C1[3] + 0 0 0 0 * B1[0] B1[1] 0 0
I3 C2' [0] C2' [1] C2' [2] C2' [3] = C2[0] C2[1] C2[2] C2[3] + a2 a2 a2 a2 * B2[0] 0 0 0
I4 C3' [0] C3' [1] C3' [2] C3' [3] = C3[0] C3[1] C3[2] C3[3] + a3 a3 a3 a3 * 0 B3[1] B3[2] 0
T C0[0] C0[2] C2[0] C3[1] + a0 a0 a2 a3 * B0[0] B0[2] B2[0] B3[1]
Res C0' [0] C0' [2] C2' [0] C3' [1]
VPU Compute

(b) A single compaction via horizontal compression. The data from a VFMA's effectual lane can be assigned to any lane in T.

Fig. 5: Comparison of compaction methods. $a0$-$a3$ are scalars broadcasted to all vector lanes; $B0$-$B3$ are vectors that contain non-broadcasted sparsity. Gray lanes are effectual and assigned to T; black lanes are effectual but will be scheduled later due to resource limitation; white lanes are ineffectual lanes. Lanes with "/" in T are unfilled. Arrows show the transfer of data.

an unscheduled effectual lane in the corresponding position. (Lines 3-9). This is done in a single cycle with conventional priority-based *select logic* [54]. If an effectual lane is found, it assigns the input operands to the temp (Lines 5-6) and records the source VFMA of the lane (Line 7) so that, after the computation, it can write the lane's result back to its proper destination. The ELM bit of a lane is cleared when the lane is assigned to the temp. After that, the scheduler issues a VPU operation if the temp contains effectual lane(s) (Lines 10-11) and removes from the RS any VFMA without unscheduled effectual lanes (Lines 12-14).

Algorithm 1: Scheduling of vertical coalescing

```
definition: temp operands $T$, reservation stations $RS$, vector
            processing unit $VPU$, effectual lane mask $ELM$,
            $\mu$op identifier $ID$, vector length $V$
1  clear(T);
2  for lane in V in parallel do
3      for $\mu op$ in RS do
4          if isVFMA($\mu op$) AND ready($\mu op$) AND
            $\mu op$.ELM[lane] then
5              T.accum_base[lane] = $\mu$op.accum_base[lane];
6              T.multiplicands[lane] = $\mu$op.multiplicands[lane];
7              T.ID[lane] = $\mu$op.ID;
8              $\mu$op.ELM[lane] = 0;
9              break
10 if NOT empty(T.ID) then
11     VPU.issue(T);
12 for $\mu op$ in RS in parallel do
13     if isVFMA($\mu op$) AND empty($\mu op$.ELM) then
14         RS.remove($\mu$op);
```

For simplicity, the algorithm describes scheduling to a single VPU. With N VPUs, each VPU has a temp. For a given lane position in the temps (Line 2), the algorithm selects up to N effectual lanes from ready VFMAs and assigns them to the N temps. This is a common practice when scheduling to multiple functional units [54]. Finally, for any VPU with a nonempty temp, we issue the compacted computation.

The algorithm can also handle BS because BS resembles a special case of NBS where all lanes are ineffectual. For BS, since all ELM bits are zero initially, the ineffectual μop is directly removed from the RS (Lines 12-14).

SAVE uses a VPU's existing input latch to hold the *temp* and thus avoids additional storage. However, SAVE needs to keep track of the source VFMA of each *temp* lane. The bookkeeping overhead is $VP\log_2(N_{RS})$ bits per VPU, where V is the vector length, P is the number of VPU pipeline stages, and N_{RS} is the number of RS entries.

Fig. 5a illustrates a single compaction via vertical coalescing with four quad-lane VFMA instructions, I_1-I_4, in program order. Each accumulator is shown both as an input (C) and as an output (C'), which are renamed to separate physical registers. $a0$-$a3$ are scalars broadcasted to all vector lanes. The three right-hand-side (RHS) inputs of the instructions' effectual lanes are combined into the temp T shown below them. The VPU then produces the result *Res* from T. Finally, each lane in *Res* is written back to the corresponding positions in the instructions' destinations. Because T is assembled from only the RHS inputs, in the rest of this paper's figures, we may omit the accumulation outputs for simplicity.

In the example, T gets I_1's lane 0, I_4's lane 1, and I_1's lane 2. Since all of I_1's effectual lanes are issued, we remove it from the RS. I_2 has BS, so it is entirely ineffectual and removed from the RS directly. No instruction has an effectual lane 3, so T's lane 3 is unused. Since vertical coalescing does not move elements across vector lanes, I_3's lane 0 and I_4's lane 2 cannot fill the hole in lane 3, and must wait. Lane conflicts like this can cause load imbalance when NBS in the CW is unevenly distributed among lanes.

When an exception occurs, if there are unscheduled effectual lanes from VFMAs that are before the faulting instruction in program order, the scheduling algorithm keeps executing until all those lanes complete. On the other hand, completed lanes from VFMAs that are after the faulting instruction are discarded when those VFMAs are squashed. Hence, the coalescing scheme does not jeopardize precise exceptions.

Another scheme for exploiting NBS is *Horizontal Compression*. This technique first bubble-collapses the ineffectual lanes of a VFMA. Then, it concatenates multiple VFMAs' effectual lanes into the temp. After the computation, it bubble-expands the results. Fig. 5b illustrates a single compaction via horizontal compression. In the example, T gets lanes 0 and 1 from I_1's lanes 0 and 2, lane 2 from I_3's lane 0, and lane 3 from I_4's lane 1. After issuing the VPU operation, I_1-I_3 are removed from the RS. Only I_4's lane 2 is left to be scheduled later.

Horizontal compression does not suffer from lane conflicts. However, bubble-expanding and collapsing add non-trivial latency and require expensive crossbars. Previous works have used horizontal compression to reduce the memory traffic in DNN workloads [2], [51]. Using it to reduce memory traffic is acceptable because (de)compression is only performed when loading from or storing to memory, so using existing permutation hardware in the VPU is sufficient. However, here we would need to bubble collapse and expand for each VFMA instruction, needing additional highly-expensive crossbars to keep up with the VFMA throughput. Hence, SAVE eschews horizontal compression but embraces vertical coalescing with additional optimizations to combat load imbalance.

With compaction, the scheduler may fill T with operands from multiple instructions. As a result, in each cycle, it may read more than the usual number of entries from the vector register file (RF). We could add read ports to the vector RF for this, but another option is available. Specifically, for each vector lane, we read only a single set of input elements (i.e., A, B, and C). Therefore, SAVE adopts a vector RF design where each lane of a vector register can be accessed independently. With this design, a vector RF with V lanes per vector register functions analogously to V independent scalar RFs.

IV. Advanced Features

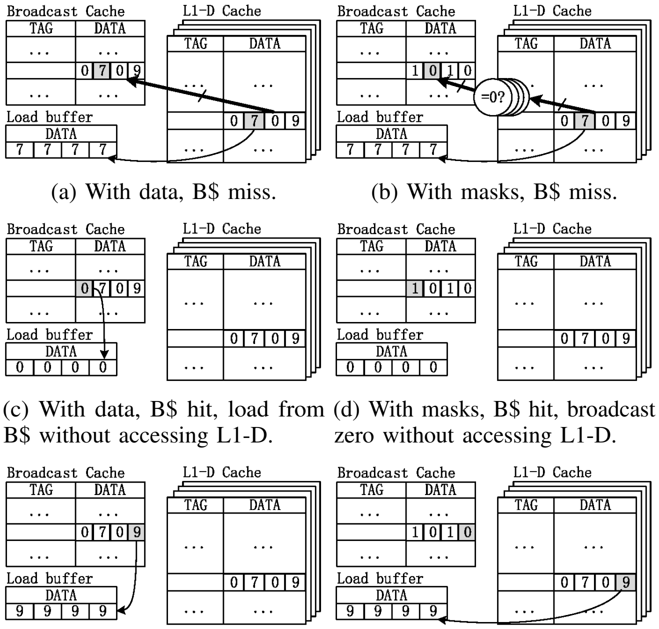
SAVE is enhanced with additional features to improve performance. First, because GEMM frequently issues vector broadcasts, we add a small high-bandwidth *Broadcast Cache* to exploit locality and improve broadcast throughput. Second, to load-balance VPU lanes, we introduce the *Rotate-Vertical Coalescing Scheme* and the *Lane-Wise Dependence Scheme*. Finally, we disable one VPU when it is idle due to high sparsity, and boost the core frequency. We now consider each technique.

A. Broadcast Cache

The basic SAVE design speeds up computation when VFMA throughput is the only bottleneck, such as in the explicit broadcast pattern (Sec. II-B), when broadcasted inputs are reused. However, the embedded broadcast pattern is limited by *both* VFMA throughput and L1-D cache read bandwidth. For example, in modern architectures such as Intel Skylake or AMD Zen, the number of L1-D read ports matches VFMA throughput [15]. Since the design so far does not reduce L1-D traffic, it hardly benefits embedded broadcast.

SAVE is enhanced to reduce memory pressure by exploiting spatial locality in the broadcasted values. In GEMM, different scalar values in the same cache line are broadcasted nearby in time. Hence, we capture this locality with a small, read-only cache, called the *Broadcast Cache (B$)*, that exclusively serves the broadcast load requests.

We propose two B$ designs: one where a line contains the values from the L1-D line that are broadcasted, and one where a line contains a mask indicating if each element in the L1-D line is zero. The designs are shown in the left and the right sides, respectively, of Fig. 6. In the figure, cache lines hold four vector elements.



(a) With data, B$ miss.

(b) With masks, B$ miss.

(c) With data, B$ hit, load from B$ without accessing L1-D.

(d) With masks, B$ hit, broadcast zero without accessing L1-D.

(e) With data, B$ hit, load from B$ without accessing L1-D.

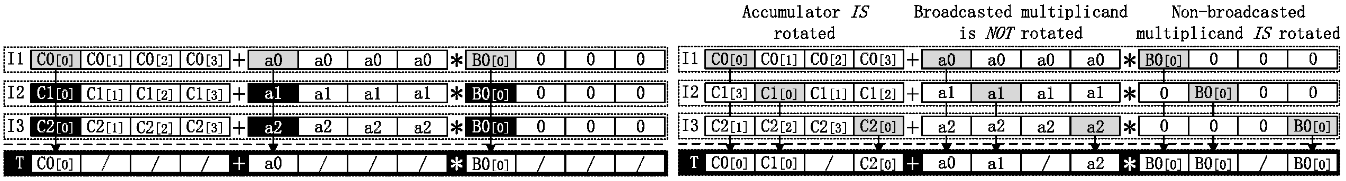
(f) With masks, B$ hit, broadcast non-zero, load from L1-D.

Fig. 6: Broadcast Cache with data (left) or with masks (right).

In the first B$ design, a broadcast μop checks the B$. On a miss (Fig. 6a), SAVE fetches the corresponding line from L1-D, stores it in the B$, and broadcasts the requested value to the load buffer. Future broadcast μops may hit in the B$, directly obtaining the broadcasted element from the B$, regardless of whether the element is zero (Fig. 6c) or not (Fig. 6e).

The second B$ design is shown on the right side of the figure. B$ only needs 16 bits per line, if we assume that the L1-D has 64B lines and 4B elements. When a broadcast μop misses in the B$, SAVE fetches the requested line from the L1-D and compares each element to zero, to generate the mask for the B$ (Fig. 6b). In addition, it broadcasts the requested element into the load buffer.

When a broadcast μop hits in the B$, SAVE checks the corresponding mask bit. If set (Fig. 6d), SAVE populates the load buffer with zeros and does not read the data from the


Accumulator IS rotated
Broadcasted multiplicand is NOT rotated
Non-broadcasted multiplicand IS rotated
I1 C0[0] C0[1] C0[2] C0[3] + a0 a0 a0 a0 * B0[0] 0 0 0
I2 C1[0] C1[1] C1[2] C1[3] + a1 a1 a1 a1 * B0[0] 0 0 0
I3 C2[0] C2[1] C2[2] C2[3] + a2 a2 a2 a2 * B0[0] 0 0 0
T C0[0] / / / + a0 / / / * B0[0] / / /
I1 C0[0] C0[1] C0[2] C0[3] + a0 a0 a0 a0 * B0[0] 0 0 0
I2 C1[3] C1[0] C1[1] C1[2] + a1 a1 a1 a1 * 0 B0[0] 0 0
I3 C2[1] C2[2] C2[3] C2[0] + a2 a2 a2 a2 * 0 0 0 B0[0]
T C0[0] C1[0] / C2[0] + a0 a1 / a2 * B0[0] B0[0] / B0[0]

(a) Vertical coalescing cannot combine lanes from instructions sharing a non-broadcasted multiplicand.

(b) Rotate-vertical coalescing. The operands from instruction I_2 are rotated right one lane; those from I_3 are rotated left one lane.

Fig. 7: Operand rotation combats load imbalance in vertical coalescing.

L1-D. Otherwise (Fig. 6f), SAVE fetches the data from the L1-D. Overall, this B$ design needs less storage, but it only skips an L1-D access when broadcasting zeros.

B$'s ideal size depends on how GEMM is register-tiled. We need one B$ line per accumulation buffer for C. The example in Fig. 1 uses 2 such buffers; therefore, we only need 2 entries in the B$ to capture the locality in A. More generally, the maximum number of B$ entries needed is the total number of architectural vector registers; the number of accumulation buffers cannot exceed this. In the context of AVX-512 with 32 vector registers, we give the B$ 32 entries. With a direct-mapped B$, we see $> 90\%$ hit rates for all tested DNN kernels.

This small B$ size allows more ports at a low cost. For our modeled core, 4 read ports are sufficient. We add additional address generation logic to support the ports. We keep the B$ coherent with the L1-D. Since the broadcasted inputs are read-only in GEMM, we do not expect B$ invalidations from other cores.

B. The Rotate-Vertical Coalescing Scheme

Vertical coalescing is sensitive to imbalanced load across lanes. Such imbalance is inherent when we reuse a register holding non-broadcasted data. Fig. 7a shows this case. All 3 instructions use register $B0$. Thus, their sparsity patterns are the same. Therefore, vertical coalescing cannot assign any effectual lanes from I_2 and I_3 to T due to conflicts. When we have such reuse, the *effective* CW shrinks significantly — the CW size is divided by the average number of reuses per register.

SAVE improves vertical coalescing by assigning a *Rotational State (R-state)* to each VFMA. Depending on the state, we rotate a VFMA's operands to the left or right by one lane, or do not rotate them at all. In this way, we limit the rotations and thus the hardware cost. Rotation eases the imbalance triggered by register reuse. We call this *Rotate-Vertical Coalescing*.

Fig. 7b shows an example. Fig. 7b is like Fig. 7a except that the operands of I_2 and I_3 are rotated right and left, respectively, both by one lane. After the rotations, the effectual lanes from the 3 instructions no longer conflict. The 3 R-states increase the effective CW by up to 3x.

Keeping copies of differently rotated operands consumes more physical registers; therefore, SAVE applies two optimizations to minimize the additional registers needed. First, because all scalar elements in the broadcasted multiplicand is the same, rotating it makes no difference. Hence, SAVE uses a single copy of the broadcasted multiplicand for all rotations. This is also reflected in Fig. 7b.

Second, SAVE assigns the same R-state to instructions with the same logical register as their accumulator; this ensures that a VFMA producing an accumulator and a VFMA consuming it are rotated the same way. Consequently, SAVE can keep a single copy of each accumulator. To implement it, SAVE determines an instruction's R-state by taking the logical register number for the accumulator, and performing a modulo operation with the total number of rotational states, which is 3. This also relieves SAVE from bookkeeping each instruction's R-state since it can be easily inquired through a table lookup.

With the two optimizations, SAVE only needs to store up to 3 copies of each non-broadcasted multiplicand (one for each R-state). Since the multiplicands are highly reused in GEMM kernels, the actual number of additional registers needed is low. We observe that, when running a typical explicit broadcast kernel, rotation consumes less than 25% additional registers. The number is much lower, less than 5%, when running a typical embedded broadcast kernel. We found that the size of the physical register file does not become a bottleneck with such additional consumption. As a result, we do not expand the register file.

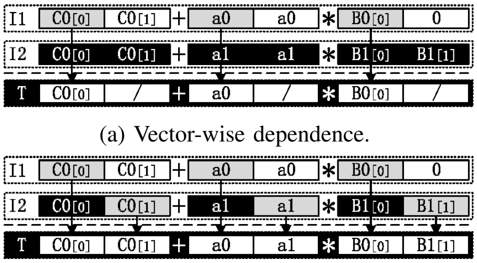
C. The Lane-Wise Dependence Scheme

Current SIMD processors track data dependences at the vector register granularity. A dependent VFMA is ready when all lanes in the source VFMA complete. We call this a *Vector-Wise Dependence*, which may create *false dependences* between VFMAs when vertical coalesing is employed.

We say that a dependent VFMA's lane i falsely depends on a source VFMA when, 1) lane i in the source is ineffectual or completed, and 2) some lanes in the source are not completed. Under these conditions, the inputs for the dependent's lane i are available, but we cannot schedule the lane. When the distances of true (RAW) dependences are short, false dependences frequently block otherwise issueable lanes.

Fig. 8a is an example of a false dependence. For simplicity, we consider two-lane vectors. Since I_1's lane 1 is ineffectual, we want to issue I_2's lane 1 simultaneously with I_1's lane 0. However, since both instructions accumulate into $C0$, a vector-wise dependence requires I_2 to wait.

SAVE eliminates false dependences by enforcing dependences at the lane level, called *Lane-Wise Dependence*. In this case, a dependent VFMA's lane i is ready as soon as the source VFMA's lane i completes. Fig. 8b illustrates that the scheme allows I_2's lane 1 to be issued along with I_1's lane 0. This is compatible with rotate-vertical coalescing because,



(a) Vector-wise dependence.

(b) Lane-wise dependence.

Fig. 8: Vector-wise dependence prevents I_2's lane 1 from issuing with I_1's lane 0. Lane-wise dependence does not.

as discussed, instructions with the same accumulator have the same R-state; thus, their lanes are still aligned after the rotation.

The naïve way to implement lane-wise dependence is to replicate the dependence logic for each lane. SAVE circumvents this by recognizing that the execution of VFMAs with the same accumulator respects true dependences. We honor dependences by scheduling effectual lanes in program order.

First, SAVE does not stall a VFMA if its only unresolved dependence is its accumulator's true dependence on a prior VFMA's accumulator. For example, in Fig. 8b, as soon as $a0$, $B0$, and $C0$ are available for I_1, and $a1$ and $B1$ are available for I_2, SAVE marks both I_1 and I_2 ready, despite the fact that I_2's accumulator $C0$ still depends on I_1's $C0$. Then, when SAVE schedules with Algorithm 1, for each lane, it selects the pending effectual lane from the *earliest* ready VFMA in program order in Lines 3-9. For example, SAVE schedules I_1's lane 0 before I_2's lane 0 because I_1 is earlier in program order. Prioritizing by program order is a well-known heuristic in conventional select logic and has mature implementations [54].

D. Power Saving and Frequency Boosting

Today's VPUs are so power hungry that the power managers may reduce core frequency when running vector code. For example, Intel downclocks its CPUs when running wide SIMD code [35]. However, at high sparsity, there are insufficient effectual lanes to keep all VPUs occupied. This means that reducing the number of VPUs would have little performance impact. Therefore, at high sparsity, we propose SAVE to disable a subset of the VPUs to save power. SAVE may change the number of active VPUs either statically through control registers, or dynamically through heuristics from performance counters. After disabling a VPU, the power manager may increase core frequency.

V. Mixed-Precision Techniques

Recently, manufacturers started to support mixed-precision VFMAs [6], [28], mostly for DNNs. The x86 implementation was described in Sec. II-B. We now extend SAVE to apply the techniques in Sec. III and IV to mixed-precision VFMAs.

For mixed-precision VFMAs, SAVE uses rotate-vertical coalescing to skip ineffectual FP32 Accumulator Lanes (ALs) in the C vector. However, because two BF16 Multiplicand Lanes (MLs) map to one FP32 AL, an AL is ineffectual only if *both* MLs are ineffectual. Consequently, sparsity in the multiplicands is typically not fully exploited.

Consider the example in Fig. 9, which omits rotation for simplicity. It shows 2 ALs, each mapped to 2 MLs. The dot product (denoted with •) of MLs $[0:1]$ accumulates into AL 0, and that of MLs $[2:3]$ accumulates into AL 1. In the figure, I_1's ML 1 is ineffectual. However, one cannot skip I_1's ML 1 because I_1's AL 0 needs to be scheduled due to I_1's ML 0 being effectual. Consequently, we cannot schedule I_2's AL 0 in this cycle. On the other hand, we can skip I_1's AL 1 and schedule I_2's AL 1 because ML 2 and 3 for I_1 are both ineffectual. Hence, T receives I_1's AL 0 and I_2's AL 1. However, MLs 1 and 3 in T are ineffectual. In general, if the multiplicands have random sparsity patterns, the level of exploitable sparsity is the square of the actual sparsity. e.g., when the multiplicands are 50% sparse, we only leverage $0.5^2 = 0.25$ or 25% sparsity.

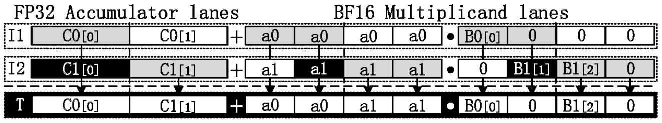


Fig. 9: Vertical coalescing is inefficient for mixed-precision. An accumulator lane can be skipped only when both multiplicand lane pairs mapped to it are ineffectual. The operator • stands for the dot product of two-lane multiplicand sub-vectors.

A. Horizontal Compression on Multiplicands

To address the above problem, SAVE combines effectual MLs from multiple VFMAs with the same accumulator. For example, assume that two instructions, I_1 and I_2, both accumulate to $C0$. Their ML $[2i{:}2i+1]$ map to AL i. Suppose that the ML $2i+1$ of I_1 and the ML $2i$ of I_2 are both ineffectual. Their computation for AL i becomes:

$$\begin{aligned} I_1 &: C0_i = C0_i + A0_{2i}B0_{2i} + 0 \\ I_2 &: C0_i = C0_i + 0 + A1_{2i+1}B1_{2i+1} \end{aligned} \tag{3}$$

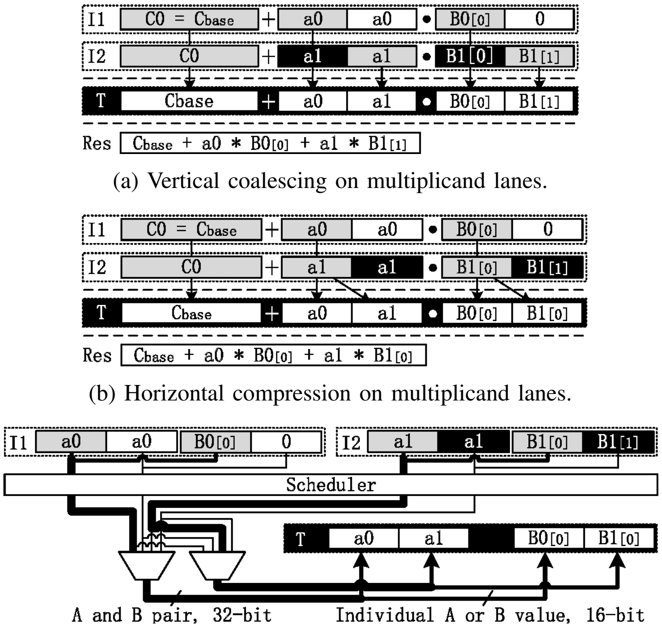
We can combine the two operations into a single one as:

$$C0_i = C0_i + A0_{2i}B0_{2i} + A1_{2i+1}B1_{2i+1} \tag{4}$$

We may combine the MLs via either horizontal compression or vertical coalescing, and both methods are correct with real number arithmetic. However, horizontal compression maintains the accumulation order, while vertical coalescing does not. Preserving the order is crucial to produce deterministic results with floating-point arithmetic.

For example, Fig. 10a *vertically* combines I_1's ML 0 and I_2's ML 1. I_2's ML 1 is accumulated into $C0$ before I_2's ML 0. This changes the accumulation order. In contrast, Fig. 10b *horizontally* schedules I_1's ML 0 and then I_2's ML 0, therefore preserving the accumulation order. The figures show the accumulated results in both cases, in *Res*.

For this reason, SAVE uses horizontal compression to combine MLs from mixed-precision VFMAs with the same accumulator. In Sec. III, we claimed that SAVE forsakes horizontal compression on the 16-lane vector due to hardware complexity. However, in the mixed-precision case, it is



(a) Vertical coalescing on multiplicand lanes.

(b) Horizontal compression on multiplicand lanes.

(c) Hardware of horizontal compression for mixed-precision VFMAs. Data pass through the thick lines in the case of the example in (b).

Fig. 10: Horizontal compression on multiplicand lanes preserves accumulation order while vertical coalescing does not.

acceptable to perform horizontal compression. This is because the complexity of the crossbar needed to perform horizontal compression is quadratic to the number of lanes. For the mixed-precision VFMAs, we permute MLs only within the 2 possible positions that map to the same AL; for the 16-lane vector, we would need to permute within 16 possible positions.

SAVE implements horizontal compression within each AL with two cheap 32-bit 4-to-1 multiplexers, as shown in Fig. 10c. Each multiplexer selects a pair of A and B multiplicands from 4 candidates. Furthermore, it only needs bubble-collapsing to compact the inputs into T; it does not need to bubble-expand the data out of the VPU after the computation.

B. Properly Writing Back Results

The above technique produces the correct result for the last instruction in a chain of VFMAs with the same accumulator. We define any VFMA before the last one in the chain as an *intermediate VFMA*. Although all instructions in the chain write to the same ISA register, with register renaming, their actual destinations are different physical registers. To be transparent to software and to support precise exceptions, SAVE also needs to write the correct results of all intermediate VFMAs to the destination physical registers. Taking Fig. 10b as an example, the result for I_1 should be $C_{base} + a0 \times B0[0]$, and the result for I_2 should be $C_{base} + a0 \times B0[0] + a1 \times B1[0] + a1 \times B1[1]$. However, the intermediate result computed with the temp is $R = C_{base} + a0 \times B0[0] + a1 \times B1[0]$, which is the proper result for neither I_1 nor I_2. The next step of the computation will issue I_2's ML1 and produce the correct final result for I_2.

By design, a mixed-precision VFMA performs two consecutive accumulations for each AL. The VPU uses the result from the first accumulation as the base of the second one. To produce correct values for the destination registers of intermediate VFMAs, we utilize both accumulation results.

When either result is available for an AL, we mark the ML that produces the result as completed. However, we do not write the result to the AL's destination until both of the MLs of the AL are completed. Otherwise, if a VPU operation's second result is not the final result of an AL for any VFMA, we define it as a *partial result*. A partial result is transient in nature and only useful as the base for a future accumulation. Hence, it should not update the architectural state of any instruction. Furthermore, if an exception happens, we discard the partial result and recompute it after serving the exception. In SAVE, we avoid storing the partial result by immediately scheduling the next VPU operation in the chain, and *forwarding* the partial result to the VPU as the accumulation base.

C. Example of a Mixed-Precision VFMA

For simplicity, we show an example instead of listing the complete algorithm. Fig. 11 illustrates how SAVE generates proper results for a single AL from 3 instructions with the same accumulator $C0$. The figure also illustrates the register renaming for the accumulator. In the following, we assume that a VFMA takes two cycles to finish, and that the first result is out after one cycle. The instructions have RAW dependence on $C0$, so the example does not pipeline the VPU operations.

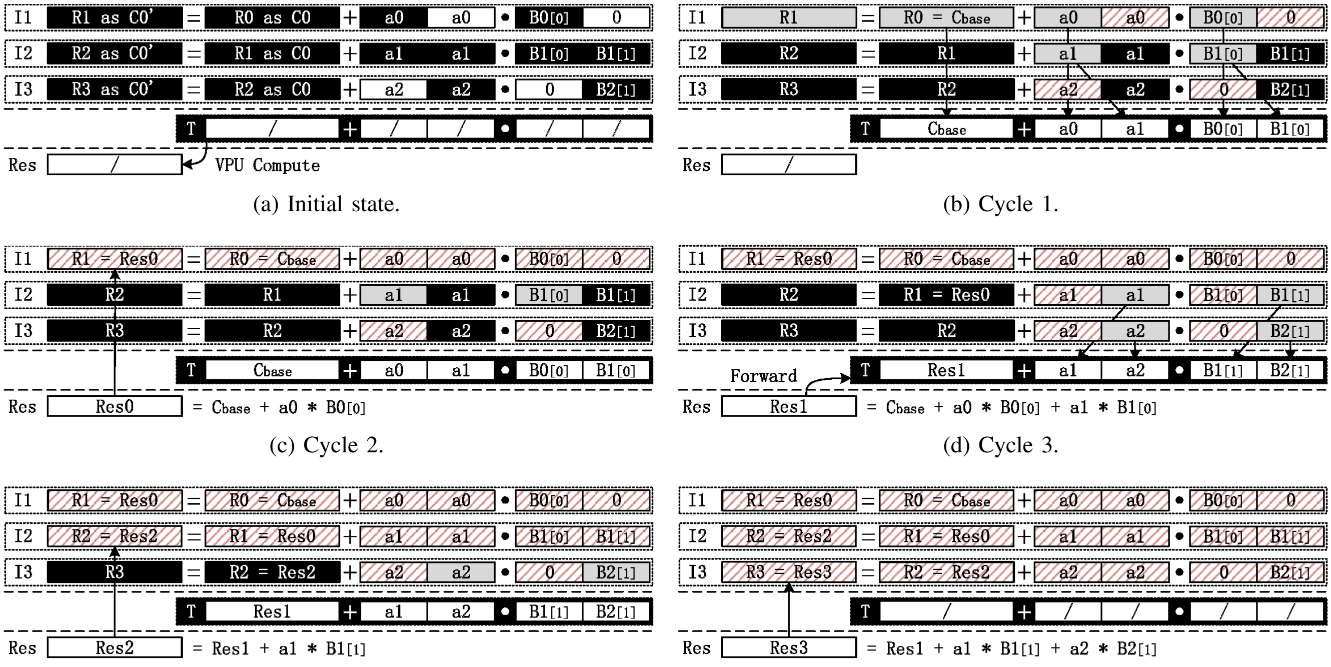
Fig. 11a shows the initial state, before any operation occurs. No effectual lane has been scheduled, and T is empty. In each instruction, $C0$ is renamed to two physical registers: one as the accumulation base (e.g., $R0$ in I_1), and the other as the accumulation destination (e.g., $R1$ in I_1). A subsequent instruction's accumulation base is renamed to the same physical register as the previous instruction's destination (e.g. $R1$ in both I_1 and I_2). SAVE fills T with the accumulation base and the multiplicands. After the computation, SAVE takes the output result (*Res* in the figure) and updates the instruction's destination register accordingly. The rest of the examples replace $C0$ with the actual physical registers: $R1$ to $R3$.

In cycle 1 (Fig. 11b), the initial value of C_0 is C_{base}, held in $R0$. We issue the first VPU operation with ML 0 from both I_1 and I_2, and use *Cbase* as the accumulation base. The ineffectual lanes are marked as completed with a slashed pattern.

In cycle 2 (Fig. 11c), the first result (Res_0) of the first VPU operation is available. I_1's ML 0 completes. Because both MLs of I_1 are completed, we update I_1's destination R_1 with Res_0.

In cycle 3 (Fig. 11d), there are two operations. First, the second result (Res_1) of the first VPU operation is produced. We mark I_2's ML 0 as completed. However, since I_2's ML 1 is unprocessed, Res_1 is not the result for I_2, so we do not write it to I_2's destination R_2. The second operation is scheduling the next VPU operation on accumulator $C0$. Because Res_1 is the accumulation base, we forward Res_1 to the VPU input directly (see forward arrow to T). We schedule ML 1 of both I_2 and I_3 for the next VPU operation.

In cycle 4 (Fig. 11e), the first result of the second VPU operation (Res_2) is out. We mark I_2's ML 1 complete. Since


I1 R1 as C0' = R0 as C0 + a0 a0 • B0[0] 0
I2 R2 as C0' = R1 as C0 + a1 a1 • B1[0] B1[1]
I3 R3 as C0' = R2 as C0 + a2 a2 • 0 B2[1]
T / + / / • / /
Res / VPU Compute
I1 R1 = R0 = Cbase + a0 a0 • B0[0] 0
I2 R2 = R1 + a1 a1 • B1[0] B1[1]
I3 R3 = R2 + a2 a2 • 0 B2[1]
T Cbase + a0 a1 • B0[0] B1[0]
Res /
I1 R1 = Res0 = R0 = Cbase + a0 a0 • B0[0] 0
I2 R2 = R1 + a1 a1 • B1[0] B1[1]
I3 R3 = R2 + a2 a2 • 0 B2[1]
T Cbase + a0 a1 • B0[0] B1[0]
Res Res0 = Cbase + a0 * B0[0]
I1 R1 = Res0 = R0 = Cbase + a0 a0 • B0[0] 0
I2 R2 = R1 = Res0 + a1 a1 • B1[0] B1[1]
I3 R3 = R2 + a2 a2 • 0 B2[1]
Forward
T Res1 + a1 a2 • B1[1] B2[1]
Res Res1 = Cbase + a0 * B0[0] + a1 * B1[0]
I1 R1 = Res0 = R0 = Cbase + a0 a0 • B0[0] 0
I2 R2 = Res2 = R1 = Res0 + a1 a1 • B1[0] B1[1]
I3 R3 = R2 = Res2 + a2 a2 • 0 B2[1]
T Res1 + a1 a2 • B1[1] B2[1]
Res Res2 = Res1 + a1 * B1[1]
I1 R1 = Res0 = R0 = Cbase + a0 a0 • B0[0] 0
I2 R2 = Res2 = R1 = Res0 + a1 a1 • B1[0] B1[1]
I3 R3 = Res3 = R2 = Res2 + a2 a2 • 0 B2[1]
T / + / / • / /
Res Res3 = Res1 + a1 * B1[1] + a2 * B2[1]

(a) Initial state.

(b) Cycle 1.

(c) Cycle 2.

(d) Cycle 3.

(e) Cycle 4.

(f) Cycle 5.

Fig. 11: Example of using mixed-precision VPU operation to properly update the destination registers in a set of instructions using the same accumulator. The completed lanes of each VFMA are shown in a slashed pattern.

both MLs of I_2 are complete, we update I_2's destination $R2$ with Res_2. Finally, in cycle 5 (Fig. 11f), the second VPU operation finishes. The result Res_3 is written back to I_3's destination $R3$.

Because this design updates every destination register with the correct result, SAVE guarantees correct architectural state when completing any mixed-precision VFMA. Hence, SAVE supports precise exceptions.

VI. Experimental Setup

We implement SAVE in the Sniper multicore simulator [10]. Table I lists the modeled processor, which resembles the 28-core Intel Skylake Xeon 8180 CPU, with the exception that we widen the issue-width to 5 μop/cycle, up from 4. This reflects a change in the newer Sunny Cove architecture. In addition, because Sniper does not support non-inclusive caches, we model Skylake's 1.375MB/core non-inclusive L3 with a 2.375MB/core inclusive cache. We model the latency and execution ports of common instructions [14]. We set the latency of a FP32 VFMA to 4 cycles, as in the Skylake; we set the unknown latency of a mixed-precision VFMA to 6 cycles, since it needs simpler multipliers but an additional accumulation.

In each cycle, the Xeon 8180 can execute up to two 256-bit AVX2 instructions at 2.1GHz or up to two AVX-512 instructions at 1.7GHz [35]. Because one 512-bit VPU is broken down into two 256-bit units when executing AVX2 code [15], executing one 512-bit VFMA draws power comparable to executing two 256-bit VFMAs. Therefore, we evaluate SAVE at 1.7GHz with two 512-bit VPUs and at 2.1GHz with one 512-bit VPU. The baseline has two 512-bit VPUs at 1.7GHz. The core frequency affects L1 and L2 but not L3.

TABLE I: Architecture configuration.

Core	28 cores, no SMT, 97 RS entries, 224 ROB entries, 5-issue, 1 VPU at 2.1GHz or 2 VPUs at 1.7GHz
B$	32 lines direct-mapped, with data or with masks
L1-D/I	32KB/core private, 8-way, LRU
L2	1MB/core private, inclusive, 16-way, LRU
L3	2.375MB/core, shared, inclusive, 19-way, SRRIP, NUCA
NoC	2D-mesh, XY routing, 2-cycle hop
Memory	119.2GB/s BW, 6 channels, 50ns latency

TABLE II: Storage structures in SAVE modeled at 22nm.

	Only supports FP32			FP32 and mixed-precision		
	Size	P_{leak}	E_{access}	Size	P_{leak}	E_{access}
T per VPU	56B	N/A	N/A	168B	N/A	N/A
B$ w/ mask	276B	0.24mW	2.9E-4nJ	340B	0.29mW	3.8E-4nJ
B$ w/ data	2260B	3.2mW	1.6E-2nJ	2260B	3.2mW	1.6E-2nJ

With the above configurations, we list SAVE's storage overhead in Table II. We also model the leakage power and access energy of the broadcast cache (B$) configurations using CACTI 7.0 [7] at the 22nm process.

We evaluate the training and inference of popular CNNs and LSTMs. To compute the convolutional (conv) layers and the LSTM cells, we use the kernels from Intel DNNL [32] (formerly MKL-DNN), a state-of-the-art AVX-512 DNN library.

For CNNs, we choose ResNet-50 [26] and VGG16 [56] on ImageNet-1K [13]. Because VGG16's activation sparsity is high [51], evaluating a pruned version would not provide additional insights. Therefore, we evaluate VGG16 with dense weights. In ResNet-50, the residual connections lower the activation sparsity by adding positive bias before ReLU. Its use of batch normalization (BatchNorm) [37] further eliminates the sparsity in the output gradient during training. Therefore, we evaluate ResNet-50 with both dense and pruned weights.

For LSTMs, we choose GNMT [64] on WMT'16 EN-DE. Since GNMT does not employ ReLU, the activation sparsity is from dropout with a constant rate of 20%. The activation sparsity further diminishes when the input is concatenated with the previous output. Therefore, we only evaluate GNMT with pruned weights because the activation sparsity is low.

Table III lists the types of sparsity (Broadcasted Sparsity or Non-Broadcasted Sparsity) that are present in inference and in different phases of training. For CNNs, DNNL has two phases in the backward propagation: propagation of input and propagation of weights. For LSTMs, the two phases are merged. Note that when training ResNet-50 without pruning, the backward propagation of input has no sparsity.

TABLE III: Types of sparsity in the evaluated networks.

CNN	forward/inference		backward input		backward weights	
	BS	NBS	BS	NBS	BS	NBS
dense VGG16	✓		✓		✓	✓
dense ResNet-50	✓				✓	
pruned ResNet-50	✓	✓		✓	✓	

LSTM	forward/inference		backward	
	BS	NBS	BS	NBS
pruned GNMT	✓	✓	✓	✓

Because full training in a simulator is infeasible, we use a sampling method to estimate SAVE's performance. First, we need the realistic weight and activation sparsity during full training runs. For VGG16, we use the sparsity progression reported by Rhu et al. [51]. For ResNet-50, we profile the sparsity during training with and without pruning. The 90-epoch dense training gives a 76.7% top-1 accuracy. We prune using a magnitude based method [69] with the hyperparameters from [17] that yields a 75.4% top-1 accuracy. We start pruning at epoch 32 and stop at 80% target sparsity in epoch 60. The training stops at epoch 102. The weights in each layer are pruned at the same rate. We do not compress the pruned model. For GNMT, we start pruning at iteration 40K and stop at 90% target sparsity at iteration 190K. The training stops at iteration 340K. The final BLEU score is 28.4 [31].

Fig. 12 presents the progression of activation sparsity during training. We omit GNMT since its activation sparsity is constantly 20%. Fig. 13 shows the schedule of weight pruning. We assume that, without pruning, the weights are fully dense.

Next, for each layer, we simulate SAVE with both weight and activation sparsities of 0%-90% at 10% intervals, using a uniform random distribution. The result is a 2D surface of execution times with 100 different combinations of weight and activation sparsities. We warm up L3 with the output from the previous DNN operation: for forward propagation, it is the input activation; for backward propagation, it is the output gradient. The weights and the layer's results are cold.

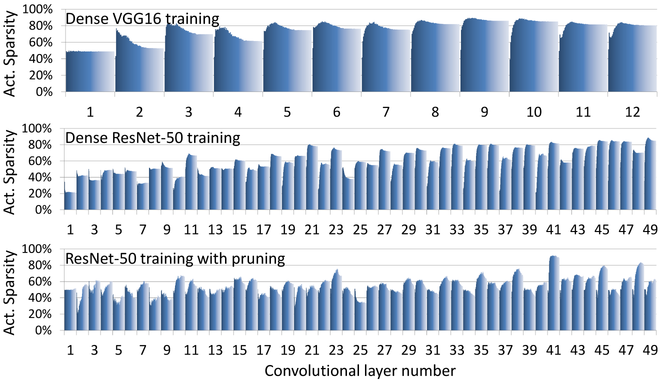


Fig. 12: Activation sparsity during end-to-end training. Each x-axis segment shows a layer. Within a segment, from left to right are the sparsities from the first epoch to the last one.

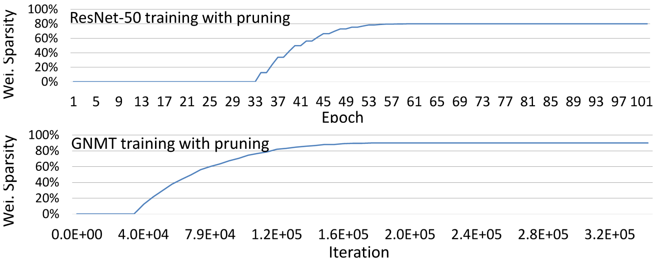


Fig. 13: Schedule of weight pruning.

Finally, we calculate SAVE's mean performance on end-to-end training. For each epoch and layer, we linearly map the profiled weight and activation sparsities to the 2D surface of execution times computed above, and obtain the execution time of the layer at the epoch. We sum all the layers' execution times at an epoch to get the run time of the whole network at the epoch. At last, we take the average of all the epochs as SAVE's mean network execution time during training.

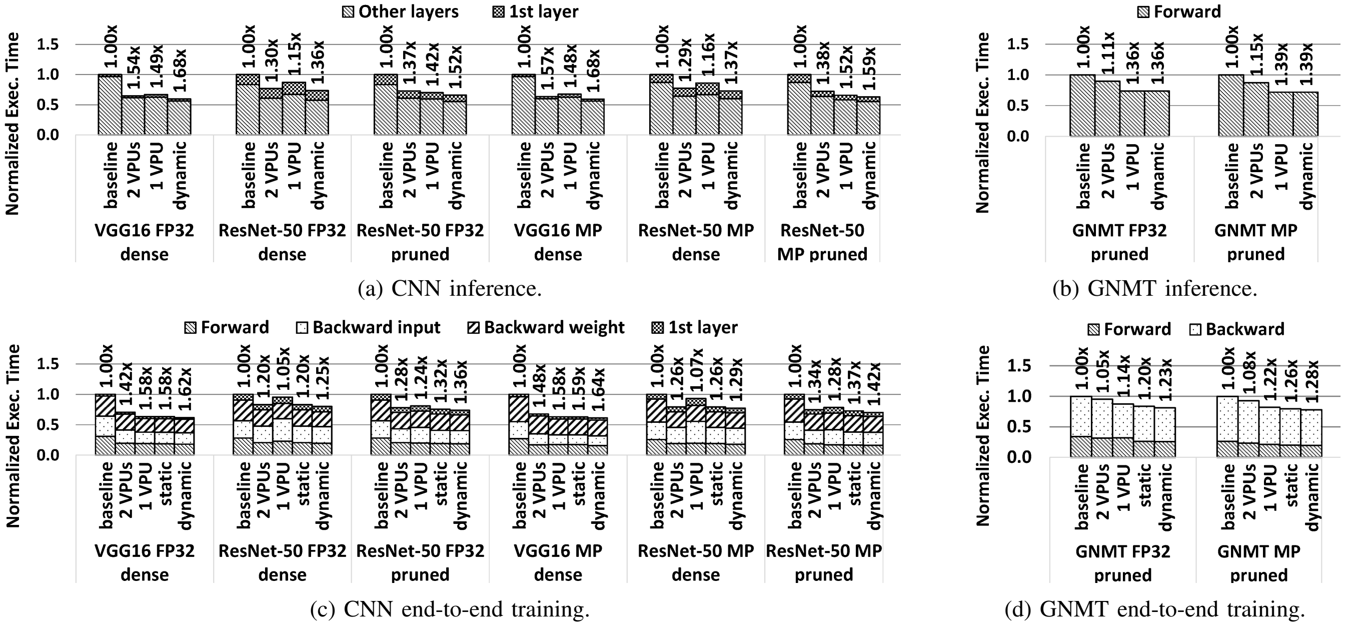
To compute the execution time of inference, we simulate with the sparsity obtained at the end of training.

VII. Evaluation

We first present SAVE's performance on whole-network training/inference with the complete set of SAVE's features. We then discuss the impact of the different features of SAVE.

A. *Whole Neural Network Performance*

We assess SAVE's whole-network training and inference performance with all SAVE features. We configure the broadcast cache to store the data. Fig. 14 shows the normalized execution time of all conv layers or LSTM cells in the studied networks. For each network, we show bars for the baseline and for several configurations of SAVE: 1) using two VPUs, 2) using one VPU at higher frequency, 3) for each training epoch, statically using the better of one or two VPUs (*static* bars), and 4) for each DNN kernel, dynamically using the better of one or two VPUs (*dynamic* bars). Configuration 3 does not apply to inference because the switching interval is much coarser than an inference. Configuration 4 neglects



(a) CNN inference.

(b) GNMT inference.

(c) CNN end-to-end training.

(d) GNMT end-to-end training.

Fig. 14: Execution time of all conv layers or LSTM cells in the studied networks at realistic sparsity, normalized to the baseline.

any overhead for enabling/disabling a VPU and changing the frequency. The reason is that the switching overhead of a typical DVFS manager is around ten microseconds, while our configuration switches at tens of milliseconds. In addition, a VPU's warm-up period is even smaller. Each bar is labeled with the speedup of the configuration over the baseline.

Fig. 14a and Fig. 14c are for CNN inference and training respectively. They show times for dense VGG16, dense ResNet-50, and pruned ResNet-50 at realistic sparsity, each with FP32 and with mixed precision (MP). The bars are broken down into two or more categories. For inference and training, we separate the first layer because 1) it does not have sparse input activations, and 2) it does not compute the back-propagation of input. For training, we also distinguish between forward propagation and backward input and weight propagation.

The figures show that SAVE delivers substantial speedups over the baseline. Configuration 4 performs the best: SAVE with mixed precision speeds-ups dense VGG16, dense ResNet-50, and pruned ResNet-50 by 1.68x, 1.37x, and 1.59x, for inference, and by 1.64x, 1.29x, and 1.42x, for training. The speedups are slightly lower for FP32 and for other configurations.

When SAVE uses a fixed number of VPUs, most workloads perform better with two VPUs. This is because, while many kernels have high sparsity and can benefit from using one VPU at higher frequency, some kernels have dense inputs, and thus prefer two VPUs. For example, the first layer in a CNN has no activation sparsity, and for training the dense ResNet-50, back-propagation of input has sparsity in neither weights nor activations due to Batch Normalization [37].

Configuration 3 performs better than using a fixed number of VPUs since the sparsity level changes during training. Configuration 4 further speeds-ups both training and inference because each kernel's input has different sparsity levels.

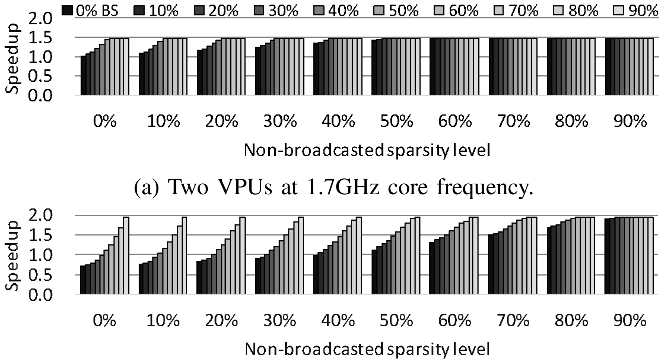
SAVE achieves higher speed-up on VGG16 than on ResNet-50. One reason is that, in VGG16, the first layer (which has no activation sparsity) contributes a smaller portion of the total execution time than in ResNet-50. Also, VGG16 does not incorporate Batch Normalization, so its back-propagation of input has sparsity in the activation gradient. Finally, VGG16's activation sparsity is on average higher than ResNet-50's.

Fig. 14b and Fig. 14d are for GNMT inference and training respectively. In inference, the bars are not broken down; in training, they are broken down into forward and backward. We see that SAVE delivers sizeable speedups over the baseline. For the dynamic configuration, SAVE with mixed precision attains a speedup of 1.39x for inference and 1.28x for training.

Because LSTM have lower compute-to-memory ratios than CNN, it becomes memory bound more easily as SAVE reduces computation. Hence, the speedups are on average lower than on the CNNs. It can be shown that, with two VPUs, the speedup is capped when the weights are 20% pruned; with one VPU, we continue to see speedup until the weights are 60% pruned.

B. Boosting Frequency with Fewer VPUs

We study the effect of using one or two VPUs at different core frequencies. Fig. 15 shows SAVE's speedup on the ResNet2_2 kernel with two VPUs (a) or one VPU (b) at different sparsity levels. Each bar group corresponds to a different NBS level. Within a group, each bar corresponds to a different BS level. At 0% total sparsity, using two VPUs matches the baseline performance, while using one VPU gives a 29% slowdown. As sparsity increases, SAVE's benefit increases. With two VPUs, SAVE's benefit is capped at 1.49x, when either the BS or NBS level reaches around 60%. Then, the execution is no longer throttled by VPU throughput. With one VPU, we benefit from higher sparsity, up to at least 90% of either type, and reach a maximum speedup of 1.96x. When either type of sparsity exceeds 70%, one VPU outperforms two.



(a) Two VPUs at 1.7GHz core frequency.

(b) One VPU at 2.1GHz core frequency.

Fig. 15: SAVE speedups on the mixed-precision forward propagation of ResNet2_2 with 1 or 2 VPUs.

At high sparsity, the speedup reaches a ceiling because the execution becomes memory, frontend, or latency bound, depending on the kernel. Unless the execution is L3 or DRAM bound, higher core frequency usually helps. The speedup caps of the 93 studied kernels are considered in Fig. 16, for FP32 and mixed precision (MP), and 2 and 1 VPUs. The figure counts the number of kernels whose speedup caps are within a range, for conv layers and LSTM cells. We see that using 1 VPU and boosting the frequency effectively lifts the caps. For FP32, the geometric mean of the speedup cap is 1.39x with two VPUs and 1.62x with one VPU. For mixed precision, it is 1.48x with two VPUs and 1.77x with one VPU.

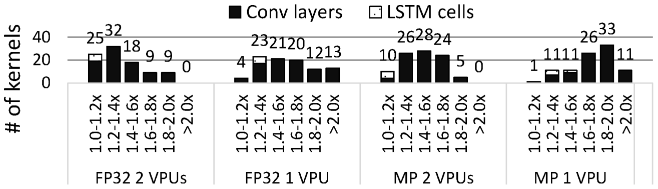


Fig. 16: Histograms of the speedup caps. Each bar counts the number of kernels whose speedup caps are within a range.

C. Broadcast Cache Designs

To address the L1-D read bandwidth limitation under an embedded broadcast pattern, SAVE introduces the B$. We proposed two designs of the B$: one holds data and the other holds masks. The second design saves storage. However, the requested non-zero elements are always fetched from L1-D. If a workload with embedded broadcast only has BS, this is not a problem because the reduction in L1-D read requests from sparsity matches the reduction in VFMA operations. However, if the workload also has NBS, the reduction in VPU operations may make L1-D bandwidth a bottleneck again.

Fig. 17 shows the speedups from SAVE with the two B$ designs running a kernel with an embedded broadcast pattern. It also shows the speedups without a B$. The figure shows BS levels of 0% and 40%, and different NBS levels. Without a B$, we do not get speedup at any level of NBS or BS. Without NBS, as BS increases, both types of B$ designs deliver speedups. However, as NBS increases, B$ with data typically delivers additional speedup, while B$ with masks does not due to the L1-D bandwidth bottleneck discussed earlier. Consequently, a B$ is essential to speeding up the embedded broadcast pattern, and a B$ with data performs much better than a B$ with masks.

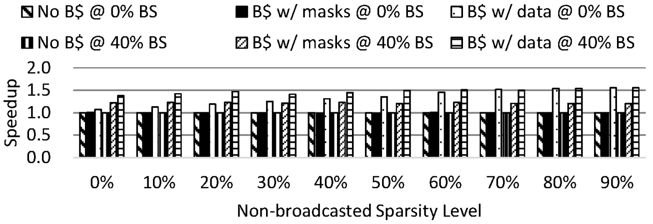
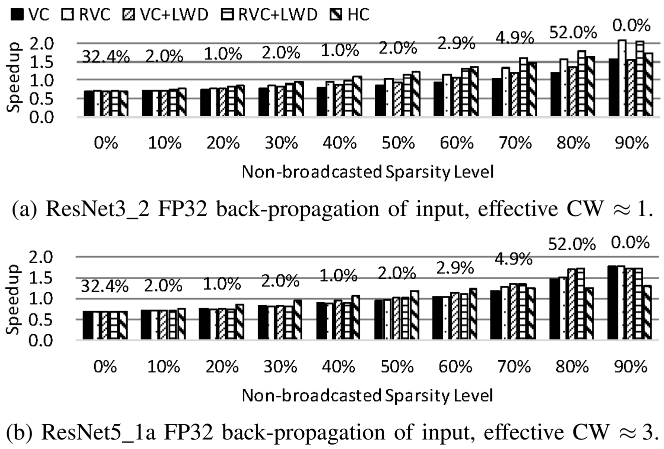


Fig. 17: SAVE speedups with different B$ designs on the FP32 back-propagation of weights of ResNet3_2 with two VPUs.

D. Techniques for Load-Balancing VPU Lanes

We now compare vertical coalescing (VC), rotate-vertical coalescing (RVC), lane-wise dependence (LWD), and combinations of them in an environment with only NBS. We also include the impractical horizontal compression (HC) for comparison. For HC, we use the 3-cycle latency of AVX-512 vector permutation (i.e., VPERMPS) [14] as the cost of bubble collapsing/expanding, so we add 6 cycles to VFMA's latency.

Fig. 18 shows, for two kernels, the speedups of these techniques over the two-VPU baseline. We use NBS levels of 0%-90%, 0% BS, and one VPU. We choose two kernels of back-propagation of input in pruned ResNet-50 because this is the only case when NBS is present while BS is not (Table III). To correlate the speedups with realistic sparsity, we list, above the bars, the percentage of training iterations where the NBS of the layer is ±5% of the sparsity written below the bars. Therefore, bars with higher numbers are more representative.



(a) ResNet3_2 FP32 back-propagation of input, effective CW ≈ 1.

(b) ResNet5_1a FP32 back-propagation of input, effective CW ≈ 3.

Fig. 18: SAVE speedups with different techniques for load balancing VPU lanes.

Recall that adding rotation to VC increases the effective combination window (CW), so RVC benefits more when the CW is small. On the other hand, LWD tackles the severe false dependences when the dependence distance is short.

Fig. 18a shows a kernel that uses 28 accumulators. Both the dependence distance and the CW size are 28. However, each non-broadcasted multiplicand is reused 28 times, so the effective CW size is around 1. This is a common situation among kernels with the embedded broadcast pattern. In the figure, we see that VC suffers from severe load imbalance and has low performance. RVC mitigates the load imbalance and performs well. VC+LWD provides less benefit than RVC because the effective CW is extremely small while the dependence distance is long. RVC+LWD performs the best, which indicates that the two optimizations are synergistic. We also see that RVC+LWD performs close to HC at medium sparsity. However, HC is slower than RVC+LWD at high sparsity, where the kernel becomes latency sensitive, and HC's 6 additional cycles harm performance.

Fig. 18b shows a kernel that uses 21 accumulators. The dependence distance is 21. Each non-broadcasted multiplicand is reused 7 times, so the effective CW size is approximately 3. For this kernel, VC+LWD is more beneficial than RVC. This is because, compared with the other kernel, the effective CW is larger while the dependence distance is shorter. Moreover, HC is less effective, since the shorter dependence distance makes the kernel more sensitive to HC's additional latency.

Overall, combining the RVC and LWD optimizations gives the best performance across different kernel behaviors.

E. Mixed-Precision Technique

We now consider the impact of SAVE's optimization on mixed-precision VFMAs. The technique exploits the sparsity when only some of the MLs mapping to an AL are ineffectual. Fig. 19 shows the speedups of a mixed-precision kernel with the one-VPU SAVE, either with or without SAVE's mixed-precision (MP) optimization, over the two-VPU baseline. The experiments are at 0% BS and various NBS levels. As before, we list the percentage of pruned ResNet-50 training iterations where the NBS of the layer is ±5% of the sparsity written below the bars. We see that the mixed-precision technique improves speedups at all sparsity levels, sometimes substantially.

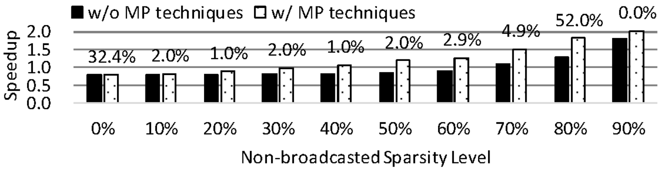


Fig. 19: SAVE speedups on the mixed-precision ResNet4_1a back-propagation of input with SAVE, either with or without SAVE's mixed-precision (MP) technique.

VIII. Related Works

Google's TPU, NVIDIA's Tensor Core, and Intel's Cooper Lake all support mixed-precision DNN training. Henry et al. [28] suggest that BF16/FP16 systolic arrays may provide 8-32x more compute potential than a FP32 vector engine. Micikevicius et al. [44] demonstrate that mixed-precision DNN workloads on Volta GPU see a 2-6x speedup over FP32.

Model pruning [23], [24], [69] sparsifies the weights. Gale et al. [17] pruned weights to 95% with low accuracy loss. However, their unstructured-pruned models can perform badly on conventional parallel hardware. Structured pruning [5], [62] is hardware-friendly for inference, but it usually prunes to a lesser degree and results in worse accuracy. It is also very difficult to exploit structured pruning during training.

PruneTrain [42] prunes entire channels and reconfigures the model to a smaller dense form during training. Our work is orthogonal to it and both techniques can work together.

Several accelerators exploit sparsity during inference. Cnvlutin [3] uses activation sparsity to skip ineffectual computations. Eyeriss [12] clock-gates hardware units when a zero is detected. It saves energy but not time. Cambricon-X [67] skips multiplications with pruned weights. EIE [22] exploits weight/activation sparsity with a compressed representation, but it is limited to matrix-vector multiplication. SCNN [47] accelerates convolutions with weight/activation sparsity. Proposals targeting CPUs and/or training are scarce.

SparCE [53] saves front-end bandwidth of light-weight CPUs by annotating skippable code blocks in software and checking for sparse inputs in hardware. It requires co-design and mainly works on scalar code. SAVE targets high-performance SIMD CPUs with spare front-end bandwidth and software transparent.

ZCOMP [2] introduces instructions to load/store compressed vectors. It synergizes with SAVE since its memory reduction is proportional to SAVE's computation reduction, and SAVE can directly use the vector loaded by ZCOMP for VFMA. Rhu et al. [51] also use a similar compression method to reduce the PCIe traffic between GPUs and the CPU.

SparseTrain [20] is a pure software approach to exploit the ReLU-induced dynamic sparsity in both training and inference. It only leverages broadcasted sparsity while SAVE exploits both broadcasted and non-broadcasted sparsity.

Control divergence induces ineffectual lanes in GPU SIMT hardware. Fung et al. [16] dynamically create warps from threads with the same next PC, and they identify the issue of aligned divergence, similar to the lane imbalance that we face. Rhu et al. [50] tackle the aligned divergence by statically permuting the thread-to-lane mapping. Their method is suitable for the coarse-grained control divergence but not the fine-grained lane imbalance discussed in this work.

Finally, this work is related to works exploring masked execution of conditional operations in vector code [57].

IX. Conclusion

We propose SAVE, the first sparsity-aware CPU vector engine. SAVE skips operations on zero values, and combines non-zero operations from multiple VFMA instructions. It is also transparent to software. SAVE includes optimizations to mitigate VPU lane imbalance, to alleviate the cache bandwidth bottleneck, and also to exploit mixed-precision computations. Using simulations of a 28-core machine running DNN workloads at realistic sparsity, we showed that SAVE accelerates inference by on average 1.37x-1.68x and end-to-end training by on average 1.28x-1.64x.

REFERENCES

[1] I. Adamski, "Solving Atari games with distributed reinforcement learning," Oct 2017. [Online]. Available: https://blog.deepsense.ai/solving-atari-games-with-distributed-reinforcement-learning/

[2] B. Akin, Z. A. Chishti, and A. R. Alameldeen, "ZCOMP: Reducing DNN Cross-Layer Memory Footprint Using Vector Extensions," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 126–138.

[3] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing," in *Proceedings of the 43th Annual International Symposium on Computer Architecture*, 2016.

[4] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, E. Elsen, J. Engel, L. Fan, C. Fougner, T. Han, A. Hannun, B. Jun, P. LeGresley, L. Lin, S. Narang, A. Ng, S. Ozair, R. Prenger, J. Raiman, S. Satheesh, D. Seetapun, S. Sengupta, Y. Wang, Z. Wang, C. Wang, B. Xiao, D. Yogatama, J. Zhan, and Z. Zhu, "Deep Speech 2: End-to-End Speech Recognition in English and Mandarin," 2015.

[5] S. Anwar, K. Hwang, and W. Sung, "Structured Pruning of Deep Convolutional Neural Networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, p. 32, 2017.

[6] ARM, "BFloat16 extensions for Armv8-A - Machine Learning IP blog - Processors - Arm Community," https://community.arm.com/developer/ip-products/processors/b/ml-ip-blog/posts/bfloat16-processing-for-neural-networks-on-armv8_2d00_a, (Accessed on 11/26/2019).

[7] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, Jun. 2017.

[8] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable Methods for 8-bit Training of Neural Networks," in *Advances in Neural Information Processing Systems*, 2018, pp. 5145–5153.

[9] J. Barr, "Natural Language Processing at Clemson University – 1.1 Million vCPUs & EC2 Spot Instances," Sept 2017. [Online]. Available: https://aws.amazon.com/blogs/aws/natural-language-processing-at-clemson-university-1-1-million-vcpus-ec2-spot-instances/

[10] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2011, pp. 52:1–52:12.

[11] K. Chellapilla, S. Puri, and P. Simard, "High Performance Convolutional Neural Networks for Document Processing," in *10th International Workshop on Frontiers in Handwriting Recognition*, Oct. 2006.

[12] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: a Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *Proceedings of the 43th Annual International Symposium on Computer Architecture*, 2016.

[13] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A Large-scale Hierarchical Image Database," in *2009 IEEE conference on computer vision and pattern recognition*. IEEE, 2009, pp. 248–255.

[14] A. Fog, "Instruction Tables: Lists of Instruction Latencies, Throughputs and Micro-operation Breakdowns for Intel, AMD and VIA CPUs," *Copenhagen University College of Engineering*, 2019.

[15] A. Fog, "The Microarchitecture of Intel, AMD and VIA CPUs: An Optimization Guide for Assembly Programmers and Compiler Makers," *Copenhagen University College of Engineering*, 2019.

[16] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 2007, pp. 407–420.

[17] T. Gale, E. Elsen, and S. Hooker, "The State of Sparsity in Deep Neural Networks," *arXiv preprint arXiv:1902.09574*, 2019.

[18] E. Georganas, S. Avancha, K. Banerjee, D. Kalamkar, G. Henry, H. Pabst, and A. Heinecke, "Anatomy of High-Performance Deep Learning Convolutions on SIMD Architectures," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 830–841.

[19] F. A. Gers, J. A. Schmidhuber, and F. A. Cummins, "Learning to Forget: Continual Prediction with LSTM," *Neural Comput.*, vol. 12, no. 10, pp. 2451–2471, Oct. 2000. [Online]. Available: http://dx.doi.org/10.1162/089976600300015015

[20] Z. Gong, H. Ji, C. Fletcher, C. Hughes, and J. Torrellas, "SparseTrain: Leveraging Dynamic Sparsity in Software for Training DNNs on General-Purpose SIMD Processors," in *Proceedings of the 29th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct 2020.

[21] M. Hamanaka, K. Taneishi, H. Iwata, J. Ye, J. Pei, J. Hou, and Y. Okuno, "CGBVS-DNN: Prediction of Compound-protein Interactions Based on Deep Learning," *Molecular informatics*, vol. 36, no. 1-2, p. 1600045, 2017.

[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 *Proceedings of the 43th Annual International Symposium on Computer Architecture*, 2016.

[23] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding," in *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, 2016.

[24] 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.

[25] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 620–629.

[26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[27] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "ExTensor: An Accelerator for Sparse Tensor Algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 319–333.

[28] G. Henry, P. T. P. Tang, and A. Heinecke, "Leveraging the bfloat16 Artificial Intelligence Datatype For Higher-Precision Computations," *arXiv preprint arXiv:1904.06376*, 2019.

[29] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[30] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely Connected Convolutional Networks," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2261–2269, 2017.

[31] Intel, "Compression of Google Neural Machine Translation Model," https://github.com/NervanaSystems/nlp-architect/tree/master/examples/sparse_gnmt.

[32] Intel, "Deep Neural Network Library (DNNL)," https://github.com/intel/mkl-dnn.

[33] Intel, "Intel Software Development Tools Optimize Deep Learning Performance for Healthcare Imaging." [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/software-development-tools-optimize-deep-learning-performance.pdf

[34] Intel, "Intel Architecture Instruction Set Extensions and Future Features Programming Reference," May 2019.

[35] Intel, "Intel Xeon Processor Scalable Family Specification Update," Nov 2019.

[36] Intel, "Intel Architecture Instruction Set Extensions Programming Reference," Jun 2020. [Online]. Available: https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html

[37] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *arXiv preprint arXiv:1502.03167*, 2015.

[38] H. Ji, L. Song, L. Jiang, H. H. Li, and Y. Chen, "ReCom: An Efficient Resistive Accelerator for Compressed Deep Neural Networks," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 237–240.

[39] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen *et al.*, "A Study of BFLOAT16 for Deep Learning Training," *arXiv preprint arXiv:1905.12322*, 2019.

[40] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, ser. NIPS, 2012.

[41] C. Louizos, M. Welling, and D. P. Kingma, "Learning Sparse Neural Networks through L0 Regularization," *arXiv preprint arXiv:1712.01312*, 2017.

[42] S. Lym, E. Choukse, S. Zangeneh, W. Wen, S. Sanghavi, and M. Erez, "PruneTrain: Fast Neural Network Training by Dynamic Sparse Model Reconfiguration," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2019, p. 36.

[43] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier Nonlinearities Improve Neural Network Acoustic Models," in *Proceedings of the 30th International Conference on Machine Learning (ICML)*, vol. 30, no. 1, 2013, p. 3.

[44] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, "Mixed Precision Training," *arXiv preprint arXiv:1710.03740*, 2017.

[45] D. Molchanov, A. Ashukha, and D. Vetrov, "Variational Dropout Sparsifies Deep Neural Networks," in *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 2017, pp. 2498–2507.

[46] V. Nair and G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," in *Proceedings of the 27th international conference on machine learning (ICML)*, 2010, pp. 807–814.

[47] 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 the 44th Annual International Symposium on Computer Architecture*, 2017.

[48] J. Park, S. Li, W. Wen, P. T. P. Tang, H. Li, Y. Chen, and P. Dubey, "Faster CNNs with Direct Sparse Convolutions and Guided Pruning," in *International Conference on Learning Representations (ICLR)*, 2016.

[49] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," *arXiv preprint arXiv:1511.06434*, 2015.

[50] M. Rhu and M. Erez, "Maximizing SIMD Resource Utilization in GPGPUs with SIMD Lane Permutation," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 356–367.

[51] 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 *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 78–91.

[52] A. Rodriguez, W. Li, J. Dai, F. Zhang, J. Gong, and C. Yu, "Intel Processors for Deep Learning Training," Nov 2017. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-processors-for-deep-learning-training.html

[53] S. Sen, S. Jain, S. Venkataramani, and A. Raghunathan, "SparCE: Sparsity Aware General-Purpose Core Extensions to Accelerate Deep Neural Networks," *IEEE Transactions on Computers*, vol. 68, no. 6, pp. 912–925, 2018.

[54] J. P. Shen and M. H. Lipasti, *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[55] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the Game of Go with Deep Neural Networks and Tree Search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan 2016.

[56] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[57] J. E. Smith, G. Faanes, and R. Sugumar, "Vector Instruction Set Support for Conditional Operations," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 260–269, 2000.

[58] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[59] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The ARM Scalable Vector Extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, Mar. 2017.

[60] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[61] D. Takahashi, "Gadi Singer interview - How Intel designs processors in the AI era," Sep 2018. [Online]. Available: https://venturebeat.com/2018/09/09/gadi-singer-interview-how-intel-designs-processors-in-the-ai-era/

[62] 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.

[63] wikichip, "Sunny Cove - Microarchitectures - Intel - WikiChip," https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove, (Accessed on 11/20/2019).

[64] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation," *arXiv preprint arXiv:1609.08144*, 2016.

[65] K. Yamada, W. Li, and P. Dubey, "Intel's MLPerf Results Show Robust CPU-Based Training Performance For a Range of Workloads," Jul 2020. [Online]. Available: https://www.intel.com/content/www/us/en/artificial-intelligence/posts/intels-mlperf-results.html

[66] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[67] 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 *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[68] Y. Zhang and J. Gong, "Manufacturing Package Fault Detection Using Deep Learning," Aug 2017. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/manufacturing-package-fault-detection-using-deep-learning.html

[69] M. Zhu and S. Gupta, "To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression," *arXiv preprint arXiv:1710.01878*, 2017.