EXPLOITING AND COPING WITH SPARSITY TO ACCELERATE DNNS ON CPUS

BY

ZHANGXIAOWEN GONG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Doctoral Committee:

Professor Josep Torrellas, Chair
Assistant Professor Christopher W. Fletcher
Dr. Christopher J. Hughes, Intel Labs
Professor Wen-mei Hwu
Professor David Padua

ABSTRACT

Deep Neural Networks (DNNs) have become ubiquitous, achieving state-of-the-art results across a wide range of tasks. While GPUs and domain specific accelerators are emerging, general-purpose CPUs hold a firm position in the DNN market due to their high flexibility, high availability, high memory capacity, and low latency.

Various working sets in DNN workloads can be sparse, i.e., contain zeros. Depending on the source of the sparsity, the level of the sparsity varies. First, when the level is low enough, traditional sparse algorithms are not competitive against dense algorithms. In such cases, the common practice is to apply dense algorithms on uncompressed sparse inputs. However, this implies that a fraction of the computations are ineffectual because they operate on zero-valued inputs. Second, when the level is high, one may apply traditional sparse algorithms on compressed sparse inputs. Although such approach does not induce ineffectual computations, the indirection in a compressed format often causes irregular memory accesses, hampering the performance. This thesis studies how to improve DNN training and inference performance on CPUs by both discovering work-skipping opportunity in the first case and coping with the irregularity in the second case.

To tackle the first case, this thesis proposes both a pure software approach and a software-transparent hardware approach. The software approach is called *SparseTrain*. It leverages the moderately sparse activations in Convolutional Neural Networks (CNNs) to speed up their training and inference. Such sparsity changes dynamically and is unstructured, i.e. it has no discernible patterns. *SparseTrain* detects the zeros inside a dense representation and dynamically skips over useless computations at run-time.

The hardware approach is called the Sparsity Aware Vector Engine (SAVE). SAVE exploits the unstructured sparsity in both the activations and the weights. Similar to *SparseTrain*, SAVE also dynamically detects zeros in a dense representation and then skips ineffectual work. SAVE augments a CPU's vector processing pipeline. It assembles denser vector operands by combining effectual vector lanes from multiple vector instructions that contain ineffectual lanes. SAVE is general purpose. It accelerates any vector workload that has zeros in the inputs. Nonetheless, it contains optimizations targeting matrix multiplication based DNN models. Both *SparseTrain* and SAVE accelerate DNN training and inference on CPUs significantly.

For the second case, this thesis focuses on a type of DNN that is severely impacted by the irregularity from sparsity — Graph Neural Networks (GNNs). GNNs take graphs

as the input, and graphs often contain highly sparse connections. This thesis proposes software optimizations that (i) overlap the irregular memory accesses with the compute, (ii) compress and decompress the features dynamically, and (iii) improve the temporal reuse of the features. The optimized implementation significantly outperforms a state-of-the-art GNN implementation. In addition, this thesis discusses the idea of offloading a GNN's irregular memory access phase to an augmented Direct Memory Access (DMA) engine, as a future work.

To my family and friends.

ACKNOWLEDGMENTS

I would like to express my gratitude to various people who helped and supported me along my journey that leads to this dissertation. First and foremost, I would like to thank my advisor, Professor Josep Torrellas, for his invaluable guidance. Josep always patiently listened to my crude ideas and helped shape them into tangible research directions. He also taught me a high standard of technical writing and spent extensive time to perfect my papers. Beyond research, Josep is the most polite and gentle person I have ever met. He cared for the well-being of all his students and would not hesitate to help. I will always remember that once I was sick and was not responsive to his email. He was so worried about me that he asked whether anyone in our research group knew where I lived so that one could check on me. Another time, when Josep learnt that I had an irregular sleep schedule, he spoke from experience that I should sleep enough while still being young, or I might regret later. I could not have asked for a better advisor.

Besides my advisor, I am also grateful to the rest of my committee. Professor Christopher Fletcher is a key collaborator in my thesis work. Being a junior faculty member, Chris was always energetic and full of exciting ideas. I was inexperienced in deep learning while searching for my thesis topic. His expertise in both deep learning and computer architecture helped kick start my deep learning projects, and subsequently helped me build my thesis around them.

Dr. Christopher Hughes, also a key collaborator, provided me priceless insights on the current landscape and the future trend in the deep learning hardware industry. These insights, along with his advice on realistic hardware design, helped bring my research close to the industry instead of staying in an ivory tower. In addition, his knowledge on the Intel instruction set and micro-architectures greatly helped me understand the behaviors of my studied workloads so that I was able to optimize my designs to the extreme. Last but not least, he kindly offered me an internship at Intel Labs that opened my eyes to the industrial research environment, which I enjoyed very much.

I collaborated with Professor David Padua on projects before I started my thesis work. Working with him allowed me to accumulate valuable experience in compiler optimization and vectorization. Later in my thesis work, the experience facilitated my design of highly efficient vectorized algorithms. Working with David was always enjoyable. Our meetings were often filled with his signature laughter that echoed across the hall way.

When I was an undergraduate student, I took Professor Wen-mei Hwu's GPU parallel

programming course, which sparked my interest in parallel programming and parallel architecture. Later I attended his talk, where he shared his life story, and in particular, how he decided to pursue a Ph.D. The talk inspired me to work towards my own Ph.D. I am grateful and honored to have all of my committee members.

During my road to the Ph.D., I had the honor to work with many people, including Professor Alexandru Nicolau, Professor Alexander Veidenbaum, Professor David Kuck, Professor Gerald DeJong, Dr. Sara Baghsorkhi, Dr. David Wong, Dr. Zehra Sura, Dr. Saeed Maleki, Dr. Zhi Chen, Dr. Neftali Watkinson, and Justin Szaday. Before that, during my undergraduate study, Professor Steve Lumetta first taught me parallel programming and then led me into the research world. Professor Lippold Haken and Professor Zuofu Cheng provided me an excellent environment to get my hands dirty on embedded hardware. I am grateful to all of them.

The i-acoma research group is filled with brilliant minds. I would like to thank all the members in the group for accompanying me over the years. In particular, I want to thank Houxiang Ji and Yao Yao for helping me with my experiments and co-authoring my papers. I also want to thank Mengjia Yan, whom I have built a firm friendship with, for providing numerous advice on how I may approach my research.

I came to the United States 13 years ago as a high school student, and I have spent 11 years in Champaign-Urbana, including both my undergraduate and graduate studies. This would not be possible without having so many great friends. I am lucky to have Xiangyu Chen, Kailing Liao, Jie Lv, Ou Ou, Shuyan Wang, Yixiao Wei, Xiaoli Yan, Jiehan Yao, Yangyang Yu, Hanyue Zheng, Yijie Zhou, and many more. While I was away from my family in China, my distant uncles Sheng Long Chang, Sheng Chong Chang, and their families in Los Angeles offered me great hospitality. My "American mother" Amal Bernal treated me like her own son. I am forever indebted to them.

Finally, and most importantly, I am grateful to my family. I want to thank my fiancée Yue Liu for her loving support during the last year of my graduate study, my parents Jie Zhang and Guohua Gong for their unparalleled love, and my late maternal grandparents Zhang Chen and Yishi Zhang for giving me the best childhood that I could dream of. My parents have always been my firmest supporters. They backed every life decision that I made, regardless of how unconventional it might be. Unfortunately, my father was diagnosed with cancer two years ago. Although right now I am unable to be by his side fighting the cancer, I would like to dedicate this dissertation to his health.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 DEEP NEURAL NETWORKS (DNNS) ON CPUS

Deep Neural Networks (DNNs) have attained state-of-the-art results in a variety of tasks such as image recognition [1], speech recognition [2], scene generation [3], and game playing [4]. General-purpose CPUs are widely deployed in datacenter, client, and edge devices. Therefore, utilizing CPUs for DNN workloads lowers the Total Cost of Ownership (TCO) for the DNN market [5, 6, 7, 8]. 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 [9]. For DNN training, GPUs and accelerators provide higher raw compute power. However, the high memory capacity on CPU platforms (e.g., up to 4.5TB per socket with the 3rd-gen Intel Xeon Scalable processors) makes training with large datasets and/or models easier [7]. Also, the high availability of datacenter CPUs encourages companies to distributedly train DNNs on CPUs during off-peak periods [6]. For example, Facebook trains their *Sigma* and *Facer* frameworks either entirely or partially on CPUs [9]. Other examples of training on CPUs include Intel's assembly and test factory [10], deepsense.ai's reinforcement learning [11], Kyoto University's drug design [12], Clemson University's natural language processing [13], GE Healthcare's medical imaging [14], and more [8]. Further, CPU makers have recently introduced features to accelerate training, such as BFloat-16 [7] and Intel Advanced Matrix Extensions [15]. Therefore, accelerating both DNN training and inference on CPUs is an important yet undervalued area.

1.2 SPARSITY IN DNN DATA STRUCTURES

Sparsity stands for the zeros inside a data structure such as a vector or a matrix. Various DNN data structures can be sparse due to different reasons.

1.2.1 Sparse Activations

Activations, a.k.a features, are the outputs of a DNN layer. In a DNN, each output value is usually passed through an activation function to introduce non-linearity. One ubiquitous activation function is the Rectified Linear Unit (ReLU). It has the form:

$$f(x) = max(0, x) \tag{1.1}$$

and its derivative is:

$$f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{1.2}$$

Note that, the derivative at $x = 0$ is undefined but usually set to 0.

By definition, ReLU and its derivative produce 50% sparsity when the distribution of x is centered at 0. When ReLU-activated layers are cascaded, the sparse output activations of a layer become the inputs to the next layer. During training, the sparsity from ReLU often begins at approximately 50% but increases rapidly in the first several epoches, and then slowly decreases. Also, later conv layers generally have higher sparsity then earlier layers [16]. Figure 1.1 presents the sparsity of each ReLU's output during end-to-end training of VGG16. In this example, the average sparsity of each layer typically ranges from 50% to 90%.

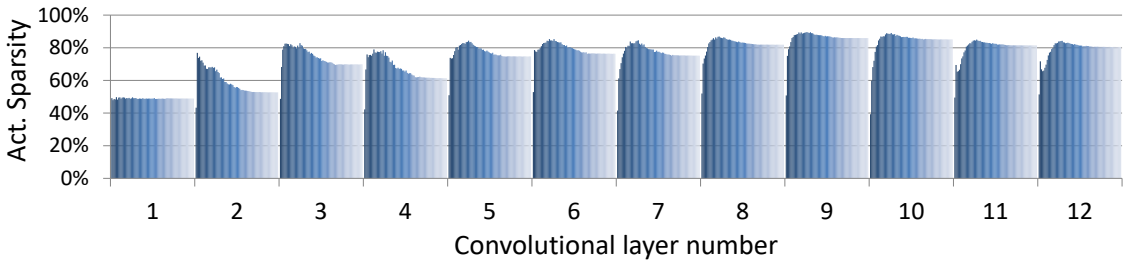


Figure 1.1: Activation sparsity during the training of VGG16. Each x-axis segment shows the outputs from a ReLU-activated layer. Within a segment, from left to right shows the sparsity from the first epoch to the last.

Besides ReLU, *dropout* can also sparsify the activations. Dropout is a widely adopted technique to reduce overfitting [17]. During training, a predefined fraction, often 50%, of the hidden features, i.e., the output activations of the non-output layers, are randomly selected and set to zero.

Both ReLU and dropout produce a moderate level of sparsity, and the sparsity pattern has no discernible structure. Moreover, the sparsity constantly changes over time. Therefore, we call it unstructured dynamic sparsity.

1.2.2 Sparse Weights

The weights in DNNs can be sparse thanks to weight pruning, which is a popular technique to reduce the size of the DNN models to lower the cost of inference [18, 19, 20]. During training, the weights are initially dense. As the training progresses, a pruning algorithm gradually fixes a portion of the weights to zero according to a certain criteria until the sparsity in the weights reaches a predetermined level. The training often continues for a

while to fine tune the pruned model in order to achieve better network accuracy. Figure 1.2 shows an example schedule to prune the weights in ResNet-50 during end-to-end training. Initially, the weights are dense. After the pruning starts round epoch 33, the sparsity level gradually rises until around epoch 57. The rest of the training epochs fine-tune the pruned model.

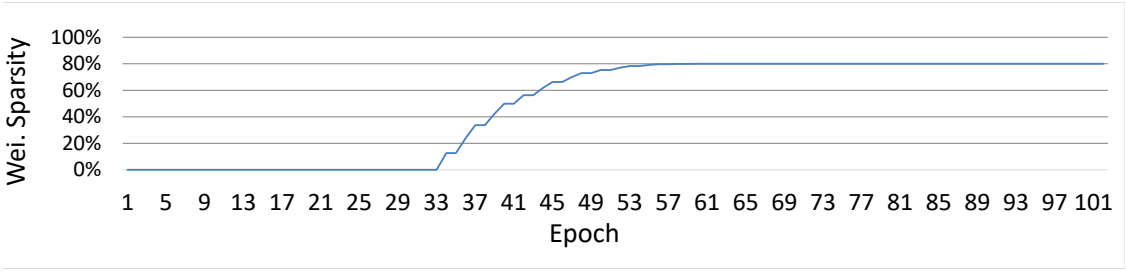


Figure 1.2: An example schedule to prune the weights in ResNet-50 during training.

Arguably the simplest yet most effective pruning method is the magnitude-based pruning [18], where a percentage of the weights with the lowest absolute values are pruned after each pruning iteration. It has been shown that this method can prune the weights to 95% sparse with low network accuracy loss [21].

The magnitude-based method is easy to implement and results in both high compression rate and low accuracy loss. However, it creates unstructured sparsity, which is unfriendly to conventional parallel hardware such as Single Instruction Multiple Data (SIMD) CPUs and GPUs. To align the sparsity pattern with the underlying hardware, structured pruning emerges [22, 23, 24]. These methods typically prune a continuous chunk of weights at a time so that parallel hardware may skip computing with the block. However, they often lower the accuracy of the model more than unstructured pruning does at the same pruning rate.

In inference, the weights do not change, so one may store the highly pruned weights in a compressed format without worrying about frequent compression. However, during training, weight sparsity evolves over time, so the common practice is to store the sparse weights in an uncompressed format and use masks to mark the pruned locations.

1.2.3 Sparse Graph Connections

Both weights and activations are common working sets shared by all types of DNNs. On the other hand, some DNN models operate on additional working sets. Graph Neural Networks (GNNs) are some of them. GNNs have gathered much attention recently due to their ability to process non-Euclidean data, which other types of DNNs struggle with [25, 26]. The input data to GNNs are graphs, so GNN workloads share similarities with traditional

graph algorithms. In particularly, the input graphs often have highly sparse connections, i.e., each vertex is connected to a small subset of all vertices. When expressing the connections in an adjacency matrix, $\mathbf{A}$, the matrix is usually over 99% sparse. For example, the Amazon product co-purchasing network graph [27] has 2.4M vertices and 62M undirected edges, suggesting that $\mathbf{A}$ is 99.998% sparse. When uncompressed, the footprint of A is $O(|\mathcal{V}|^2)$, where $\mathcal{V}$ is the set of all vertices. On the contrary, the space requirement of a typical compressed format such as Compressed Sparse Row (CSR) is only approximately $O(|\mathcal{E}|+|\mathcal{V}|)$, where $\mathcal{E}$ is the set of all edges. In addition, the graph structure is fixed, so one only needs to compress $\mathbf{A}$ once. Hence, the adjacency matrix is often encoded in a compressed format.

1.3 CHALLENGES FROM SPARSITY

Sparsity poses challenges to both the compute and the memory performance of DNN workloads. We first look at the compute performance. When one or more operands are zero, some computations become *ineffectual*, meaning that they do not affect the end results. Among them, the Multiply-Accumulate (MAC) operation is arguably the most important one since it is employed in many high-performance software, including DNN kernels. A MAC has the form $c = c + a \cdot b$, when either a or b is zero, c does not change. Therefore, such a MAC is ineffectual. When the input working sets are sparse, many computations become ineffectual.

Naturally, we do not want to perform these ineffectual computations. However, depending on the level of sparsity in the inputs, efficiently skipping ineffectual work can be challenging. On the software side, traditional sparse methods such as sparse-dense matrix multiplication (SpMM) operate on compressed sparse inputs. Well-adopted compressed formats such as CSR introduce storage overhead that increases as the number of non-zero values rises. When the sparsity level is low enough, the compressed data can take more space than the original uncompressed data. This is often the case in DNNs, where the working sets can sometimes contain less than 50% zeros. This can happen, for example, from the use of ReLU or dropout.

Besides the storage overhead, there is another reason that renders the compressed format undesired. As discussed earlier, in DNN data structures, the pattern of the zeros and the level of the sparsity often change dynamically. As a result, if a compressed format is employed, one needs to frequently perform new compression, which is a significant time overhead.

Finally, modern processors are optimized for regular and dense compute. Conventional wisdom tells us that traditional sparse methods require the inputs to be highly sparse (e.g. over 95%) in order to outperform optimized dense methods. Consequently, in many cases, using traditional sparse methods in DNNs is ineffective. Instead, people usually ignore the

sparsity and opt to use dense methods. Many DNN workloads are compute intensive because their major computations are matrix multiplication and/or convolution. Therefore, although using dense methods is a reasonable choice, performing the ineffectual computations hinders the performance.

We then look at the memory performance. There exists scenarios in DNNs that fit the requirements for using compressed formats and sparse methods. GNNs are some of the more pronounced ones. The input graphs to GNNs often contain compressed sparse adjacency matrices. As a result, applying sparse methods avoids performing ineffectual computations. However, the extra indirection in the compressed formats as well as the irregular memory accesses in the sparse method can cause the computation to stray away from the peak performance of the system.

On the hardware side, to tackle these challenges, the community has proposed domain-specific accelerators that skip ineffectual computations in compute-intensive models [28, 29, 30, 31] or more efficiently execute the memory-intensive GNN workloads [32, 33, 34]. Nevertheless, few works have been done to add hardware support for exploiting DNN sparsity on general-purpose CPUs. Proposing hardware supports on CPUs is challenging because the hardware addition needs to be general-purpose enough to fit the design philosophy of CPUs.

1.4 THESIS CONTRIBUTIONS

Given the challenges discussed above, this thesis proposes both software and hardware methods that exploit the sparsity in DNN data structures to accelerate DNN workloads on CPUs. This thesis makes the following contributions.

***SparseTrain*: Leveraging Dynamic Sparsity in Software for Training DNNs on General-Purpose SIMD Processors** *SparseTrain* is the first software-only CPU algorithm that leverages the dynamic sparsity in the activations [35]. It dynamically checks for zero in the inputs in a regular (uncompressed) representation and skips a batch of ineffectual computations upon detecting a zero. It employs various optimizations to mitigate the overhead from exploiting sparsity. *SparseTrain* outperforms a highly-optimized dense implementation when the inputs are on average 20% sparse. At realistic sparsity levels, *SparseTrain* can speedup both training and inference significantly.

SAVE: Sparsity-Aware Vector Engine for Accelerating DNN Training and Inference on CPUs SAVE is the first sparsity aware vector engine for CPUs [36]. SAVE exploits the unstructured sparsity in both the activations and the weights. it combines

effectual vector lanes from multiple ready instructions to assemble a set of temporary vector operands. It then issues a SIMD computation with the temporary operands. SAVE is enhanced with techniques that increase the rate of compaction, and it only adds limited additional hardware to the existing CPU vector pipeline. SAVE is transparent to software and can accelerate legacy code. It is general-purpose and is not limited to accelerating DNN workloads. Instead, it benefits any vector workload with zeros in its inputs. When the inputs are dense, SAVE does not introduce perceptible time overhead. At realistic sparsity levels, SAVE can notably accelerate both training and inference.

Optimizing Graph Neural Networks on CPUs: Reducing Memory Bandwidth Needs In this contribution, we devise software optimizations to speed up GNN workloads on CPUs. We first characterize GNN workloads on CPUs and identify that DRAM bandwidth is a major bottleneck. We then propose optimizations to relieve the DRAM bandwidth pressure. The optimizations include both mitigating the irregularity from the sparse graph connections and reducing unnecessary memory accesses to the sparse features. Finally, we validate our approach on popular GNN models with medium to large scale graphs, on a server CPU. Our implementation outperforms a state-of-the-art GNN layer implementation significantly for both training and inference.

1.5 THESIS ORGANIZATION

The thesis is organized as follows. Chapter 2 and 3 tackle the compute performance issue. Chapter 2 presents the software approach: *SparseTrain*, and Chapter 3 presents the hardware approach: SAVE. Chapter 4 and 5 address the memory performance issue. Chapter 4 describes our GNN software optimizations, and Chapter 5 discusses a future work that aims to offload a GNN's irregular memory access phase to an augmented Direct Memory Access (DMA) engine. Chapter 6 concludes the thesis. In addition to the above contributions, this thesis also includes my other works that assess compilers' ability to optimize loop nests, in Appendix A.

CHAPTER 2: SPARSETRAIN: LEVERAGING DYNAMIC SPARSITY IN SOFTWARE FOR TRAINING DNNS ON GENERAL-PURPOSE SIMD PROCESSORS

2.1 INTRODUCTION

As discussed in Section 1.3, an effective approach to accelerating compute-intensive DNN workloads is to remove useless computations on zero values in the data. Prior efforts spanning hardware to software and algorithms have exploited sparsity to eliminate computation or data transfers at different points in DNN computations. Most of these efforts, though, require hardware changes [16, 28, 29, 30, 31, 37, 38] and/or apply only to *inference* [24, 28, 29, 30, 31, 37, 39, 40, 41]. This is not ideal, since most of real-world DNN computations are performed on conventional CPUs and GPUs, and significant time goes into *training*.

In this chapter, we present *SparseTrain*, a *software only* effort that addresses these shortcomings and speeds up both DNN training and inference on CPUs. *SparseTrain* skips the ineffectual computations induced by sparsity, on unmodified general-purpose CPUs. It targets the sparsity in the activations from ReLU and/or dropout, which is moderate (typically 40-90% sparse), unstructured (no discernible patterns), and dynamic (changes with each input). *SparseTrain* has the following properties:

- It checks for zeros dynamically in runtime and skips a batch of ineffectual computations upon detecting each zero. It applies optimizations to mitigate the branch mispredictions from the dynamic predications.
- It loads/stores data in a regular (dense) representation so that there is no overhead associated with (de)compressing data. Also, it avoids the irregular memory accesses from using a compressed (sparse) representation.
- It generates parallelized and vectorized kernels with a just-in-time (JIT) assembler. For each DNN layer, it specializes the code only according to the layer shape; therefore, it only generates the kernel once for a given layer so that the overhead is negligible.
- It is applicable to both inference and all phases of DNN training, namely the forward propagation (same as inference), the backward input propagation, and the backward weight propagation.

The amount of computation skipped due to a zero input depends on the number of reuse of the input. High reuse helps amortize the overheads incurred while detecting and exploiting

sparsity. Among different types of DNNs, CNNs have the highest reuse of their neurons. Therefore, while the approach is generally applicable to any DNN employing ReLU and/or dropout, we focus on CNNs in this chapter.

Our experiments on a 6-core Intel Skylake-X server show that *SparseTrain* is very effective. In end-to-end training of VGG16 [42], ResNet-34, and ResNet-50 [43] with the ImageNet dataset [44], *SparseTrain* outperforms a highly-optimized direct convolution on the non-initial convolutional layers by 2.19x, 1.37x, and 1.31x, respectively. *SparseTrain* also benefits inference. It accelerates the non-initial convolutional layers of the aforementioned models by 1.88x, 1.64x, and 1.44x respectively.

2.2 BACKGROUND

2.2.1 Training Convolutional Neural Networks

A CNN is a type of DNN that is effective for analyzing images. Within a CNN, the convolutional layers are the most time consuming components; thus, reducing the amount of computation in them can greatly boost performance. In the following discussion, we use the symbols listed in Table 2.1.

Table 2.1: List of the symbols and their dimensions & iterators.

	Description	**Iterator**		**Description**	**Dimension**	**Iterator**
N	minibatch size	i	D	input tensor	$NCWH$	i, c, x, y
C	input channels	c	Y	output tensor	$NKW'H'$	i, k, x', y'
K	output channels	k	G	weight tensor	$KCRS$	k, c, u, v
W	input width	x	L	loss function		
H	input height	y	V	vector length		
R	filter width	u	T	# of skippable ops		
S	filter height	v	M	minibatch tile size		
O	horizontal stride		Q	output channel tile		
P	vertical stride			size		

The convolution on a minibatch of N images with C channels and size $H \times W$ correlates a set of K filters with C channels and size $S \times R$ on the images, producing a minibatch of N images with K channels and size $H/P \times W/O$, where P and O are the strides of the two dimensions, respectively. We denote filter elements as $G_{k,c,u,v}$ and image elements as $D_{i,c,x,y}$. The forward convolution for output $Y_{i,k,x',y'}$ is:

$$Y_{i,k,x',y'} = \sum_{c=0}^{C-1} \sum_{u=0}^{R-1} \sum_{v=0}^{S-1} D_{i,c,x' \times O+u,y' \times P+v} \times G_{k,c,u,v} \tag{2.1}$$

In the backward propagation of a convolutional layer, the gradient of the loss function L with respect to the weights G is calculated by applying the chain rule:

$$\frac{\partial L}{\partial G} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial G} \tag{2.2}$$

We need $\partial L/\partial Y$ from the next layer, and compute $\partial L/\partial D$ for the previous layer if needed. $\partial L/\partial D$ is a convolution of $\partial L/\partial Y$ with the layer's filters transposed. The gradient with respect to the weights is a convolution of D with $\partial L/\partial Y$, producing $S \times R$ outputs for each input/output channel combination.

2.2.2 Baseline Platform

We consider a shared-memory server comprising general-purpose processors with multiple cores and SIMD support. While we tune and evaluate on a specific platform described in Section 2.4, our approach is applicable to most modern shared-memory nodes with processors supporting SIMD. Further, our approach is fully compatible with multi-node implementations; it will simply accelerate the work done on each node.

In this chapter, We study a system with Intel Skylake cores. In each cycle, each core can execute two AVX-512 arithmetic instructions (e.g., vector fused multiply-add, or VFMA), read two cache lines (64B) and write one cache line from/to the L1 data cache, and retire four instructions. Each core has 32 vector registers, a 32KB L1 data cache, a 1MB L2 cache and a 1.375MB non-inclusive shared L3 cache.

We implement our work as new convolution kernels in *MKL-DNN* [45], a highly tuned DNN library. We specialize the kernels according to the size of the convolution and the hardware parameters via JIT compilation. Prior works also demonstrated that JIT-ing achieves higher performance than statically-tuned BLAS-calls for convolution [46, 47]. Because for a given convolutional layer, we only JIT the kernels once during the whole training process, the kernel generation overhead is virtually non-existent. Being low-level software, our implementation can be incorporated to DNN frameworks like *TensorFlow* [48] or *PyTorch* [49].

2.3 SPARSETRAIN ALGORITHM

2.3.1 Naïve Forward Propagation

SparseTrain is based on direct convolution. Algorithm 2.1 describes a naïve vectorized approach that skips computation in the forward propagation upon detecting a zero input.

Line 2 and Line 5 represent collapsed loop nests. For simplicity, the algorithm assumes unit stride, but can be easily expanded for strided convolution. In the rest of the chapter, we assume unit stride unless otherwise specified. The sparse algorithm for the backward input propagation is similar to that of the forward propagation, and we will talk about the backward weight propagation separately.

The main idea is as follows. Since an input element is reused $R \times S \times K$ times, by making the input stationary in the computation loop nest, we may skip at most $R \times S \times K$ operations when we detect a zero. We parallelize along the minibatch dimension (N) and vectorize along the output channel dimension (K). The statement in Line 6 represents a vector fused multiply-add (VFMA) instruction with SIMD width V. A VFMA computes a vector MAC operation. When we detect a zero in Line 3, we skip all of the following $R \times S \times K/V$ ineffectual VFMAs. We denote the number of skippable VFMAs per check as T, which is usually large because K is often on the order of hundreds.

Algorithm 2.1: Naïve Vectorized Sparse FWD.

```
   input  : input D, filters G
   output : output Y
1  for i = 0 to N − 1 in parallel do
2      for c = 0, y = 0, x = 0 to C − 1, H − 1, W − 1 do
3          if D_{i,c,x,y} ≠ 0 then
4              for k = 0 to K − V step V do
5                  for u = 0, v = 0 to R − 1, S − 1 do
6                      Y_{i,[k:k+V−1],x−u,y−v} = Y_{i,[k:k+V−1],x−u,y−v} + D_{i,c,x,y} × G_{[k:k+V−1],c,u,v}
```

The naïve algorithm has several downsides. First, it naturally has input parallelism: it compares each D element to zero and then updates multiple Y elements. Input parallelization requires atomic updates of Y, which drastically reduces performance. Output parallelization is generally faster. The simplest such approach is to let each core work on different images in the minibatch. However, common practice on training on CPU clusters is to assign, to each multicore, only a small minibatch. As a result, it is likely that different cores will get a different number of images, resulting in load imbalance.

The second downside is that a CPU has a limited amount of logical vector registers; this is 32 in the CPU we target. If $T = R \times S \times K/V$ is greater than the number of logical registers, we must spill registers during computation, inducing overhead. Therefore, we want to confine T within the register budget.

Finally, D has an unpredictable sparsity pattern, triggering frequent branch mispredictions in the zero-checking. Limiting T to the register budget ($\sim$32) reduces our chance to amortize

the misprediction penalty.

2.3.2 Optimized Forward Propagation

To improve the naïve forward propagation, we introduce the following optimizations whose high-level ideas are presented in Algorithm 2.2.

Algorithm 2.2: Parallel Vectorized Sparse FWD.

input : input D, filters G
output : output Y

```
1  for i = 0 to N − M step M in parallel do
2    for y = 0 to H − 1 in parallel do
3      for v = 0 to S − 1 do
4        for k = 0 to K − Q step Q in parallel do
5          for c = 0 to C − V step V do
6            for i' = i to i + M − 1 in parallel do
7              for x = 0 to W − 1 do
8                m_[0:V−1] = [d ≠ 0 for d in D_i,[c:c+V−1],x,y+v]
9                for c' = 0 to V − 1 do
10                 if m_c' is true then
11                   for k' = k to k + Q − V step V do
12                     for u = 0 to R − 1 do
13                       Y_i',[k':k'+V−1],x−u,y = Y_i',[k':k'+V−1],x−u,y +
                           D_i',c+c',x,y+v × G_[k':k'+V−1],c+c',u,v
```

Vectorized Zero-Checking

The naïve algorithm compares D elements to zero one at a time. To improve it, we vectorize this check along the input channel dimension (C). Specifically, Line 8 in Algorithm 2.2 does a vector comparison to zero to generate a vector boolean mask $m_{[0:V-1]}$; each mask bit is set if the corresponding input element is not zero. We then use the mask to determine whether to skip computation.

Increasing Output Parallelism

In a convolution, a D element affects a set of spatially-grouped Y elements. Similarly, a Y element is calculated from a limited set of spatially-grouped D elements. This allows us

to increase output parallelism by reducing T.

We parallelize at an output row granularity. When a core works on an output row, it processes the D elements from S corresponding input rows, one row at a time. This approach lowers T from $R \times S \times K/V$ to $R \times K/V$. Moreover, if $R \times K/V$ is still larger than the number of ISA registers, we further reduce T to avoid register spilling. We accomplish this by tiling the output channel dimension (K) and decrease T to $R \times Q/V$, where Q is a factor of K and a multiple of V. We will discuss how we choose Q in the next section. We can process the same output row at different output channel tiles in parallel. With $T = R \times Q/V$, the number of parallel tasks rises from N in the naïve algorithm to $N \times H \times K/Q$.

Since an input row corresponds to S output rows, multiple cores may read a given input row. In a shared memory system, such reuse may be captured in a shared cache.

Efficient Vector Register Usage

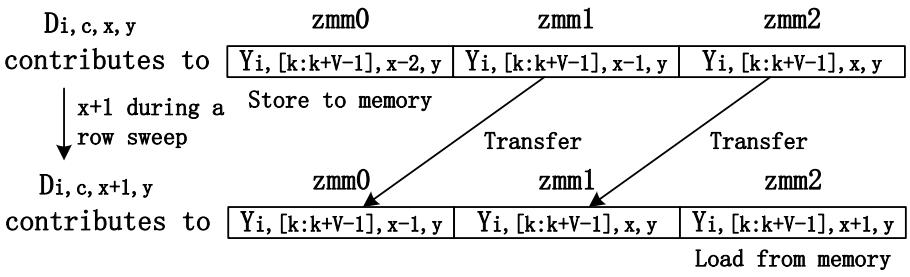
A VFMA has three operands: one accumulator vector and two multiplicand vectors. In the target ISA, one multiplicand vector can be a memory operand. In modern Intel and AMD microarchitectures such as Skylake and Zen, the L1 read bandwidth matches the VFMA throughput (2 per cycle per core) [50]; thus, utilizing the memory operand does not slow down the computation.

When we translate Line 13 of Algorithm 2.2 to a VFMA instruction, we use the multiplicand vector $G_{[k':k'+V-1],c+c',u,v}$ as a memory operand. We broadcast $D_{i',c+c',x,y+v}$ to all lanes of a vector register and use the register as the other multiplicand vector. Note that all $T = R \times Q/V$ VFMAs in the loop from Lines 11-13 share this broadcasted D element. Finally, each VFMA needs a dedicated vector register to hold the accumulator vector $Y_{i',[k':k'+V-1],x-u,y}$. Therefore, we need $T + 1$ vector registers for a given T.

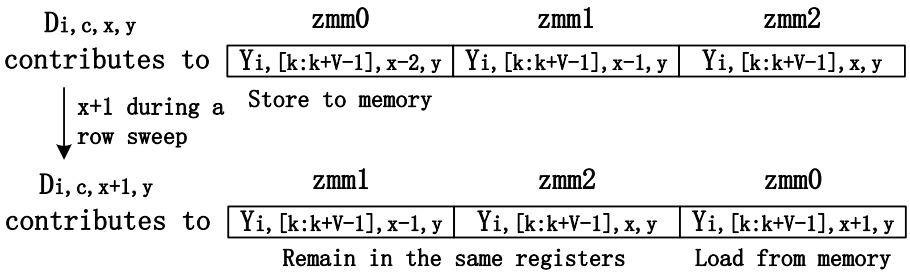
The target ISA has 32 `zmm` vector registers. Algorithm 2.2 keeps a vector of zeros for the vector compare instruction in Line 8. Hence, there are 31 vector registers available. Because we need $T + 1$ vector registers, we limit T to 30 in order not to spill the registers.

Besides avoiding register spilling, we further reduce memory operations. As shown in Lines 7-13, we scan through an input row and update the affected Y elements accordingly. We call such a scan a *Row Sweep*. Figure 2.1 illustrates examples of how we optimize both memory access and register usage during a row sweep.

Due to a convolution's spatial nature, adjacent D elements may contribute to overlapping Y elements, depending on the filter width R and the horizontal stride O. Consider the example in Figure 2.1a. When $R = 3$ and $O = 1$, $D_{i,c,x,y}$ contributes to $Y_{i,[k:k+V-1],[x-2:x],y}$. The next element $D_{i,c,x+1,y}$ contributes to $Y_{i,[k:k+V-1],[x-1:x+1],y}$. Thus, both D elements



(a) When proceeding to the next D element during a row sweep, we do not store and then reload the output Y vectors affected by both the current and the next inputs. However, a naïve implementation requires transferring data between registers.



(b) Cyclic register renaming further avoids transferring data between registers.

Figure 2.1: Examples of how *SparseTrain* minimizes both memory access and moving data between registers during a row sweep when $R = 3$ and $O = 1$.

contribute to $Y_{i,[k:k+V-1],[x-1:x],y}$. As a result, as x increments, we can keep $Y_{i,[k:k+V-1],[x-1:x],y}$ in the registers. We only need to save $Y_{i,[k:k+V-1],x-2,y}$ to memory and load $Y_{i,[k:k+V-1],x+1,y}$ from memory. Consequently, each Y vector is only read and written once during a row sweep.

However, although the shared Y vectors can stay in the registers as x advances, a naïve implementation that statically uses registers according to the spatial order of the convolution still requires transferring the Y vectors from one register to another. For example, in Figure 2.1a, `zmm[0:2]` hold the Y vectors affected by a D element in the order from left (lower index in the W dimension) to right (higher index in the W dimension). As x increments, the Y vector in `zmm[1:2]` needs to be transferred to `zmm[0:1]`. Modern microarchitectures typically eliminate such register-to-register moves at the register allocation stage to bypass executing them in the back-end [50]. Nevertheless, the move instructions still consume front-end resources.

To avoid the move instructions, we devise a software scheme that simulates register renaming. As illustrated in Figure 2.1b, we use `zmm[0:2]` to hold the Y vectors. When

working on $D_{i,c,x,y}$, `zmm0` holds $Y_{i,[k:k+V-1],x-2,y}$, `zmm1` holds $Y_{i,[k:k+V-1],x-1,y}$, and `zmm2` holds $Y_{i,[k:k+V-1],x,y}$. After moving on to $D_{i,c,x+1,y}$, `zmm0` proceeds to load $Y_{i,[k:k+V-1],x+1,y}$ while $Y_{i,[k:k+V-1],x-1,y}$ and $Y_{i,[k:k+V-1],x,y}$ are kept in their previous registers.

This scheme requires unrolling the row sweep loop, starting on Line 7. For large W, fully unrolling can lead to kernels larger than the instruction cache. Since the cyclic renaming repeats every R iterations, we instead unroll by a factor of R to limit code size.

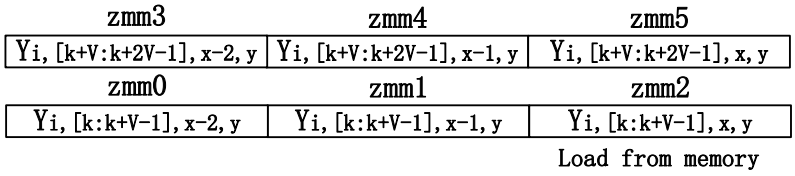
The number of registers used, how they are cyclically renamed, and the unrolling factor all depend on the parameters R and O. As a result, statically compiled code cannot implement this scheme. Hence, it is crucial to use JIT compilation.

Because R and V are fixed by the convolution configuration and the hardware, respectively, the only tunable parameter in $T = R \times Q/V$ is Q. As a result, the register budget is often underutilized. To see why, assume that we want Q to be a factor of the number of output channels K, so blocks have the same size. When $R = 5$, $V = 16$, and $K = 256$, which is a typical number of channels, a reasonable maximum value of Q is 64. As a result, $T = 20$. Recall that we have 32 vector registers in total, and we use 2 vector registers for other uses: one to hold an all-zero vector and the other to hold the broadcasted input D element. Therefore, 10 registers are unused.

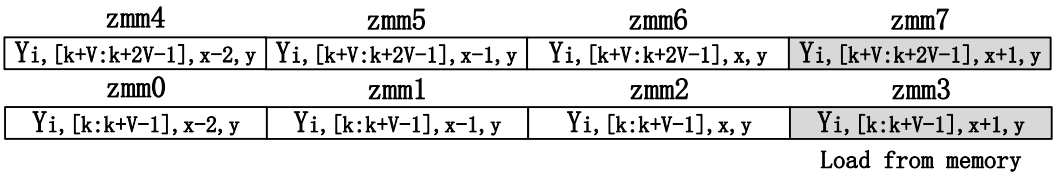
In such cases, we use the spare registers to pipeline the load of the Y vector affected by the next D element. Consider the registers in Figure 2.2 that hold Y vectors when processing $D_{i,c,x,y}$. Figure 2.2a is the case without pipelining. With $R = 3$ and $O = 1$, we need 3 registers along the W dimension. Because we vectorize along the K dimension, with $Q = 32$ and $V = 16$, we need $Q/V = 2$ registers along the K dimension. Therefore, we allocate 6 registers in total. In this case, we load $Y_{i,[k:k+V-1],x,y}$ and $Y_{i,[k+V:k+2V-1],x,y}$ from memory. $D_{i,c,x,y}$ contributes to both of them.

Figure 2.2b is the case with pipelining. Because Q/V is unchanged in the example, we also need 2 registers along the K dimension. If we have 2 spare registers, we use them to preload Y vectors along the W dimension, i.e., we preload $Y_{i,[k:k+V-1],x+1,y}$ and $Y_{i,[k+V:k+2V-1],x+1,y}$. The next $D_{i,c,x+1,y}$ contributes to them, but the current $D_{i,c,x,y}$ does not. In this way, the VFMAs depend on loads from an *earlier* iteration so that the out-of-order hardware can dispatch the VFMAs sooner. Note that, with pipelining, the unroll factor of the row sweep loop becomes $R + 1$ instead of R.

We need $(R + 1) \times Q/V$ registers as output buffers with pipelining or $R \times Q/V$ without it. Therefore, we want the number of output buffer registers to be maximized but no higher than the budget, which is 30 as discussed. At $K = 256$ and $V = 16$, the optimal values of Q for common values of the filter width R are shown in Table 2.2. The values of Q are 128 for $R = 1$ with pipelining, 128 for $R = 3$ without pipelining, and 64 for $R = 5$ with pipelining.



(a) Without pipelining, 6 registers are used.



(b) With pipelining, 8 registers are used. The pipelined loads are marked with gray background.

Figure 2.2: Example allocations of the output buffer registers that hold Y vectors affected by $D_{i,c,x,y}$ when $R = 3$, $O = 1$, $Q = 32$, and $V = 16$.

Table 2.2: Optimal value of Q for $K = 256$ and $V = 16$ at different R.

R	Q	**Pipelined?**	**# of output buffer registers**	T
1	128	Yes	$16 = (R+1)Q/V$	$8 = RQ/V$
3	128	No	$24 = RQ/V$	$24 = RQ/V$
5	64	Yes	$24 = (R+1)Q/V$	$20 = RQ/V$

For $R = 1$, we found that the alternative of $Q = 256$ without pipelining is slower. This is because when processing each D element in a row sweep, we compute $R \times Q/V$ VFMAs and load Q/V number of Y vectors from memory. Thus, the compute to load ratio is R. When $R = 1$, the ratio is so low that pipelining provides substantial benefit by hiding the load latency.

Reducing Branch Mispredictions

As discussed, the optimal T is ≤ 30 on the target CPU. Under this constraint, the zero checking and skipping method in Lines 8-13 of Algorithm 2.2 may induce so many branch mispredictions that the code actually slows down. To address the issue, we transform a series of branches to a single loop and reduce the number of branches by a factor of V. With many fewer branches, we drastically reduce the overall misprediction penalty.

Algorithm 2.3 shows the method that can replace Lines 8-13 in Algorithm 2.2. First, we compare the input vector to zeros to generate a mask (Line 1, which maps to Line 8 in

Algorithm 2.2). This is done with the `vcmpps` instruction on the target CPU. Then, we use `popcnt` (Line 2) to count the number of 1s in the mask, which represents the number of non-zero elements in the input vector. After that, the code loops this number of times as shown in Lines 3-10, where each loop iteration processes a non-zero element from the input vector.

Algorithm 2.3: Zero Checking for Branch Performance.

```
   input    : input pointer D, filter pointer G
   output   : register array Y
   constant: filter offset B
 1 m[0:V-1] = vect_cmp_neq_zero(D[0:V-1]);
 2 o = population_cnt(m[0:V-1]);
 3 for i = 0 to o − 1 do
 4     z = trailing_zero_cnt(m);
 5     D += z; G += z * B;
 6     for j = 0 to Q/V fully unrolled do
 7         for k = 0 to R fully unrolled do
 8             Y[j][k][0:V-1]+=broadcast(D[0])*G[j][k][0:V-1]
 9     m = shift_right(m, z+1);
10     D += 1; G += B;
```

In each iteration, we first count the number of trailing zeros (`z`) in the mask with the `tzcnt` instruction (Line 4). Then, we advance the input pointer by `z`, to reach the next non-zero element in the input vector. We also advance the filter pointer such that it points to the filter elements corresponding to the given non-zero input element. Finally, we do the VFMAs.

We fully unroll the loop nest in Lines 6-8. The `Y[j][k][0:V-1]` vectors shown in the loop body are actually in the output buffer registers, which are allocated though the cyclic renaming scheme discussed earlier. Finally, we shift the mask to the right by `z`+1 to reflect that we have finished processing the rightmost non-zero input element (Line 9), and also adjust the input and filter pointers accordingly (Line 10).

For readability, we omit some low-level optimizations in Algorithm 2.3. Specifically, we pipeline the vector compare instruction such that the vector mask for the next iteration is generated during the current iteration. In this way, we can overlap the compute from the current D element with the load of the next D element. We also manually schedule and pipeline the integer instructions in the loop body to minimize dependence stalls. Moreover, we use shifts and load effective address (`lea`) instructions to reduce the strength of the integer multiplications and the number of integer instructions. In the end, each loop iteration of

Lines 3-10 only contains 8 cheap integer instructions plus the VFMAs.

Memory Access Optimization

We structured both the working sets and the loop nest carefully for high memory performance. First, we set the lowest dimension of the datasets to a channel tile of size V. On the target CPU, this is the `zmm` vector register size and the cache line size. Recall that we vectorize the computation along channels. Therefore, when the channel tile is aligned to a cache line boundary, vector instructions operate efficiently on a vector of channel data.

We have 3 working sets, with different behaviors: the input D, the filters G, and the output Y. D and Y have spatial locality in a row sweep. Each row element from them is loaded/stored only once per row sweep, and adjacent elements in a row are accessed consecutively. Such a streaming pattern benefits from hardware prefetching when we assign the second lowest dimension to the row dimension. We may also strategically software-prefetch the elements of the next row to the L2 cache when the line fill buffers (LFB) are not saturated.

In contrast, G has temporal locality in a row sweep. Since we compute partial results for $W \times Q$ output elements from $W \times V$ input elements in a row sweep, we access $Q \times V \times R$ filter elements repeatedly. With the R and Q values listed in Table 2.2, when $R = \{3, 5\}$, 24KB or 20KB of G elements are used per row sweep. Thus, on a machine with a 32KB L1-D cache, the next set of G elements needs to be loaded from the L2 or below when the input/output channels of focus change. To counter the issue, we block the minibatch dimension (N) with a tile size of M to reuse each G element M times, as in Lines 1 and 6 in Algorithm 2.2. The heuristic is that $M = 16$ is appropriate for most convolution configurations.

Layers such as ReLU, pooling, LRN, normalization, and batch concatenation can be efficiently implemented on the same layout that the convolutional layers use [46], so in most cases we do not need to transpose the activations between layers.

2.3.3 Backward Propagation by Input (BWI)

For a unit-stride convolution, BWI is virtually the same as FWD, with the exception that the filters are flipped. However, non-unit strides introduce some differences. Specifically, when applying the register usage optimization described earlier with horizontal stride $O > 1$, in FWD, we load Q/V new Y vectors into the accumulator registers after we finish processing O vectors of D. However, in BWI, we load $O \times Q/V$ new $\partial L/\partial D$ vectors into the accumulator registers after we finish processing one $\partial L/\partial Y$ vector.

Also, in a FWD row sweep, some D elements may contribute to a number of Y vectors that is less than T due to the horizontal stride; however, in a BWI row sweep, except for the image boundaries, an $\partial L/\partial Y$ element always contributes to T $\partial L/\partial D$ vectors. We generate the appropriate number of skippable VFMAs through JIT.

Finally, the unroll factor of the row sweep loop in FWD is $W \times O$; it is the least common multiple of W and O in BWI.

2.3.4 Backward Propagation by Weights (BWW)

Algorithm 2.4 is a naïve sparse algorithm for BWW. It checks for zeros in D. We can easily modify the algorithm to check for zeros in $\partial L/\partial Y$ instead, if we expect more sparsity in $\partial L/\partial Y$ of the target layer. In Algorithm 2.5, we improve on Algorithm 2.4 by applying output-parallelization and similar optimizations used in FWD and BWI, with some changes.

Algorithm 2.4: Naïve Vectorized Sparse BWW.

input : input D, output gradients dY
output : filter gradients dG

1 **for** $i = 0, c = 0, y = 0, x = 0$ **to** $N-1, C-1, H-1, W-1$ **do**
2 **if** $D_{i,c,x,y} \neq 0$ **then**
3 **for** $k = 0$ **to** $K - V$ **step** V **do**
4 **for** $u = 0, v = 0$ **to** $R-1, S-1$ **do**
5 $dG_{[k:k+V-1],c,u,v} = dG_{[k:k+V-1],c,u,v} + D_{i,c,x,y} \times dY_{i,[k:k+V-1],x-u,y-v}$

In Algorithm 2.5, we vectorize the zero-checking along the minibatch dimension (N) instead of the channel dimension as in FWD and BWI, reflected in Line 7. This is because in BWW, the destination of the VFMA operation, $dG_{[k:k+V-1],c,u,v}$, changes as the input channel c changes. As a result, if we vectorize the zero-checking along the input channel dimension (C), we need to store the previous group of $dG_{[k:k+V-1],c,u,v}$ vectors to memory and load a new group before entering the loop starting at Line 10, and this frequent register spilling may harm performance significantly. Luckily, because $dG_{[k:k+V-1],c,u,v}$ is minibatch-invariant, all input elements from the vector $D_{[i:i+V-1],c,x,y+v}$ contribute to the same group of $dG_{[k:k+V-1],c,u,v}$ vectors. Therefore, vectorizing the zero-checking along the minibatch dimension avoids spilling the registers.

Due to the change in vectorization scheme, we transpose D such that the lowest dimension is a minibatch tile of size V. This allows us to load D vectors directly as opposed to gathering from locations apart.

Algorithm 2.5: Parallel Vectorized Sparse BWW.

input : input D, output gradients dY
output : filter gradients dG

```
1  for i = 0 to N − V step V do
2      for y = 0 to H − 1 do
3          for v = 0 to S − 1 in parallel do
4              for k = 0 to K − Q step Q in parallel do
5                  for c = 0 to C − 1 in parallel do
6                      for x = 0 to W − 1 do
7                          m_[0:V−1] = [d ≠ 0 for d in D_[i:i+V−1],c,x,y+v]
8                          for i' = 0 to V − 1 do
9                              if m_i' is true then
10                                 for k' = k to k + Q − V step V do
11                                     for u = 0 to R − 1 do
12                                         dG_[k':k'+V−1],c,u,v = dG_[k':k'+V−1],c,u,v +
                                               D_i+i',c,x,y+v × dY_i+i',[k':k'+V−1],x−u,y
```

In a row sweep, a core works on $R \times Q$ filter gradients. Because the total number of filter gradients is $R \times S \times K \times C$, the maximum parallelism becomes $S \times C \times K/Q$.

Since the set of filter gradient elements is constant during a row sweep, if we limit the number of filter gradient vectors being worked on, which is $T = R \times Q/V$, to the register budget, they can stay in the registers during the entire row sweep. Consequently, we do not apply the cyclic register load/store and renaming scheme described in Section 13. This also lifts the restriction on the unrolling factor for the row sweep loop so that it can be chosen freely.

Instead of loading the previous partial results of the ∂G vectors at the beginning of a row sweep, and storing the new partial results to memory at the end, we clear the accumulator registers at the beginning and store the VFMA results in them during a row sweep. At the end, we load the previous partial results and add them to the accumulator registers as the new partial results, and we immediately store them back to memory afterwards. Therefore, the filter gradient elements are only accessed twice in succession at the end. We also prefetch the filter gradient elements in software at the beginning. With this optimization, we do not need to tile the minibatch dimension to reuse the filter elements.

The two multiplicand vectors of the VFMA instructions in BWW are the broadcasted input element $D_{i+i',c,x,y+v}$ in a vector register and the $\partial L/\partial Y$ vector $dY_{i+i',[k':k'+V-1],x-u,y}$ as a memory operand.

2.3.5 Generalization to Other Hardware

We implement *SparseTrain* with AVX-512's vector FMA and vector comparison instructions. Other ISAs beyond x86, such as ARM Neon [51], also support them. Nevertheless, the techniques are generalizable to ISAs without them. Without vector comparison, we may fall back to comparing each scalar element to zero. Note that we still need to compare a batch of scalar elements at once and then apply Algorithm 2.3 to combat branch misprediction. On machines without vector FMA, *SparseTrain* is actually more effective. This is because with scalar FMAs, the number of skippable instructions per zero input is much higher, which can more easily hide the branch misprediction penalty.

Our general idea is also applicable to GPUs. On CPUs, the main challenge is to reduce branch misprediction. On the other hand, on GPUs we need to minimize control divergence, which happens when threads in the same SIMD group, or *warp* in CUDA terms, take different paths in a control sequence. One possible solution is to let threads in a given warp compare the same input with zero simultaneously; therefore, all threads in the same warp may either issue or skip the computation. However, the method only works with general MAC computations, and not with hardware accelerated GEMM instructions such as Nvidia's Tensor Core MMA instructions, which compute a GEMM tile directly [52]. As a result, it may be hard for a GPU *SparseTrain* implementation to beat the Tensor Core accelerated GEMM. Nevertheless, the method can be useful on GPUs without a hardware GEMM accelerator (e.g., the integrated GPUs used for inference on edge devices [53]), or when we desire higher precision than the one supported by the accelerator.

2.4 EXPERIMENTAL SETUP

We build *SparseTrain* as new kernels in *MKL-DNN* [45]. We use the *xbyak* JIT assembler [54] to generate the code. Because *TensorFlow* [48] uses MKL-DNN as the backend library on CPUs, we also integrate our new kernels into TensorFlow. We evaluate full network training/inference using TensorFlow with the *SparseTrain*-augmented *MKL-DNN*.

We use *MKL-DNN*'s direct convolution as the baseline, which we refer to as *direct*. *MKL-DNN* has three other implementations of convolution: (1) a method that first flattens the inputs with *im2col* and then applies a GEMM, (2) a vectorized *Winograd* convolution [55] for unit-stride 3×3 convolutions only, and (3) a special kernel that optimizes 1×1 convolutions. We compare *SparseTrain* to them when applicable.

We run our experiments on an Intel Skylake-X server with 6 cores. Each core has two AVX-512 vector units, 32KB L1 I- and D-caches, and a 1MB L2 cache. There is a non-

inclusive 8.25MB shared L3 cache. We disable hyperthreading as well as dynamic frequency scaling. We run 6 threads in parallel.

We evaluate *SparseTrain* with VGG16 [42], ResNet-34, and ResNet-50 [43]. Batch Normalization (*BatchNorm*) [56] affects *SparseTrain*'s effectiveness because when the *conv-BatchNorm-ReLU* structure is present, $\partial L/\partial Y$ becomes dense. Since VGG16 does not employ the structure, *SparseTrain* benefits all of its FWD, BWI, and BWW. However, the two ResNet variants have the structure. Therefore, *SparseTrain* does not accelerate their BWI. Zhang et al. [57, 58] demonstrated that, with proper initialization and data augmentation, one can train ResNet without BatchNorm with marginal accuracy loss. Therefore, we also experiment with the BatchNorm-free ResNet-50, called *Fixup* ResNet-50. To preserve the activation sparsity in both the forward pass and the backward pass, we use a variant of the *Fixup* ResNet that does not contain a scalar bias between each ReLU and its subsequent conv layer.

We first examine *SparseTrain*'s training performance with CIFAR-10 [59] as a proof of concept, and then evaluate with the larger ImageNet-1K [44] data set. Because CIFAR-10 is small, we train ResNet-34 with *SparseTrain* from end to end, and time all conv layers to obtain the training performance. However, training multiple DNNs with ImageNet on a small CPU server takes an unreasonable amount of time, so we adopt the following sampling-based method:

1. We train a network from scratch on GPUs. During training, we checkpoint models at each epoch.

2. For each epoch, we randomly sample 5 mini-batches. For each mini-batch, we run a training iteration with *SparseTrain* using the checkpoint model. We record the average execution time from the 5 samples as *SparseTrain*'s mean performance at the given epoch.

3. We take the average sampled run time across all epochs as *SparseTrain*'s mean training performance.

We observe that the sparsity progression between adjacent epochs is smooth, and that the randomly-sampled mini-batches have low sparsity variations. Therefore, we are confident that the sampled performance faithfully approximates the overall training performance.

Since *SparseTrain* also benefits inference, we use the trained models to evaluate *SparseTrain*'s inference performance.

Besides whole-network performance, we also assess *SparseTrain*'s performance on individual layers at various sparsity levels. For this, we generate synthetic inputs with random

Table 2.3: Evaluated layer configurations from VGG and ResNet v1.5.

Name	C	K	H	W	R	S	O	P
vgg1_2	64	64	224	224	3	3	1	1
vgg2_1	64	128	112	112	3	3	1	1
vgg2_2	128	128	112	112	3	3	1	1
vgg3_1	128	256	56	56	3	3	1	1
vgg3_2	256	256	56	56	3	3	1	1
vgg4_1	256	512	28	28	3	3	1	1
vgg4_2	512	512	28	28	3	3	1	1
vgg5_1	512	512	14	14	3	3	1	1
resnet2_1a	64	64	56	56	1	1	1	1
resnet2_1b	256	64	56	56	1	1	1	1
resnet2_2	64	64	56	56	3	3	1	1
resnet2_3	64	256	56	56	1	1	1	1
resnet3_1a	256	128	56	56	1	1	1	1
resnet3_1b	512	128	28	28	1	1	1	1
resnet3_2	128	128	28	28	3	3	1	1
resnet3_2/r	128	128	56	56	3	3	2	2
resnet3_3	128	512	28	28	1	1	1	1
resnet4_1a	512	256	28	28	1	1	1	1
resnet4_1b	1024	256	14	14	1	1	1	1
resnet4_2	256	256	14	14	3	3	1	1
resnet4_2/r	256	256	28	28	3	3	2	2
resnet4_3	256	1024	14	14	1	1	1	1
resnet5_1a	1024	512	14	14	1	1	1	1
resnet5_1b	2048	512	7	7	1	1	1	1
resnet5_2	512	512	7	7	3	3	1	1
resnet5_2/r	512	512	14	14	3	3	2	2
resnet5_3	512	2048	7	7	1	1	1	1

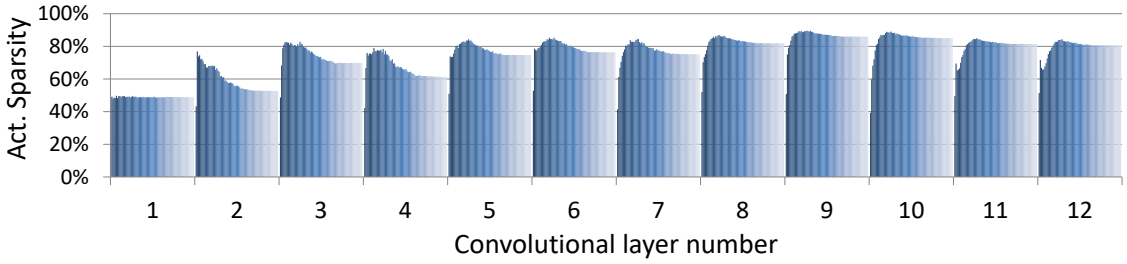
sparse patterns and experiment on all but the first conv layers of VGG and ResNet. We use a batch size of 16 during the experiments. Table 2.3 lists the layer configurations used.

2.5 EVALUATION

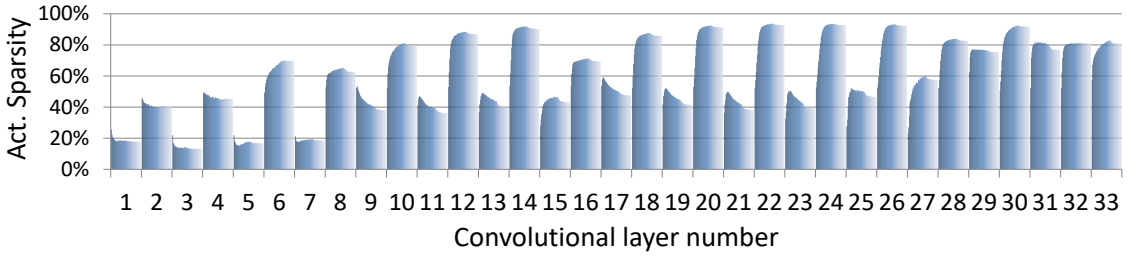
2.5.1 Activation Sparsity in Training

Figure 2.3 presents the ReLU output sparsity at each epoch in the end-to-end training of VGG16, ResNet-34, ResNet-50, and Fixup ResNet-50 with the ImageNet dataset. Each plot shows numbered segments. Each segment is the output from a *conv-ReLU* cluster in VGG16/Fixup ResNet-50, or a *conv-BatchNorm-ReLU* cluster in ResNet-34/50. Within a segment, from left (in darker color) to right (in lighter color) shows the sparsity from the first epoch to the last.

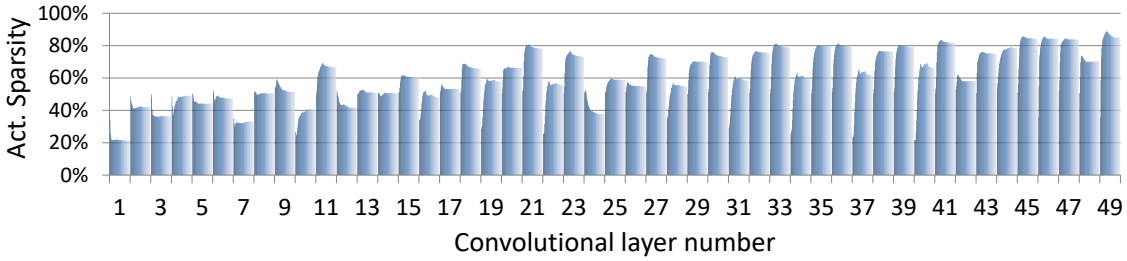
The figure shows that the average sparsity of a layer typically ranges from 20% to 90%.



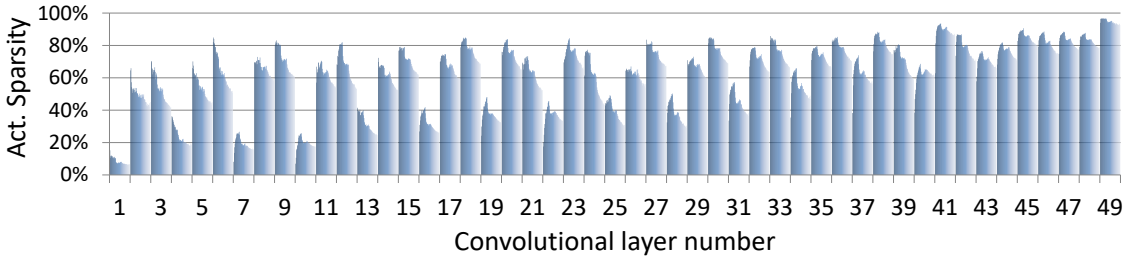
(a) VGG16.



(b) ResNet-34.



(c) ResNet-50.



(d) Fixup ResNet-50.

Figure 2.3: Measured activation sparsity during training with ImageNet. Each numbered segment of the x-axis corresponds to a ReLU-activated convolutional layer in the network. Within a segment, from left to right shows the sparsity from the first epoch to the last.

Later layers generally have higher sparsity then earlier ones [16]. In addition, we also discover that, in the ResNet variants, the sparsity of adjacent layers fluctuates periodically. This is caused by the shortcut in each residual block, which adds positive biases before ReLU and, therefore, lowers the sparsity.

The *conv-ReLU* cluster and the *conv-BatchNorm-ReLU* cluster result in different sparse

inputs to each training component. We list them in Table 2.4. Note that D of a convolutional layer is from the cluster before the layer, while $\partial L/\partial Y$ of a convolutional layer is from the cluster that contains the layer.

Table 2.4: The sparse input to different training components of conv layers.

	FWD	**BWI**	**BWW**
VGG16	D	$\partial L/\partial Y$	D and $\partial L/\partial Y$
ResNet-34	D	N/A	D
ResNet-50	D	N/A	D
Fixup ResNet-50	D	$\partial L/\partial Y$	D and $\partial L/\partial Y$

The table shows that, in vanilla ResNet-34/50, BWI has no sparse input at all due to BatchNorm, making *SparseTrain* ineffective. In this case, one may prefer to use a dense kernel instead. When we evaluate whole-network training, we substitute *SparseTrain* with *direct* for ResNet-34/50's BWI.

On the other hand, when BatchNorm is absent such as in VGG16 and Fixup ResNet-50, BWW's both inputs (D and $\partial L/\partial Y$) are sparse. Therefore, with heuristics or online profiling, one can configure *SparseTrain* to take advantage of the input that has a higher sparsity.

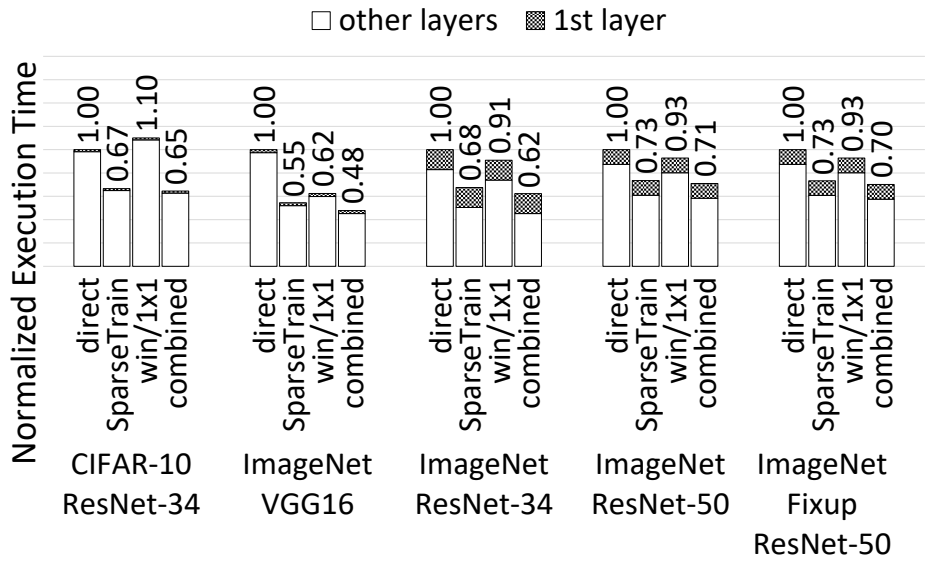
2.5.2 Whole-Network Performance

We now present *SparseTrain*'s whole-network performance. Figure 2.4 shows the end-to-end training (a) and inference (b) time of the convolutional layers with different networks and algorithms. For each network and algorithm, the execution time is normalized to that of *direct*. For training, we break the time into FWD, BWI, and BWW. Because *SparseTrain* is not applicable to the first layer in the network due to the input images often being zero-free, we show the first layer as a separate component.

In the figure, the *SparseTrain* bars correspond to using only *SparseTrain*, or in the case of the ResNet-34/50, using *SparseTrain* for FWD and BWW and *direct* for BWI. The *win/1x1* bars correspond to using the *Winograd* convolution or the optimized *1x1* kernel when possible; otherwise, we use *direct*. The *combined* bars combine the fastest algorithm of each layer. Finally, we find that the method using *im2col* plus GEMM is consistently over 2x slower than *direct*, so we omit it in the figure.

In general, the CIFAR-10 and the ImageNet results are similar. *SparseTrain* achieves notable speedups on all networks. In contrast, *Winograd* performs well on VGG16 but worse on ResNet. Further, its performance with CIFAR-10 is much lower because it performs badly

(a) End-to-end training.



(b) Inference.

Figure 2.4: Breakdown of the execution time of all convolutional layers from different networks, normalized to the dense direct convolution.

with small input width and height. Since CIFAR-10 has small input images (32×32), the input width and height are as low as 4 in the later layers.

Table 2.5 lists the speedups of the different algorithms over *direct*, both including and excluding the first layer. The first layer contributes 1-3% of the execution time of CIFAR-10 ResNet-34 and VGG16, but rises to 9-12% for the ImageNet ResNet variants. This is

because the ImageNet ResNet variants have costly 7×7 first layers.

Table 2.5: Speedup of the different algorithms over the dense direct convolution on all of the evaluated networks' convolutional layers at realistic sparsity levels.

		Including the first layer			Excluding the first layer		
		SparseTrain	win/1x1	combined	SparseTrain	win/1x1	combined
Training							
CIFAR-10	ResNet-34	1.35x	0.71x	1.47x	1.36x	0.70x	1.48x
ImageNet	VGG16	2.15x	1.66x	2.35x	2.19x	1.68x	2.40x
	ResNet-34	1.31x	0.99x	1.48x	1.37x	0.98x	1.58x
	ResNet-50	1.28x	1.08x	1.39x	1.31x	1.09x	1.44x
	Fixup ResNet-50	1.45x	1.08x	1.53x	1.51x	1.09x	1.62x
Inference							
CIFAR-10	ResNet-34	1.50x	0.91x	1.55x	1.51x	0.91x	1.57x
ImageNet	VGG16	1.83x	1.60x	2.09x	1.88x	1.63x	2.15x
	ResNet-34	1.48x	1.10x	1.61x	1.64x	1.12x	1.83x
	ResNet-50	1.36x	1.08x	1.41x	1.44x	1.09x	1.50x
	Fixup ResNet-50	1.36x	1.08x	1.43x	1.44x	1.09x	1.52x

We can see that, when including the first layer, *SparseTrain* speeds up the training of the convolutional layers in the studied networks by 1.28-2.15x. By choosing the best algorithm for each layer, we can speed up training by 1.39-2.35x. The speedup on VGG16 is notably higher than that on the ResNet variants because: (1) VGG16 has higher dynamic sparsity, and (2) VGG16 does not have BatchNorm, so *SparseTrain* is applicable to its BWI. Note that *SparseTrain* speeds up Fixup ResNet-50 by 1.45x instead of 1.28x on the original ResNet-50. The reason is also the absence of BatchNorm. Without including the first layer, the speedups of *SparseTrain* are 2.19x, 1.37x, 1.31x, and 1.51x for VGG16, ResNet-34, ResNet-50, and Fixup ResNet-50, respectively, with ImageNet.

For inference, when including the first layer, *SparseTrain* speeds up the conv layers in the studied networks by 1.36-1.83x. The numbers increase to 1.41-2.09x after choosing the best algorithm for each layer. Without including the first layer, the speedups of *SparseTrain* are 1.88x, 1.64x, 1.44x, and 1.44x for VGG16, ResNet-34, ResNet-50, and Fixup ResNet-50, respectively, with ImageNet. Overall, *SparseTrain* delivers good speedups over the state-of-the-art across networks for end-to-end training and inference.

2.5.3 3×3 Convolutional Layers

We now consider *SparseTrain*'s performance on individual convolutional layers with different filter sizes. We first discuss 3×3 ($R = S = 3$) filters, which have become more popular than larger filter sizes in recent years.

Figure 2.5 shows the speedup of *SparseTrain* at 0-90% sparsity, of *im2col*, and of *Winograd* over *direct*, for FWD, BWI, and BWW on the 3×3 layers from the evaluated networks. Table 2.6 lists the mean speedup at various levels of sparsity.

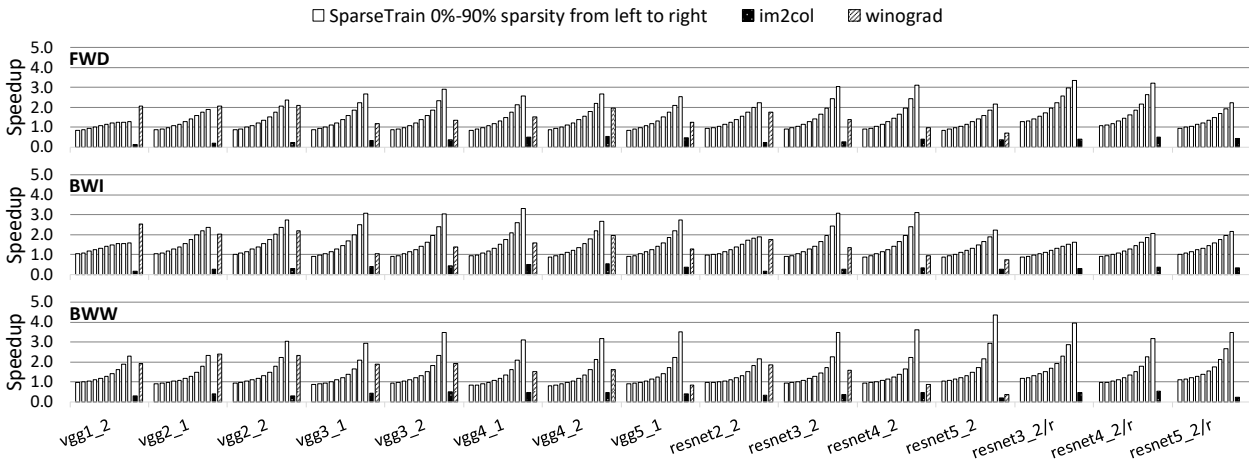


Figure 2.5: Speedup of the different algorithms over the dense direct convolution on the individual 3×3 layers at different sparsity levels.

The table shows that, at 0% sparsity (i.e., a dense input), *SparseTrain* reaches 92-95% of *direct*'s performance on average, depending on the component. This indicates that the overhead to check for and exploit sparsity is low, and the loop order, as well as the tiling strategy of *SparseTrain* are effective.

Table 2.6: Mean speedup at various sparsity for 3×3 layers.

	SparseTrain										*im2c.*	*win.*
	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%		
FWD	0.92	0.96	1.04	1.13	1.24	1.38	1.56	1.79	2.11	2.48	0.33	1.45
BWI	0.93	0.98	1.06	1.15	1.26	1.40	1.58	1.81	2.10	2.45	0.31	1.48
BWW	0.95	0.98	1.03	1.10	1.18	1.30	1.48	1.76	2.23	3.15	0.37	1.44

The speedups of *SparseTrain* increase with the sparsity. On average, the sparsity cross-over point for *SparseTrain* to outperform *direct* is between 10-20%, which is lower than the observed sparsity during training. At 50% sparsity, which is the expected value at the beginning of the training when the distribution of the weights is centered at 0, *SparseTrain* on average delivers a 1.30-1.40x speedup. Typically, the later layers in a network have higher sparsity than the earlier layers. For the later layers, the sparsity reaches over 90% for VGG16 and ResNet-34, and over 80% for ResNet-50. At such levels, *SparseTrain* is on average over 2x faster than *direct*.

im2col is always significantly slower than *direct*. Although GEMM on CPU is highly optimized, the flattening of the inputs through *im2col* incurs severe time and memory overhead.

Winograd is only applicable to unit-stride layers. For these layers, *Winograd* is on average 1.44-1.48x faster than *direct*. However, because the *Winograd* algorithm transforms the problem to the "Winograd space," it has two drawbacks. First, the transformation introduces numerical instability as the filter size increases, so its application is usually limited to 3×3 layers [60]. Second, it requires additional workspace memory.

SparseTrain performs better at later layers (e.g., *vgg5_1*), while *Winograd* dominates at earlier layers (e.g., *vgg1_2*). This is partly due to the increased sparsity at later layers; on average, it takes at least 50-60% sparsity for *SparseTrain* to surpass *Winograd*. The other reason is that early layers have a smaller number of channels, which limits the number of skippable VFMAs per input element, and thus reduces the efficiency of *SparseTrain*. For example, both *vgg1_2* and *resnet2_2* have C and K of 64, giving us only 12 skippable VFMAs in *SparseTrain*. Overall, since *SparseTrain* and *Winograd* have different specialties, they can supplement each other.

SparseTrain for BWI delivers similar performance as for FWD with unit-stride. However, for stride-2 layers (layers with the /r suffix in Figure 2.5), BWI is slower than FWD. As discussed in Section 2.3.3, $\partial L/\partial D$ needs to be loaded O^2 times more rapidly during a row sweep in BWI than Y being loaded in FWD. Therefore, BWI suffers from cache bandwidth limitations.

2.5.4 1×1 Convolutional Layers

1×1 layers ($R = S = 1$) are widely used in ResNet-50's bottleneck blocks. They are unique amongst convolutions in that the spatial reuse of $R \times S$ is absent.

Figure 2.6 shows the speedup of *SparseTrain* at 0-90% sparsity, of *im2col*, and of the specialized *1x1* kernel over *direct*, for FWD, BWI, and BWW on the 1×1 layers from the evaluated networks. Table 2.7 lists the mean speedup at various levels of sparsity.

SparseTrain exploits a convolution's high compute-to-memory ratio. However, the ratio for 1×1 layers is 9x lower than that for 3×3 layers with the same input/output/channel sizes. Thus, as we eliminate useless VFMAs, 1×1 layers may become bandwidth-bound sooner than 3×3 layers. Therefore, at high sparsity, *SparseTrain* is less effective on 1×1 layers than on 3×3 layers, only reaching 1.66-2.04x speedups on average at 80% sparsity.

We also notice that BWW behaves differently than the other two components. At 0% sparsity, *SparseTrain*'s performance is on par with the *direct* for FWD and BWI. For BWW, though, *SparseTrain* only attains 71% of *direct*. However, at high sparsity, *SparseTrain*'s

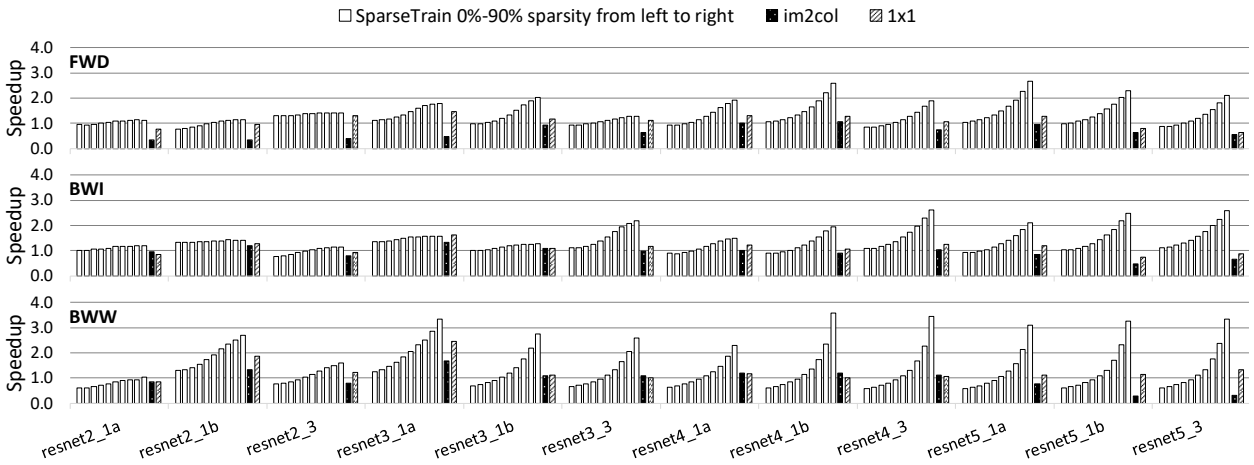


Figure 2.6: Speedup of the different algorithms over the dense direct convolution on the individual 1×1 layers at different sparsity levels.

Table 2.7: Mean speedup at various sparsity for 1×1 layers.

	SparseTrain										*im2c.*	*1x1*
	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%		
FWD	0.97	0.98	1.03	1.09	1.17	1.27	1.39	1.51	1.66	1.78	0.62	1.06
BWI	1.03	1.03	1.08	1.15	1.22	1.33	1.43	1.53	1.66	1.76	0.91	1.08
BWW	0.71	0.76	0.83	0.92	1.05	1.20	1.39	1.66	2.04	2.61	0.87	1.23

speedup is higher for BWW than for FWD and BWI.

The difference between BWW and FWD stems from two competing factors, both related to how BWW accesses $\partial L/\partial Y$ against how FWD accesses Y. First, BWW uses a different loop order, and in a row sweep touches V times more elements from $\partial L/\partial Y$ than FWD touches Y at 0% sparsity. Second, BWW reads $\partial L/\partial Y$ elements as a memory operand of a VFMA. When we skip a group of VFMAs, we also skip the access to the $\partial L/\partial Y$ elements. At high sparsity, we eliminate many such accesses. In contrast, FWD loads and stores Y elements using the cyclic register allocation scheme. Therefore, the Y elements are loaded and stored regardless of sparsity pattern. As a result, at low sparsity, BWW performs many more memory accesses, and at high sparsity, performs many fewer. This effect is less visible at 3×3 layers due to their higher compute-to-memory ratio; however, it is very obvious at 1×1 layers.

The fewer channels in earlier 1×1 layers hurts *SparseTrain* more than they do in earlier 3×3 layers due to the absence of spatial reuse. For example, *resnet2_1a* has 64 for C and K, resulting in only 4 VFMAs being skippable per zero-checking. Consequently, we can hardly see any speedup from *SparseTrain* on earlier 1×1 layers. Nonetheless, we can still efficiently leverage the dynamic sparsity in later 1×1 layers.

As shown in Table 2.7, on average, the cross-point sparsity for *SparseTrain* to surpass the specialized *1x1* kernel is around 20% for FWD, at 0% for BWI, and around 40% for BWW.

In addition to 1×1 and 3×3 layers, we also experimented with several 5×5 layers from AlexNet [1] and GoogLeNet [61] and got even higher speedups. We omit the results due to lack of popularity of the 5×5 layers. Finally, we confirmed that *SparseTrain*'s execution time scales linearly with minibatch size N by experimenting with $N = \{32, 64\}$.

2.5.5 Mitigating Branch Misprediction Penalty

The unpredictable loop branch in Line 3 of Algorithm 2.3 accounts for most of the branches in *SparseTrain* because it is in the innermost loop. Moreover, the rest of the branches are all predictable. Therefore, it is crucial for Algorithm 2.3 to hide the branch misprediction penalty. To evaluate the performance of Algorithm 2.3, we design a study that quantifies the performance headroom that *SparseTrain* has if the branch in Line 3 was instead predictable.

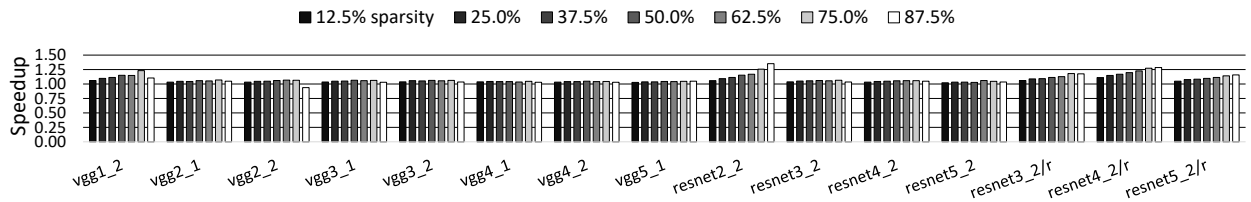
In Algorithm 2.3, branch mispredictions stem from an unpredictable number of non-zero elements in an input channel vector (`o` in Line 2). In our study, we eliminate the misprediction at steady state by using special input data that has a fixed number n of non-zero elements per input channel vector. With this input, a local history predictor can easily predict the loop branch. In the experiments, we vary n from 1 to 15 (which is the value of $V - 1$). As a result, the sparsity of the input is $1 - n/V$ for a given n.

Figure 2.7 shows the speedup from eliminating branch mispredictions at steady state in Algorithm 2.3, at selected sparsity levels. The 3×3 layers are on the top, and the 1×1 layers are at the bottom. Each bar shows *SparseTrain*'s FWD execution time with a random input (*with* branch misprediction) over *SparseTrain*'s FWD execution time with the special input (*without* branch mispredictions), at different sparsity levels.

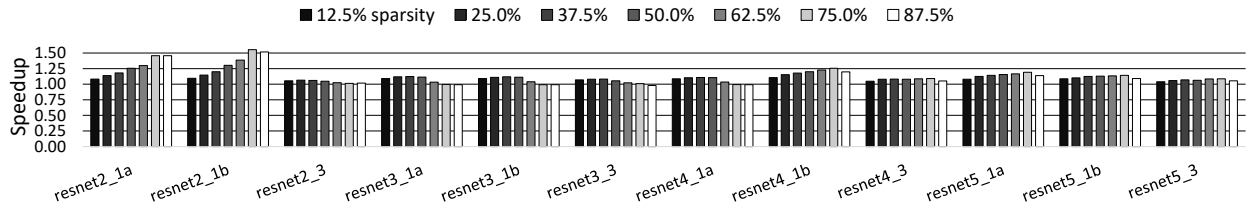
In most 3×3 layers (Figure 2.7a), *SparseTrain* sets the number of skippable VFMAs (T) to 24. As a result, the misprediction penalty is well hidden. Consequently, eliminating misprediction generally provides only up to 5% speedup.

There are, however, some exceptions. First, *vgg*1_2 and *resnet*2_2 both have a small output channel of $K = 64$. Therefore, *SparseTrain* can only set T to 12, which is insufficient to fully hide the misprediction penalty. Second, the stride-2 layers (layers with the /r suffix) have a variable T that is on average less than 24, exposing some of the penalty. As a result, eliminating misprediction speeds up these layers as sparsity increases, reaching up to 35% for *resnet*2_2 at 87.5% sparsity.

The 1×1 layers (Figure 2.7b) have a lower T of ≤ 16 due to a lack of spatial reuse. The worst case is *resnet*2_1*a* and *resnet*2_1*b*, whose T is only 4. For these layers, eliminating



(a) 3×3 layers.



(b) 1×1 layers.

Figure 2.7: Speedup over *SparseTrain* FWD when branch mispredictions in Algorithm 2.3 are eliminated.

misprediction can yield over 50% speedup at high sparsity. Nonetheless, Algorithm 2.3 performs relatively well on other 1×1 layers, leaving only a small room for improvement.

In conclusion, while Algorithm 2.3 is less effective as T decreases, it performs well and close to the upper bound on most of the studied layers. Further reducing mispredictions in software may be hard. However, previous hardware proposals [62] could help: since the loop trip count is generated outside of the loop body, the count could be communicated to the branch predictor in hardware, completely eliminating branch mispredictions.

2.6 RELATED WORKS

With the pursuit for higher accuracy, DNN models become larger with more layers. These computation and memory intensive DNN models bring heavy burden for memory system and processors. Fortunately, the DNN models are always over-parameterized to approximate the target function. Various works compress DNN models by eliminating redundant weights [20, 41]. Weight quantization [63, 64] sacrifices numerical precision to reduce model size. Structured sparsity [24] is more hardware-friendly, but it is inapplicable to training and does not exploit dynamic sparsity in the activation.

PruneTrain [23] prunes entire channels and reconfigures the model to a smaller dense form during training. *SparseTrain* is orthogonal to it. *SparseTrain* can further leverage the activation sparsity after PruneTrain reconfigures the model.

meProp [65] sparsifies the back propagation of LSTMs and MLPs by only propagating a

small number of gradients in each pass. This reduces back propagation time. Yet, it does not affect the forward propagation, nor has it been tested on CNNs. Our work is orthogonal to it and can be applied in conjunction with it.

Several DNN accelerators exploit the sparsity in weights and/or activations. Cnvlutin [31] leverages activation sparsity to skip ineffectual computations. Eyeriss [37] clock-gates the hardware when there are zeros in the activation. It saves energy but not cycles. Cambricon-X [30] skips multiplications associated with pruned weights. EIE [29] exploits sparsity in both weights and activations of fully connected layers. SCNN [28] leverages sparsity in both weights and activations of conv layers. These accelerators are application specific, while our work targets general-purpose processors.

SparCE [38] skips ineffectual code blocks based on a sparse input. It annotates skippable code blocks in software and tests conditions in hardware. Therefore, it requires hardware-software co-design. Also, it mainly works on scalar code. We target high-performance SIMD CPUs and use software only.

ZCOMP [5] adds new instructions to load/store vectors from/to memory in a compressed form. It perfectly synergizes with *SparseTrain* because its reduction in memory traffic is proportional to *SparseTrain*'s reduction in compute intensity.

Apart from sparsity, algorithmic transformations are developed to speed-up convolution. Georganas et al. [46] and Zhang et al. [66] demonstrate that on CPUs, well-tuned direct convolution is much faster than the conventional *im2col*-based convolution. For larger filter sizes, the FFT-based convolution [67] accelerates the computation by operating in the frequency space. For smaller 3×3 layers, the Winograd algorithm [55] reduces computation by incorporating the Chinese Remainder theorem. However, the transformation introduces numerical instability with larger filter sizes and erases the dynamic sparsity in the activation.

Liu et al. [68] restore the activation sparsity with the Winograd convolution by applying ReLU to the activation after transforming to the Winograd space. However, their approach changes the network structure. In addition, their focus is to reduce the operation count for running DNN inference on mobile devices, and they do not target training or an efficient vectorized implementation.

2.7 CONCLUSION

This chapter tackles the challenges posed by the moderate and dynamic sparsity in DNN activations with software methods. We present *SparseTrain* — the first approach that exploits dynamic sparsity in software for accelerating DNNs on general-purpose CPUs. *SparseTrain* identifies zeros in a dense data representation and performs vectorized computation.

It is applicable to inference and all major components of training: forward propagation, backward propagation by inputs, and backward propagation by weights.

We evaluate *SparseTrain* on a 6-core Intel Skylake-X server. In end-to-end training of VGG16, ResNet-34, and ResNet-50 with ImageNet, *SparseTrain* outperforms a highly-optimized direct convolution on the non-initial convolutional layers by 2.19x, 1.37x, and 1.31x, respectively. In inference, it accelerates the non-initial convolutional layers of the aforementioned models by 1.88x, 1.64x, and 1.44x, respectively. Overall, *SparseTrain* is effective and opens up new research directions in speeding-up computations with modest sparsity.

CHAPTER 3: SAVE: SPARSITY-AWARE VECTOR ENGINE FOR ACCELERATING DNN TRAINING AND INFERENCE ON CPUS

3.1 INTRODUCTION

Although *SparseTrain* significantly accelerates DNN training and inference on CPUs, it requires intensive software efforts and only leverages the sparsity in the activations. In order to speedup legacy DNN software and to also exploit the sparsity in the weights, we further propose hardware changes to the CPU architecture.

Exploiting unstructured sparsity on SIMD CPUs is 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 the Vector Processing Units (VPUs) that perform vector MAC, a.k.a 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 [50, 69]. 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 the Reservation Stations (RS), to find and dynamically schedule effectual operations from different instructions to available VPU lanes. This chapter presents our new vector pipeline — the *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. The result is fewer VPU operations, leading to speedup. SAVE is the first VPU pipeline that exploits sparsity and has the following properties.

- It exploits unstructured sparsity in both vector multiplicands in a VFMA. It poses no perceptible performance overhead when the inputs are fully dense.
- It is transparent to software and can accelerate legacy software without any modification.

- It is *general-purpose* that it does not only benefit DNN workloads; instead, it can speedup any vector workload that utilizes VFMAs and has zeros in the inputs.
- It requires minor hardware modifications to the existing vector pipeline. Aside from a basic compaction technique, it also incorporates various optimizations to increase the efficiency of the compression with small additional hardware cost.
- It leverages the spatial locality in the broadcasts that are common in GEMMs to prevent the L1-D read bandwidth from becoming a new bottleneck once the compute is significantly reduced by skipping ineffectual work.
- It supports mixed-precision computations.

To the best of our knowledge, SAVE is the first CPU vector pipeline that exploits unstructured sparsity. 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.

3.2 BACKGROUND

3.2.1 Matrix Multiplication

GEMM is the core operation in DNNs. LSTMs[70] and batched MLPs use GEMM as a building block. Convolution can be computed either through (un)folding a big GEMM[71] or directly with a series of small GEMMs[46].

Figure 3.1 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.

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.

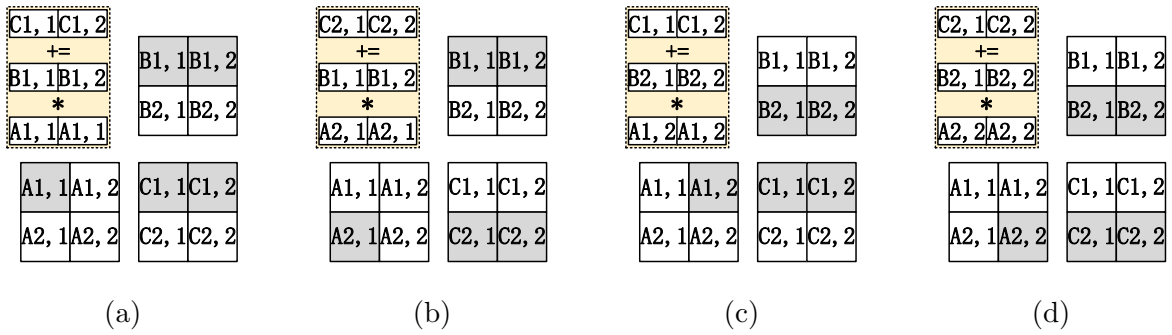


Figure 3.1: Vectorized GEMM with two lanes on a 2×2 tile.

3.2.2 Instruction Set Architecture for GEMM

We evaluate our ideas in an x86 environment with AVX-512 extensions [72]. 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 [73] and Intel AVX-512 [72], 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:

$$C'_i = C_i + A_i B_i \tag{3.1}$$

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 Figure 3.1.

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[74] and quantization[75] improve DNN performance. One can use lower precision multiplicands, but the accumulator often keeps a higher precision [76]. 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[77]. 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 [7, 74].

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:

$$C'_i = C_i + A_{[2i:2i+1]} \bullet B_{[2i:2i+1]} \tag{3.2}$$

We use $\bullet$ to denote vector dot product. VDPBF16PS computes the dot product in hardware by performing two consecutive MAC operations [72], shown in Figure 3.2.

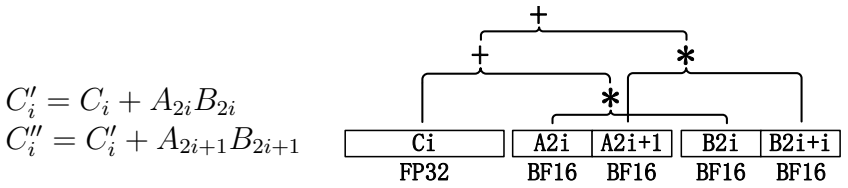


Figure 3.2: Mixed-precision operation in VDPBF16PS.

3.2.3 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[69]. Similarly, AMD's Zen has 6-wide dispatch but only 2 VPUs [50]. As a result, their front-end bandwidth is heavily underutilized in DNN computations.

3.3 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.

Figure 3.3 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 3.4, we present advanced techniques that increase SAVE's performance. Finally, Section 3.5 introduces additional SAVE support for mixed-precision VFMAs.

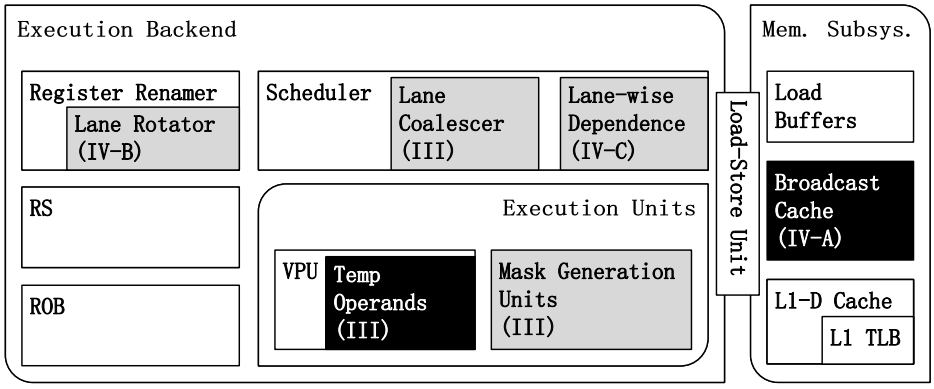


Figure 3.3: SAVE adds logic units (gray) and storage units (black) to the execution backend and the memory subsystem. Each block is discussed in the section in the parentheses.

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.

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 [72]. 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. Figure 3.4 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 bottle-

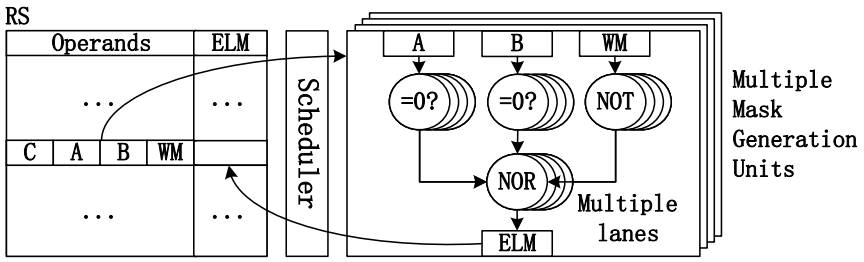


Figure 3.4: Multiple Mask Generation Units (MGUs) producing the Effectual Lane Masks (ELMs) of multiple VFMAs in parallel. The RS shows a single VFMA being worked on.

neck. 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 3.1 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 an unscheduled effectual lane in the corresponding position. (Lines 3-9). This is done in a single cycle with conventional priority-based *select logic*[78]. 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).

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 [78]. 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

Algorithm 3.1: Scheduling of vertical coalescing

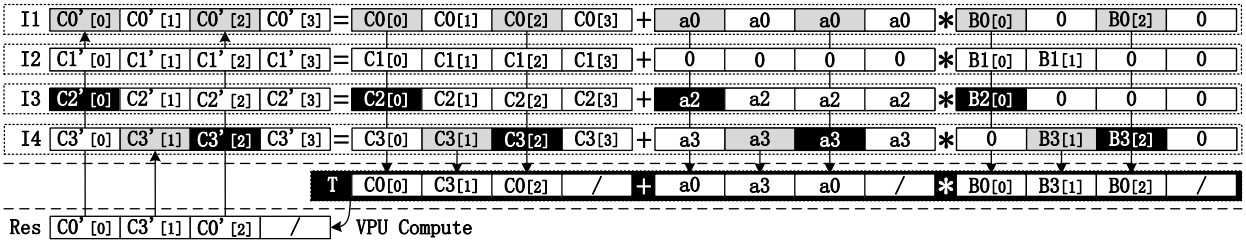
```
definition: temp operands T, reservation stations RS, vector processing unit VPU,
            effectual lane mask ELM, μop identifier ID, vector length V
clear(T);
for lane in V in parallel do
    for μop in RS do
        if isVFMA(μop) AND ready(μop) AND μop.ELM[lane] then
            T.accum_base[lane] = μop.accum_base[lane];
            T.multiplicands[lane] = μop.multiplicands[lane];
            T.ID[lane] = μop.ID;
            μop.ELM[lane] = 0;
            break
if NOT empty(T.ID) then
    VPU.issue(T);
for μop in RS in parallel do
    if isVFMA(μop) AND empty(μop.ELM) then
        RS.remove(μop);
```

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.

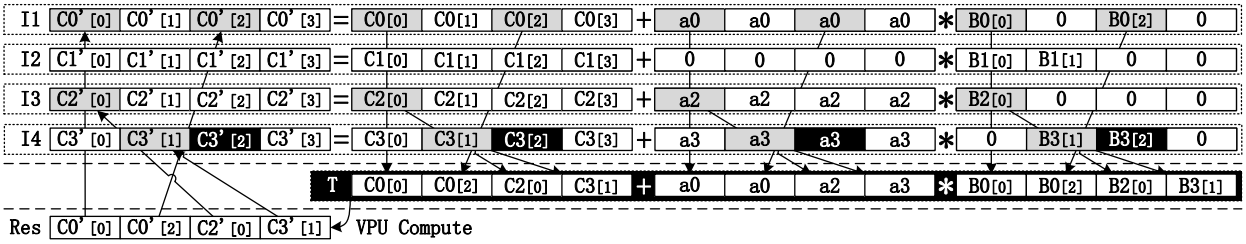
Figure 3.5a illustrates a single compaction via vertical coalescing with four quad-lane VFMA instructions, I_1-I_4, in program order. Each accumulator is shown both as an input (C) and as an output (C'), which are renamed to separate physical registers. $a0$-$a3$ are scalars broadcasted to all vector 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 chapter'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



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



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

Figure 3.5: 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.

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. Figure 3.5b 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 [5, 16]. 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.

3.4 ADVANCED FEATURES

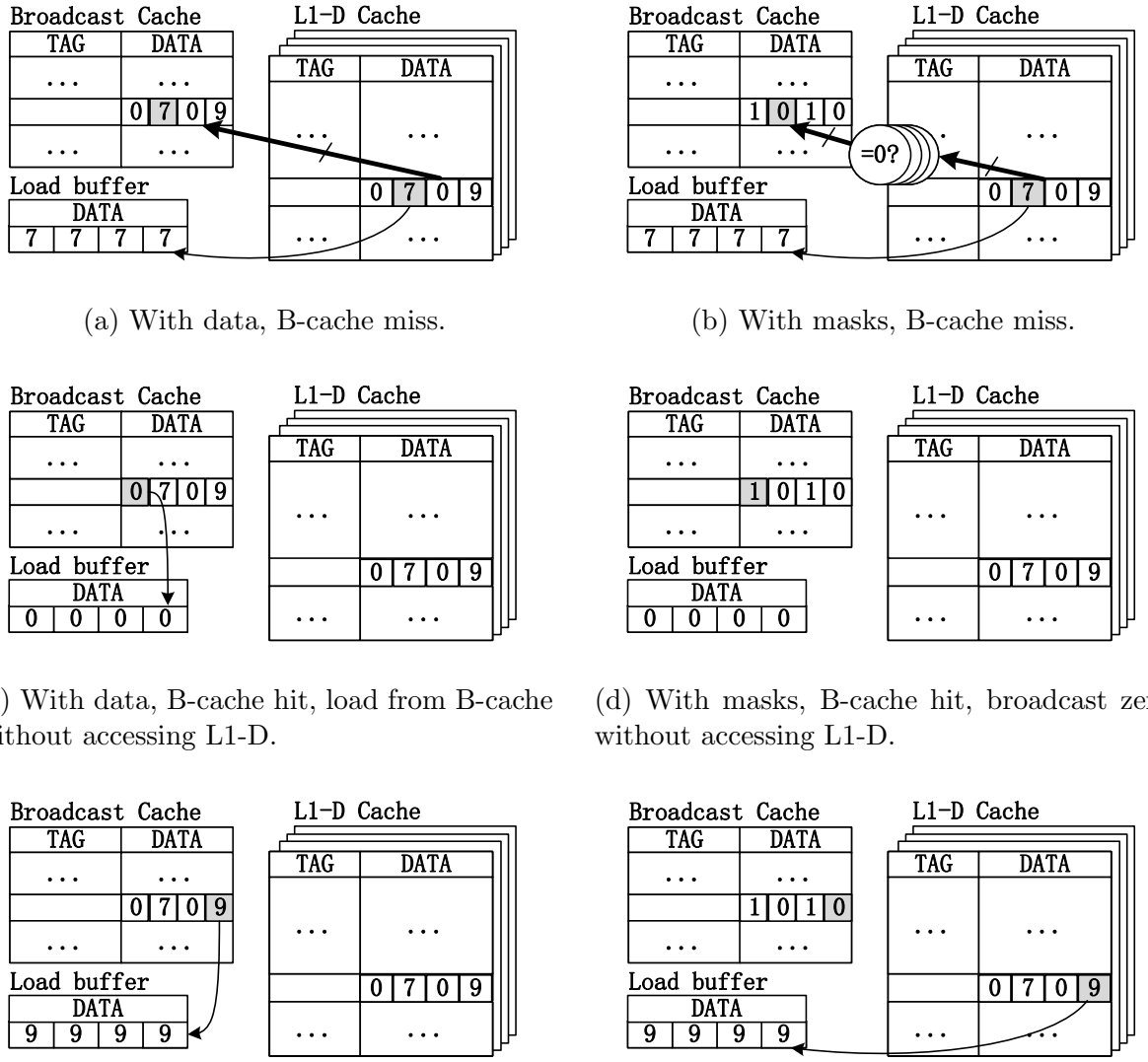
SAVE is enhanced with additional features to improve performance. First, because GEMM frequently issues vector broadcasts, we add a small high-bandwidth *Broadcast Cache* to exploit locality and improve broadcast throughput. Second, to load-balance VPU lanes, we introduce the *Rotate-Vertical Coalescing 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.

3.4.1 Broadcast Cache

The basic SAVE design speeds up computation when VFMA throughput is the only bottleneck, such as in the explicit broadcast pattern (Section 3.2.2), 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 [50]. 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 Figure 3.6. In the figure, cache lines hold four vector elements.



(a) With data, B-cache miss.

(b) With masks, B-cache miss.

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

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

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

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

Figure 3.6: Broadcast Cache with data (left) or with masks (right).

In the first B$ design, a broadcast μop checks the B$. On a miss (Figure 3.6a), SAVE fetches the corresponding line from L1-D, stores it in the B$, 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 (Figure 3.6c) or not (Figure 3.6e).

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$ (Figure 3.6b). In addition, it broadcasts the requested element into the load buffer.

When a broadcast μop hits in the B$, SAVE checks the corresponding mask bit. If set (Figure 3.6d), SAVE populates the load buffer with zeros and does not read the data from the L1-D. Otherwise (Figure 3.6f), SAVE fetches the data from the L1-D. Overall, this B$ 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 Figure 3.1 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.

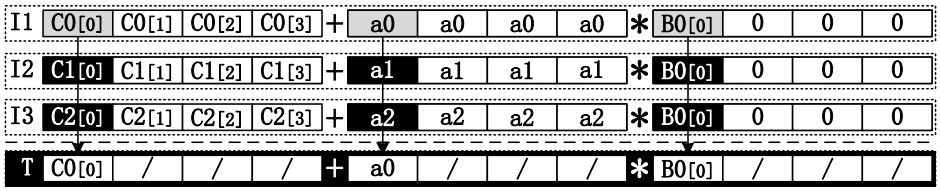
3.4.2 The Rotate-Vertical Coalescing Scheme

Vertical coalescing is sensitive to imbalanced load across lanes. Such imbalance is inherent when we reuse a register holding non-broadcasted data. Figure 3.7a 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.

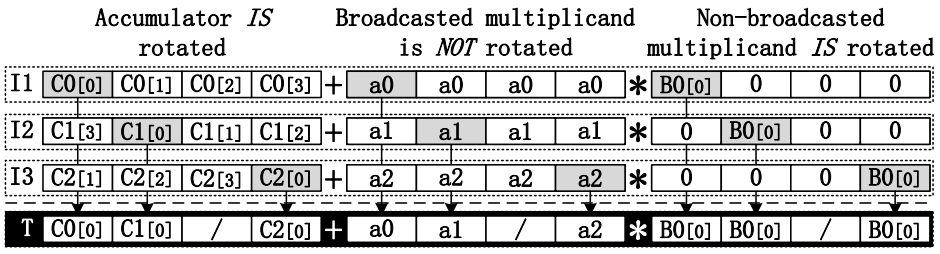
SAVE improves vertical coalescing by assigning a *Rotational State (R-state)* to each VFMA. Depending on the state, we rotate a VFMA's operands to the left or right by one lane, or do not rotate them at all. In this way, we limit the rotations and thus the hardware cost. Rotation eases the imbalance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

Figure 3.7b shows an example. Figure 3.7b is like Figure 3.7a 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; there-



(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.

Figure 3.7: Operand rotation combats load imbalance in vertical coalescing.

fore, 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 Figure 3.7b.

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.

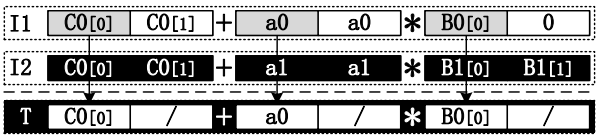
With the two optimizations, SAVE only needs to store up to 3 copies of each non-broadcasted multiplicand (one for each R-state). Since the multiplicands are highly reused in GEMM kernels, the actual number of additional registers needed is low. We observe that, when running a typical explicit broadcast 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 vector register file in the baseline micro-architecture does not become a bottleneck with such additional consumption. As a result, we do not expand the register file.

3.4.3 The Lane-Wise Dependence Scheme

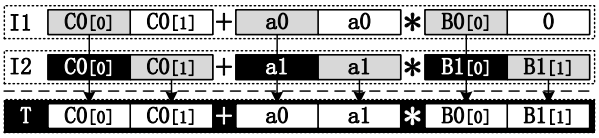
Current SIMD processors track data dependences at the vector register granularity. A dependent VFMA is ready when all lanes in the source VFMA complete. We call this a *Vector-Wise Dependence*, which may create *false dependences* between VFMAs when vertical coalesing is employed.

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.

Figure 3.8a 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.



(a) Vector-wise dependence.



(b) Lane-wise dependence.

Figure 3.8: Vector-wise dependence prevents I_2's lane 1 from issuing with I_1's lane 0. Lane-wise dependence does not.

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. Figure 3.8b 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, 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 Figure 3.8b, 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 3.1, 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 [78].

3.5 MIXED-PRECISION TECHNIQUES

Recently, manufacturers started to support mixed-precision VFMAs [76, 77], mostly for DNNs. The x86 implementation was described in Section 3.2.2. We now extend SAVE to apply the techniques in Section 3.3 and 3.4 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 Figure 3.9, which omits rotation for simplicity. It shows 2 ALs, each mapped to 2 MLs. The dot product (denoted with $\bullet$) 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.

3.5.1 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

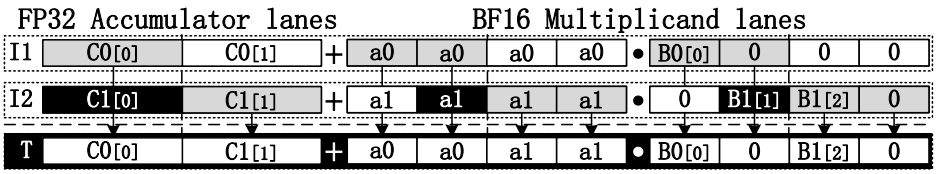


Figure 3.9: Vertical coalescing is inefficient for mixed-precision. An accumulator lane can be skipped only when both multiplicand lane pairs mapped to it are ineffectual. The operator • stands for the dot product of two-lane multiplicand sub-vectors.

$2i$ of I_2 are both ineffectual. Their computation for AL i becomes:

$$\begin{aligned} I_1 &: C0_i = C0_i + A0_{2i}B0_{2i} + 0 \\ I_2 &: C0_i = C0_i + 0 + A1_{2i+1}B1_{2i+1} \end{aligned} \tag{3.3}$$

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

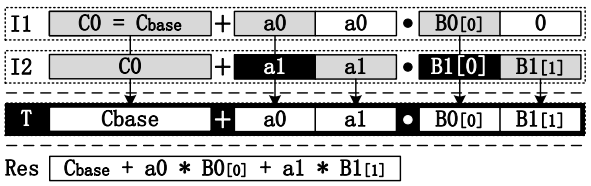
$$C0_i = C0_i + A0_{2i}B0_{2i} + A1_{2i+1}B1_{2i+1} \tag{3.4}$$

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.

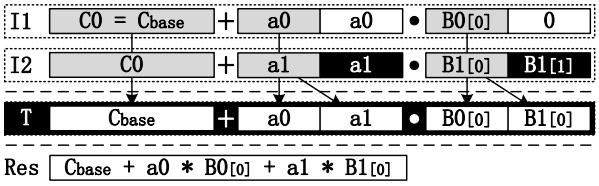
For example, Figure 3.10a *vertically* combines I_1's ML 0 and I_2's ML 1. I_2's ML 1 is accumulated into $C0$ before I_2's ML 0. This changes the accumulation order. In contrast, Figure 3.10b *horizontally* schedules I_1's ML 0 and then I_2's ML 0, therefore preserving the accumulation order. The figures 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 Section 3.3, we claimed that SAVE forsakes horizontal compression on the 16-lane vector due to hardware complexity. However, in the mixed-precision case, it is 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.

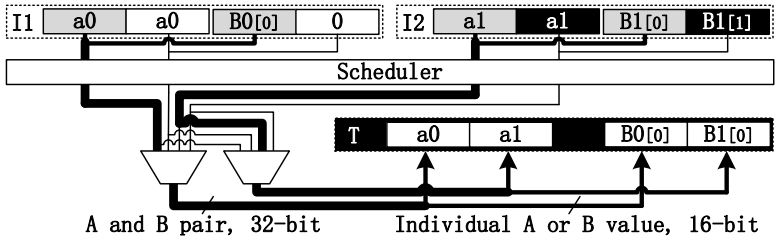
SAVE implements horizontal compression within each AL with two cheap 32-bit 4-to-1 multiplexers, as shown in Figure 3.10c. Each multiplexer selects a pair of A and B multiplicands from 4 candidates. Furthermore, it only needs bubble-collapsing to compact



(a) Vertically coalescing multiplicand lanes.



(b) Horizontally compressing 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).

Figure 3.10: Horizontal compression on multiplicand lanes preserves accumulation order while vertical coalescing does not.

the inputs into T; it does not need to bubble-expand the data out of the VPU after the computation.

3.5.2 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 Figure 3.10b 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.

3.5.3 Example of a Mixed-Precision VFMA

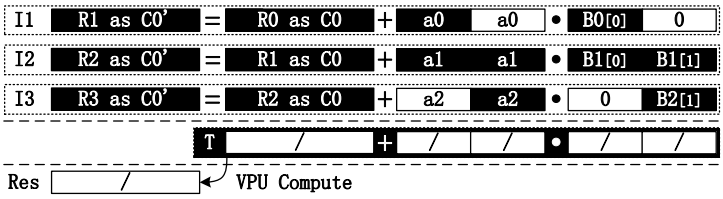
For simplicity, we show an example instead of listing the complete algorithm. Figure 3.11 illustrates how SAVE generates proper results for a single AL from 3 instructions with the same accumulator $C0$. The figure also illustrates the register renaming for the accumulator. In the following, we assume 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.

Figure 3.11a 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 $R3$.

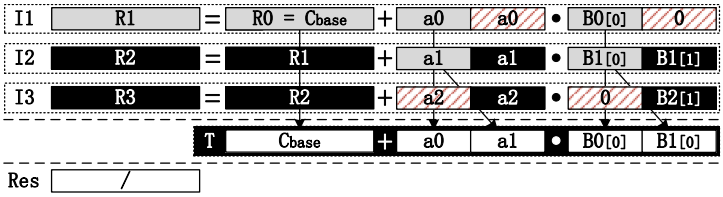
In cycle 1 (Figure 3.11b), the initial value of C_0 is C_{base}, held in $R0$. We issue the first VPU operation with ML 0 from both I_1 and I_2, and use $Cbase$ as the accumulation base. The ineffectual lanes are marked as completed with a slashed pattern.

In cycle 2 (Figure 3.11c), 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.

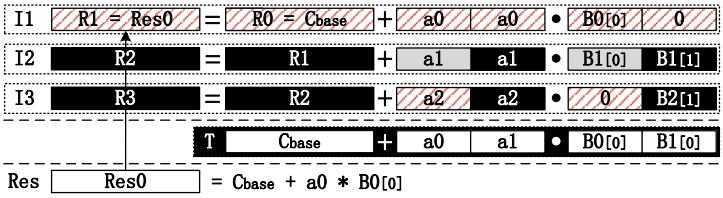
In cycle 3 (Figure 3.11d), there are two operations. First, the second result (Res_1) of the first VPU operation is produced. We mark I_2's ML 0 as completed. However, since I_2's ML



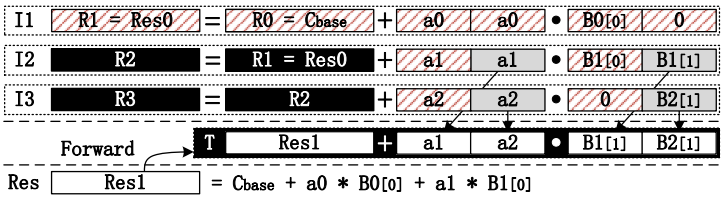
(a) Initial state.



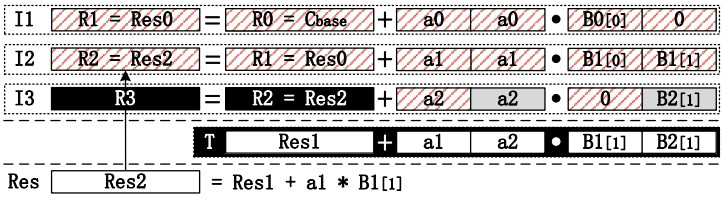
(b) Cycle 1.



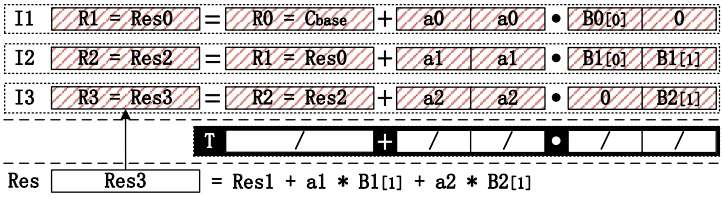
(c) Cycle 2.



(d) Cycle 3.



(e) Cycle 4.



(f) Cycle 5.

Figure 3.11: 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.

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 (Figure 3.11e), the first result of the second VPU operation (Res_2) is out. We mark I_2's ML 1 complete. Since both MLs of I_2 are complete, we update I_2's destination $R2$ with Res_2. Finally, in cycle 5 (Figure 3.11f), 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.

3.6 EXPERIMENTAL SETUP

We implement SAVE in the Sniper multicore simulator [79]. Table 3.1 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. We model the latency and execution ports of common instructions [80]. 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.

Table 3.1: 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-cache	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	1.375MB/core, shared, non-inclusive, 11-way, SRRIP, NUCA
NoC	2D-mesh, XY routing, 2-cycle hop
Memory	119.2GB/s BW, 6 channels, 50ns latency

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 [81]. Because one 512-bit VPU is broken down into two 256-bit units when executing AVX2 code [50], 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.

With the above configurations, we list SAVE's storage overhead in Table 3.2. We also model the leakage power and access energy of the broadcast cache (B$) configurations using CACTI 7.0 [82] at the 22nm process.

Table 3.2: 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

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 [83] (formerly MKL-DNN), a state-of-the-art AVX-512 DNN library.

For CNNs, we choose ResNet-50 [43] and VGG16 [42] on ImageNet-1K [44]. Because VGG16's activation sparsity is high [16], 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) [56] 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 [84] 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 3.3 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.

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. [16]. 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 [18] with the hyperparameters from [21] that yields a 75.4% top-1 accuracy. We start pruning at

Table 3.3: 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	✓	✓	✓		✓	

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 [85].

Figure 3.12 presents the progression of activation sparsity during training. We omit GNMT since its activation sparsity is constantly 20%. Figure 3.13 shows the schedule of weight pruning. We assume that, without pruning, the weights are fully dense.

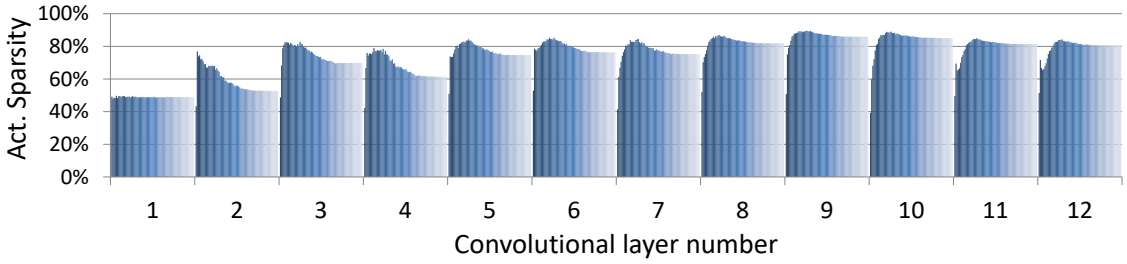
Next, for each layer, we simulate SAVE with both weight and activation sparsity of 0%-90% at 10% intervals, using a uniform random distribution. The result is a 2D surface of execution times with 100 different combinations of weight and activation sparsity. We warm up L3 with the output from the previous 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.

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 sparsity 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.

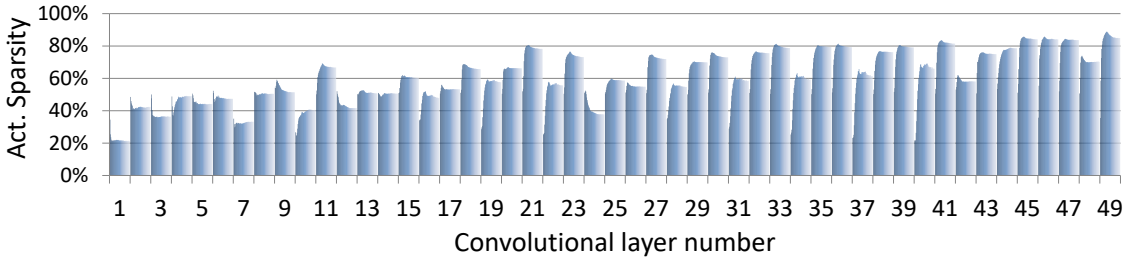
3.7 EVALUATION

3.7.1 Whole Neural Network Performance

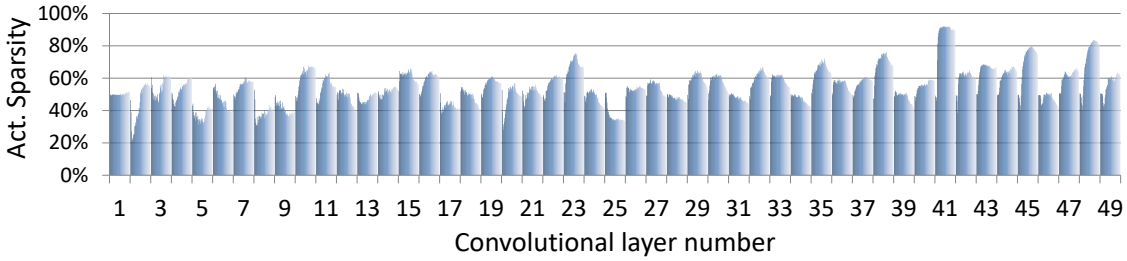
We assess SAVE's whole-network training and inference performance with all SAVE features. We configure the broadcast cache to store the data. Figure 3.14 shows the normalized



(a) Dense VGG16 training.



(b) Dense ResNet-50 training.

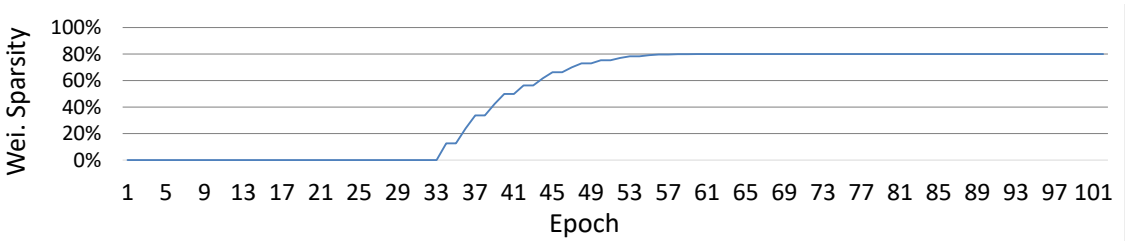


(c) ResNet-50 training with pruning.

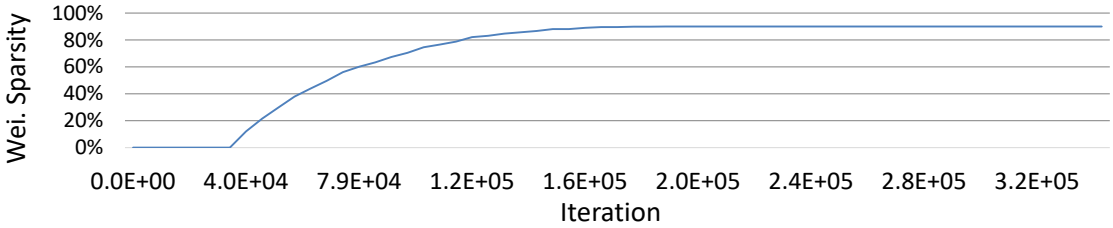
Figure 3.12: Activation sparsity during training. Each x-axis segment shows a layer. Within a segment, from left to right shows the sparsity from the first epoch to the last.

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 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.

Figure 3.14a and Figure 3.14b are for CNN inference and training respectively. They show times for dense VGG16, dense ResNet-50, and pruned ResNet-50 at realistic sparsity, each



(a) ResNet-50.



(b) GNMT.

Figure 3.13: Schedule of weight pruning.

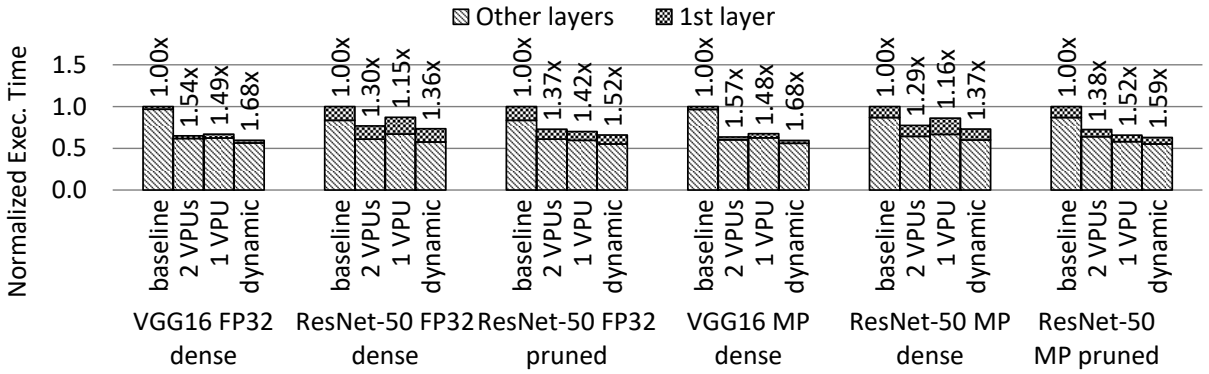
with FP32 and with mixed precision (MP). The bars are broken down into two or more categories. For inference and training, we separate the first layer because 1) it does not have sparse input activations, and 2) it does not compute the back-propagation of input. For training, we also distinguish between 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.

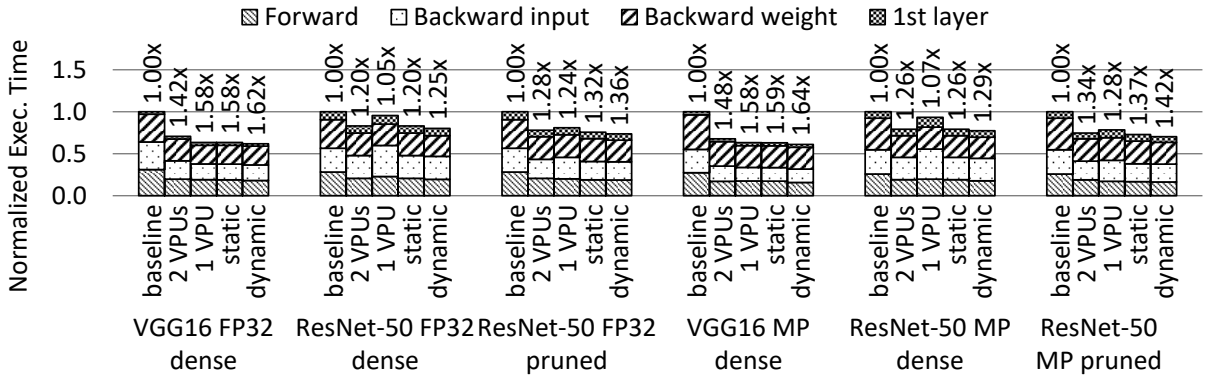
When SAVE uses a fixed number of VPUs, most workloads perform better with two VPUs. This is because, while many kernels have high sparsity and can benefit from using one VPU at higher frequency, some kernels have dense inputs, and thus prefer two VPUs. For example, the first layer in a CNN has no activation sparsity, and for training the dense ResNet-50, back-propagation of input has sparsity in neither weights nor activations due to Batch Normalization [56].

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.

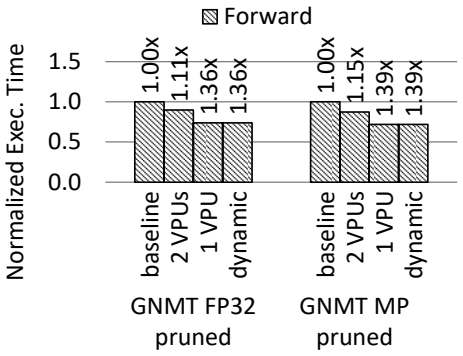
SAVE achieves higher speed-up on VGG16 than on ResNet-50. One reason is that, in VGG16, the first layer (which has no activation sparsity) contributes a smaller portion of the total execution time than in ResNet-50. Also, VGG16 does not incorporate Batch Nor-



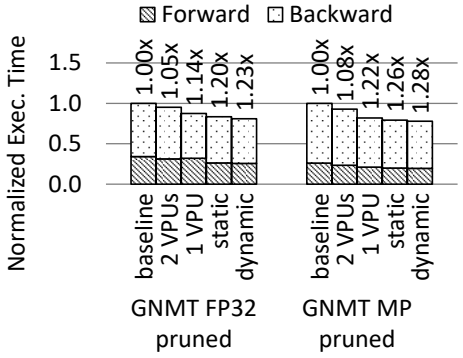
(a) CNN inference.



(b) CNN end-to-end training.



(c) GNMT inference.



(d) GNMT end-to-end training.

Figure 3.14: Execution time of all conv layers or LSTM cells in the studied networks at realistic sparsity, normalized to the baseline.

malization, 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.

Figure 3.14c and Figure 3.14d 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.

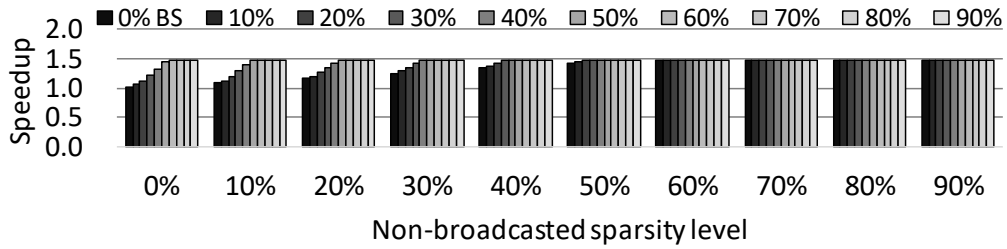
3.7.2 Boosting Frequency with Fewer VPUs

We study the effect of using one or two VPUs at different core frequencies. Figure 3.15 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.

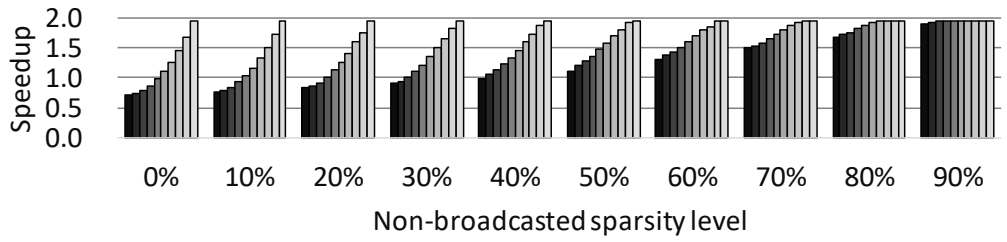
At high sparsity, the speedup reaches a ceiling because the execution becomes memory, frontend, or latency bound, depending on the kernel. Unless the execution is L3 or DRAM bound, higher core frequency usually helps. The speedup caps of the 93 studied kernels are considered in Figure 3.16, for FP32 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.

3.7.3 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



(a) Two VPUs at 1.7GHz core frequency.



(b) One VPU at 2.1GHz core frequency.

Figure 3.15: SAVE speedups on the mixed-precision forward propagation of ResNet2_2 with 1 or 2 VPUs.

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.

Figure 3.17 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

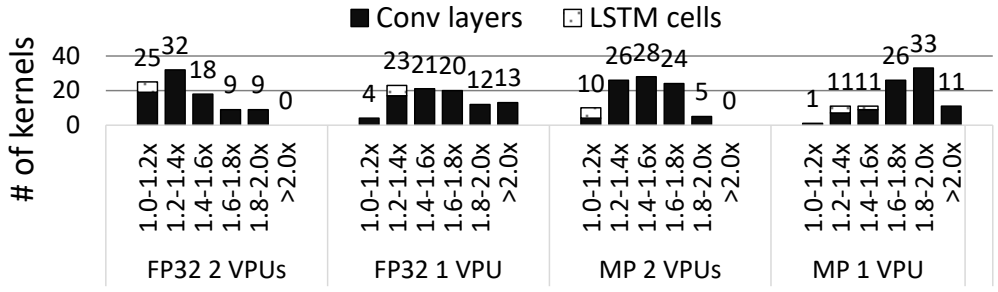


Figure 3.16: Histograms of the speedup caps. Each bar counts the number of kernels whose speedup caps are within a range.

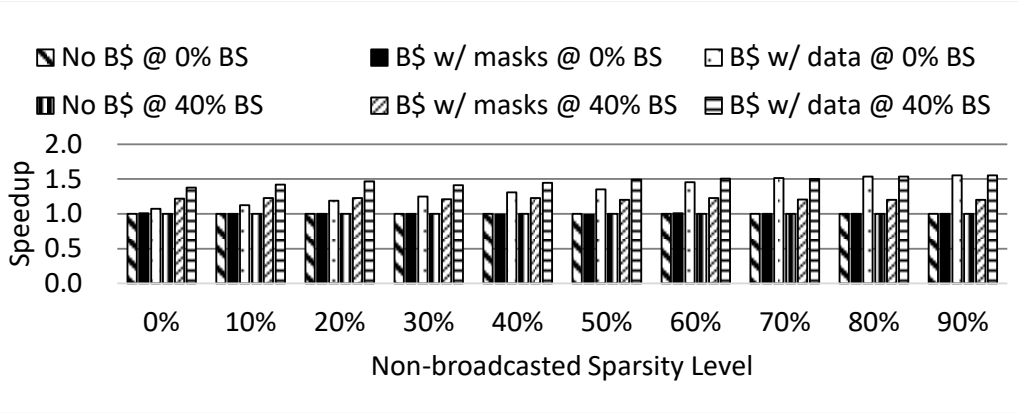


Figure 3.17: SAVE speedups with different B$ designs on the FP32 back-propagation of weights of ResNet3_2 with two VPUs.

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.

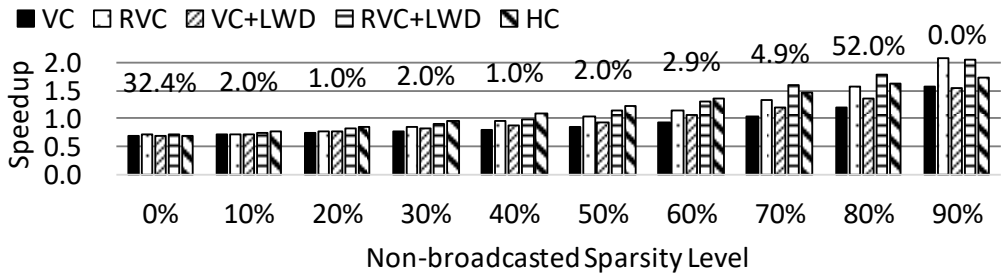
3.7.4 Techniques for Load-Balancing VPU Lanes

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) [80] as the cost of bubble collapsing/expanding, so we add 6 cycles to VFMA's latency.

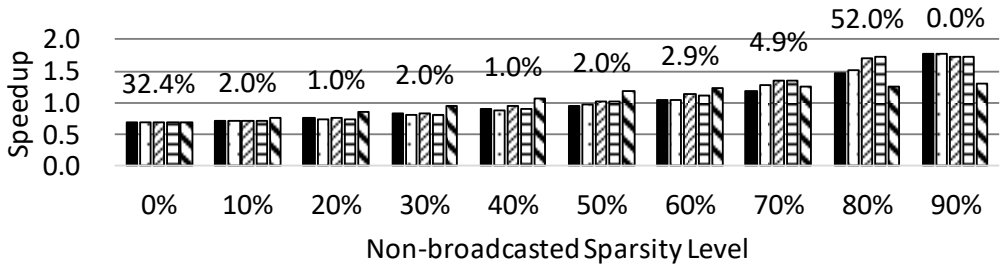
Figure 3.18 shows, for two kernels, the speedups of these techniques over the two-VPU baseline. We use NBS levels of 0%-90%, 0% BS, and one VPU. We choose two kernels of back-propagation of input in pruned ResNet-50 because this is the only case when NBS is present while BS is not (Table 3.3). To correlate the speedups with realistic sparsity, we list, above the bars, the percentage of training iterations where the NBS of the layer is $\pm 5\%$ of the sparsity written below the bars. Therefore, bars with higher numbers are more representative.

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.

Figure 3.18a 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



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



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

Figure 3.18: SAVE speedups with combinations of different techniques for load balancing VPU lanes.

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.

Figure 3.18b 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.

3.7.5 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. Figure 3.19 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.

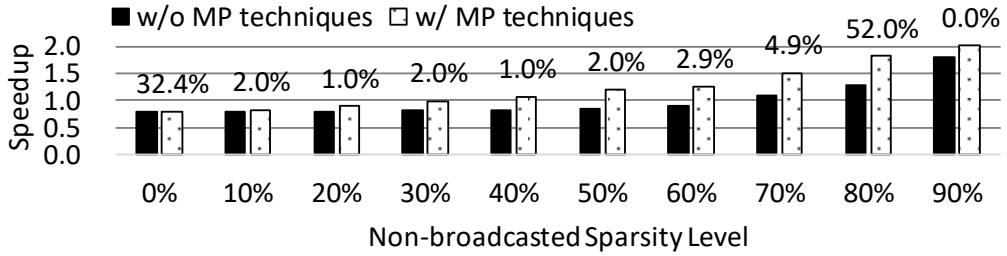


Figure 3.19: SAVE speedups on the mixed-precision ResNet4_1a back-propagation of input with SAVE, either with or without SAVE's mixed-precision (MP) technique.

3.8 RELATED WORKS

The industry has embraced mixed-precision FMA for DNN workloads. Google's TPU, NVIDIA's Tensor Core, and Intel's Cooper Lake all support mixed-precision DNN training. Henry et al. [76] suggest that BF16/FP16 systolic arrays may provide 8-32x more compute potential than a FP32 vector engine. Micikevicius et al. [86] demonstrate that mixed-precision DNN workloads on Volta GPU see a 2-6x speedup over FP32.

Model pruning [18, 19, 20] sparsifies the weights. Gale et al. [21] pruned weights to 95% with low accuracy loss. However, their unstructured-pruned models can perform badly on conventional parallel hardware. Structured pruning [22, 24] 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 [23] 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 [31] uses activation sparsity to skip ineffectual computations. Eyeriss [37] clock-gates hardware units when a zero is detected. It saves energy but not time. Cambricon-X [30] skips multiplications with pruned weights. EIE [29] exploits weight/activation sparsity with a compressed representation, but it is limited to matrix-vector multiplication. SCNN [28] accelerates convolutions with weight/activation sparsity. Proposals targeting CPUs and/or training are scarce.

SparCE [87] 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.

Prior works on reducing memory traffic based on sparsity are complementary to SAVE. In particular, ZCOMP [5] 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.[16] also use a similar compression method to reduce the PCIe traffic between GPUs and the CPU.

Control divergence induces ineffectual lanes in GPU SIMT hardware. Fung et al. [88] 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. [89] 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 [90].

3.9 CONCLUSION

This chapter presents 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.

CHAPTER 4: OPTIMIZING GRAPH NEURAL NETWORKS ON CPUS: REDUCING MEMORY BANDWIDTH NEEDS

4.1 INTRODUCTION

In this chapter, we tackle the memory performance issues caused by sparsity with software efforts. Traditional DNNs such as CNNs are only applicable to Euclidean data, e.g., an image represented as a grid of pixels. They lack the power to process non-Euclidean data, such as graphs [26]. Graphs model a set of objects in the form of vertices and their relationships in the form of edges. Many important types of data are represented as graphs. For example, a network of e-commerce products that are purchased together, the biologically meaningful associations between proteins, a citation network between papers, etc [27]. Graphs are often irregular. They can have a variable number of unordered vertices, and each vertex may link to a different number of neighbors. Consequently, operations like convolutions are difficult to apply in the graph domain [26]. Therefore, there is an increasing demand for a deep learning model that can operate on graphs. Graph Neural Networks (GNNs) have been proposed to fill this need. They are proven effective in social science [91, 92], physical systems [93], knowledge graphs [94], and other domains [95, 96].

CPUs are favorable platforms for GNNs. CPUs' memory capacity is orders of magnitude larger than GPUs'. While GPUs have tens of gigabytes of memory, CPUs can be equipped with terabytes of memory [7]. While larger memory benefits all types of DNNs, it is especially important for GNNs because real-world graphs often have millions to billions of vertices and edges, and each vertex and/or edge can attach hundreds to thousands of features. As a result, a multi-layer GNN may need hundreds of gigabytes of memory to operate on graphs of such a scale. Although techniques such as neighborhood sampling and vertex mini-batching have been proposed to cope with GPUs' limited memory capacity [91], these workarounds often reduce the accuracy of the network and introduce additional costly operations. Hence, a CPU has the advantage of being able to work with full-batches. This chapter aims to improve the performance of full-batch training and inference of GNNs on multi-core CPUs.

While traditional DNN workloads are usually regular and compute-intensive, GNNs are irregular and often memory-intensive, which is a result of the sparse connections in graphs. Therefore, GNNs pose distinct performance challenges compared with other DNNs. We profiled GNN workloads with a state-of-the-art GNN implementation [97] and found that the executions are heavily DRAM bandwidth bound. Therefore, alleviating DRAM bandwidth pressure is key to improving performance of GNN workloads on CPUs.

We propose techniques to tackle the DRAM bandwidth problem. A GNN layer is com-

posed of a memory-intensive aggregation phase, where each vertex collects information from its neighbors, and a compute-intensive update phase, where a deep learning operator such as a fully-connected layer processes the collected information [98]. Our first step is to implement an efficient parallel vectorized aggregation primitive. Then, we overlap the memory movement and the compute by fusing the two phases. The next optimization is based on the observation that the vertex features in the hidden GNN layers often contain a moderate amount of zeros due to the use of ReLU and dropout. We reduce memory traffic by (de)compressing the sparse features before writing to and reading from memory. Finally, we devise a simple yet effective algorithm to improve temporal locality by rearranging the processing order of the vertices.

We make the following contributions. First, we characterize GNN workloads on CPUs and identify that DRAM bandwidth is a major bottleneck. Second, we propose optimizations to relieve the DRAM bandwidth pressure, which are proven effective. We apply the optimizations to both inference and training. Third, we validate our approach with full-batch computation on medium to large scale graphs up to 111 million vertices and 1.6 billion edges, on a 22-core server CPU. Our implementation outperforms a state-of-the-art GNN implementation on layers from popular GNN models by 1.72-1.94x on inference and 1.60-2.63x on training .

4.2 BACKGROUND AND MOTIVATION

4.2.1 Graph Neural Networks

GNNs have become popular tools to process non-Euclidean data such as graphs, which has proven to be a hard task for other types of DNNs [26]. We first present the general formulation of GNNs. Table 4.1 describes the notations used in this chapter.

Table 4.1: List of the symbols.

	Description		Description
$\mathcal{G}$	graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$	$\mathcal{V}$	vertices of G
$\mathcal{E}$	edges of $\mathcal{G}$	D_v	degree of vertex v
$\mathcal{N}(v)$	all neighbors of vertex v	$\mathcal{S}(v)$	sampled subset of $N(v)$
$e_{u,v}$	edge between vertex u and v	$\mathbf{A}$	adjacency matrix
K	number of layers	H	vertex feature vector length
$\mathbf{h}$	feature matrix	$\mathbf{h}_v$	feature vector of vertex v
$\mathbf{a}$	aggregation feature matrix	$\mathbf{a}_v$	aggregation feature vector of vertex v
$\mathbf{W}$	update weight matrix	$\mathbf{b}$	update bias vector
ψ	feature processing function		

A deep GNN typically consists of K layers. Each layer contains an *aggregation* phase and an *update* phase [98]. In the aggregation phase of layer k, for each vertex v, we first gather the feature vector of its neighbors $\mathcal{N}(v)$ from layer $k-1$. Then, we reduce the gathered feature vectors and v's own feature vector from layer $k-1$ to create the aggregation feature vector $\mathbf{a}_v^k$ through an aggregation function.

$$\mathbf{a}_v^k = \text{AGGREGATE}(\mathbf{h}_u^{(k-1)} \mid \forall u \in \mathcal{N}(v) \cup \{v\}) \tag{4.1}$$

In the subsequent update phase, we apply an update function to transform each $\mathbf{a}_v^k$ to an output feature vector $\mathbf{h}_v^k$ for this layer.

$$\mathbf{h}_v^k = \text{UPDATE}(\mathbf{a}_v^k) \tag{4.2}$$

After K layers, each vertex's feature vector is a function of its neighbors up to K hops away.

Furthermore, in order to reduce both the computational complexity and the memory footprint, some networks *sample* a subset of each vertex's neighbors before the aggregation phase. In sampling, we randomly select up to some pre-determined number of neighbors for each vertex.

$$\mathcal{S}(v) = \text{SAMPLE}^k(\mathcal{N}(v)) \tag{4.3}$$

The sampling step is essential for executing GNNs with large input graphs on memory-limited devices such as GPUs and accelerators [91, 99]. In order to fit the footprint in device memory, one may first divide the graph into mini-batches of vertices. Then, one may perform a breadth-first search (BFS) to find the K-hop neighborhood of each vertex in a mini-batch. Finally, only the input feature vectors of the neighborhoods need to be transferred to the device memory. With a fixed sample size, the upper bound of the working set of each mini-batch is predetermined.

Different GNN models may adopt various aggregation and update functions. Table 4.2 presents two popular GNN models — the Graph Convolutional Network (GCN) [92] and GraphSage with the mean aggregator [91].

Table 4.2: Example GNN models

Model	Aggregation	Update
GCN	$\sum \mathbf{h}_u^{(k-1)} / \sqrt{D_v \cdot D_u} \mid \forall u \in \mathcal{N}(v) \cup \{v\}$	$ReLU(\mathbf{W}^k a_v^k + \mathbf{b}^k)$
GraphSAGE	$\sum \mathbf{h}_u^{(k-1)} / (D_v + 1) \mid \forall u \in \mathcal{N}(v) \cup \{v\}$	$ReLU(\mathbf{W}^k a_v^k + \mathbf{b}^k)$

Both models use the same update function: a fully-connected (FC) layer activated with ReLU. Their difference lies in the aggregation function. In GCN, each vertex first normalizes each neighbor's and its own feature vectors. It then sums the normalized feature vectors. In GraphSAGE, each vertex takes the element-wise average of its neighbors' and its own feature vectors. Despite the difference, the aggregation functions of the two models both gather each vertex's neighbors' feature vectors, process each gathered feature vector with a function ψ, and finally perform a reduction. Both models can adopt sampling by replacing $\mathcal{N}(v)$ with $\mathcal{S}(v)$ in aggregation.

Training GNNs follows the same principles as training any other type of DNN. The training process iteratively updates the trainable parameters (e.g. $\mathbf{W}$ and $\mathbf{b}$ in the two example models) with a loop of the forward pass and the backward pass. The forward pass computes the outputs with the current parameters, compares them with the ground truth, and produces errors. The backward pass propagates error gradients with the chain rule and updates the parameters accordingly.

Many popular GNN models are relatively shallow. Deeper GNNs suffer from the vanishing gradient problem, where back-propagating through the network causes over-smoothing that converges the features of the vertices to the same value. Fortunately, recent works tackle the problem by applying various techniques such as residual connections to deep GNNs and prove that deeper networks can outperform shallow ones [100, 101, 102].

As discussed in Section 1.2.3, the adjacency matrices of the input graphs are often stored in a compressed format due to the connections being highly sparse. In addition, the feature vectors may be moderately sparse from the use of ReLU and/or dropout. However, the feature sparsity does not justify using a compressed format, so the feature vectors are kept in a dense representation.

As GNNs have gained popularity, the community has developed GNN-specific frameworks on top of general DNN frameworks such as Tensorflow [48] and PyTorch [49]. Widely adopted GNN frameworks include PyTorch Geometric (PyG) [103] and the Deep Graph Library (DGL) [104].

4.2.2 GNN on CPUs

Advantages

Although the community has started using GPUs [105] and has proposed several accelerators [32, 33, 106] for GNNs, CPUs are often used for reasons listed in Section 1.1. Among CPUs' advantages, the crucial one for GNNs is CPUs' high memory capacity. Real-world

graphs can have millions to billions of vertices [107, 108], so the footprint of their feature matrices may occupy tens to hundreds of gigabytes, which far exceeds GPUs and accelerators' memory capacity. In order to run large input graphs those devices, one may use sampling and mini-batching described in Section 4.2.1 to confine the working sets. However, sampling and mini-batching have drawbacks. First, the size of the K-hop neighborhood grows exponentially with the number of layers K. With enough layers, the sub-graph used by a mini-batch can span all connected components that contain the vertices in the mini-batch. Under the circumstance, GPU users may be forced to use a tiny mini-batch size to fit the working sets in the device memory, which vastly under-utilizes the compute capacity [100]. Second, sampling may degrade the network accuracy [91, 105]. Third, both sampling and the additional BFS introduced by mini-batching induce significant overhead. We profiled the training of a sampled GraphSAGE on a GPU with different mini-batch sizes. Figure 4.1 shows the breakdown of the training epoch time. The numbers labeled in the figure are the time spent on sampling plus BFS and the GNN layer computation respectively. The figure reveals that the sampling and BFS astoundingly contribute to over 90% of the total training time, and the training time increases significantly as the mini-batch size shrinks.

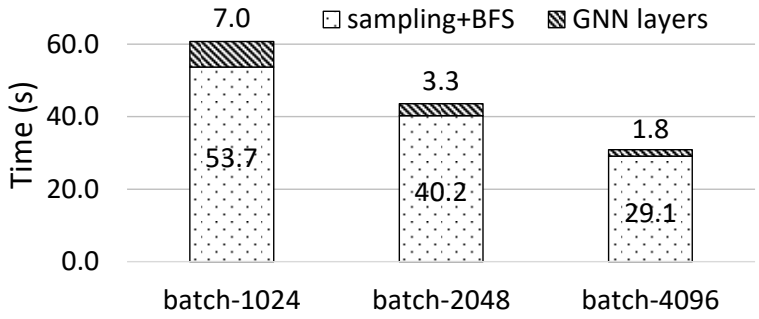


Figure 4.1: Training epoch time breakdown of a sampled GraphSAGE on a GPU with different mini-batch sizes.

While GPUs and accelerators are equipped with tens of gigabytes of memory, CPUs' memory capacity has reached the order of terabytes [7, 8]. This enables full-batch training without sampling for large graphs as well as facilitates wider and deeper network structure.

Challenges

GNN computations on CPUs have room for improvement. We profiled GNN trainings on a CPU with DGL and discovered that the aggregation phase typically constitutes over 80% of

the execution time. Since the aggregation performs a simple reduction for each vertex after gathering its neighbors' feature vectors, it is memory-intensive. Figure 4.2 is a breakdown of the pipeline slots either doing useful work or wasted on different bottlenecks during a full-batch training of GraphSAGE on a CPU. The breakdown shows that only 10.1% of the pipeline slots attribute to useful work. The memory sub-system is a severe bottleneck. 61.7% of the pipeline slots are stalled due to demanded memory load and stores. 53% of the clock cycles are stalled due to approaching DRAM bandwidth limit. Furthermore, the profiling reveals that the L1 data cache (L1D) fill buffer is full almost 100% of the time, hinting that L1 misses are often satisfied from deep in the memory hierarchy, such as from DRAM. Therefore, reducing DRAM bandwidth pressure is a key task to optimize GNN workloads.

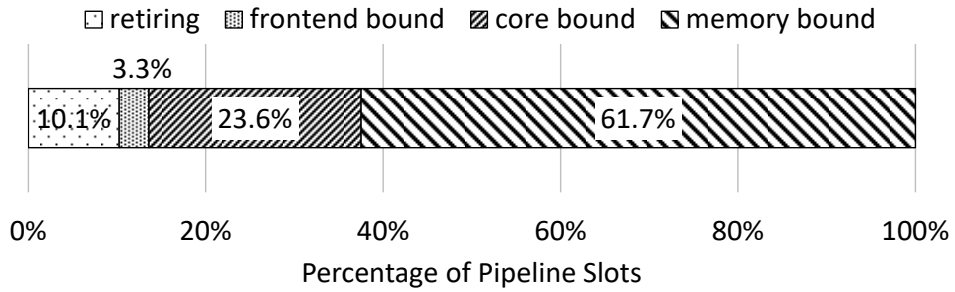


Figure 4.2: Breakdown of the pipeline slots spent on retiring micro-ops or stalled by different bottlenecks during a full-batch training of GraphSAGE on a CPU.

This chapter aims to improve the performance of full-batch training and inference on multi-core single instruction multiple data (SIMD) CPUs without sampling. For simplicity, we focus on GCN and GraphSAGE.

4.3 OPTIMIZATION TECHNIQUES

In this section, we present our techniques to optimize GNN execution on multi-core CPUs. The optimizations in Sections 4.3.1-4.3.3 benefit both training and inference. Compared with inference, training has distinct characteristics. Sections 4.3.4 discusses techniques that further optimize training.

4.3.1 Parallel Vectorized Aggregation

Section 4.2.2 discusses that the aggregation phase dominates the execution time of a GNN layer. Therefore, we first focus on the implementation of an efficient aggregation primitive.

In aggregation, each vertex, v, first gathers the feature vectors from $u \in \mathcal{N}(v) \cup \{v\}$, then performs an element-wise reduction, and finally writes to its aggregation feature vector, $\mathbf{a}_v$. This means that, during aggregation, all working sets but $\mathbf{a}^k$, are read-only. Therefore, we output-parallelize the aggregation by letting threads compute different partitions of $\mathbf{a}^k$. Output-parallelization avoids race conditions and thus requires no synchronization among cores.

Algorithm 4.1 shows our parallel vectorized aggregation. In Line 1, we divide $\mathcal{V}$ into chunks of T vertices. Each parallel task computes the aggregation feature vectors of a chunk. The processing time of each parallel task correlates with the degrees of the vertices in the chunk. Because real-world graphs often follow a power law distribution [97], the degrees can vary significantly. Consequently, each parallel task may take notably different time to finish. To balance the load among threads, we schedule the parallel tasks with OpenMP's *dynamic* scheduler.

Algorithm 4.1: Parallel vectorized aggregation.

input : graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, input feature matrix $\mathbf{h}^{k-1}$, task size T, vector length V, prefetch distance D, reduction operator $\oplus$, feature processor ψ
output : aggregation feature matrix $\mathbf{a}^k$

```
1 for i = 0 to |V| − 1 step T in parallel do
2     for j = 0 to T do
3         v = V_{i+j}
4         a_v^k = {0}
5         for u ∈ N(v) ∪ {v} do
6             for m = 0 to |a_v^k| step V do
7                 a_{v[m:m+V−1]} = a_{v[m:m+V−1]} ⊕ ψ(h^{k−1}_{u[m:m+V−1]})
8         v' = V_{i+j+D}
9         PREFETCH(h_{u'}^{(k−1)} | ∀u' ∈ N(v') ∪ {v'})
```

Each parallel task iterates through the vertices in the assigned chunk and performs aggregation on each one. Lines 4-7 perform aggregation on a vertex, v. Because the feature vector of each vertex in GNN workloads often has hundreds to thousands of elements, we vectorize the feature gathering, processing, and reduction (Line 7).

After the aggregation of each vertex, we prefetch the features needed by a later aggregation with a distance, D (Line 9). Since the execution is mainly DRAM bandwidth bound, the L1D fill buffer is often full of pending misses. In such cases, adding excessive software prefetch can actually degrade the performance. Through experiments, we determine that prefetching the first cache line of each feature vector yields optimal results.

We use a JIT assembler to generate the aggregation kernel. Dynamically generated DNN kernels often outperform statically compiled ones. By tailoring the kernel to the layer specification, the former can use registers more efficiently by using layer-specific constants. It can also avoid overhead such as unnecessary boundary checking. Moreover, one only needs to generate the code once during the entire training/inference session because the code is only specified with the model but not the data. Hence, the overhead of dynamic code generation is fully amortized [35, 46].

4.3.2 Layer Fusion

As discussed in Section 4.2.2, the two phases in a GNN layer express opposite characteristics: the aggregation phase is irregular and memory-intensive; the update phase is regular and compute-intensive. Conventionally, a GNN layer first aggregates all vertices, placing little pressure on dense compute hardware but lots on the memory hierarchy, and then updates the vertices, flipping the hardware utilization.

Layer fusion is a known technique to optimize DNN executions. Common fusion schemes include fusing the compute layer (e.g., a convolutional layer) and its subsequent element-wise activation function (e.g., ReLU) in order to avoid redundant read-write of the compute layer's output. We propose to apply layer fusion to overlap the memory movement in aggregation with the compute in update.

Algorithm 4.2 describes our layer fusion scheme. Like in Algorithm 4.1, we parallelize the fusion algorithm by partitioning the output working sets, which are both the aggregation feature matrix, a^k, and the output feature matrix, h^k. Each parallel task performs a fused aggregation-update on a chunk of $T \cdot B$ vertices in Lines 2-10. It iterates through the assigned vertices with a block size, B (Line 2). In each tiled iteration, it aggregates a block of T vertices in Lines 3-7 and then update them in Lines 8-10. The AGGREGATE function is equivalent to Lines 4-7 in Algorithm 4.1.

The block size, B, controls how finely we interleave aggregation and update operations. For sufficiently small B, we can rely on the out-of-order hardware to "unroll" the j loop for us, and overlap a current vertex's compute-heavy update with loads for later vertex's aggregations. In practice, small B implies less reuse of the weight matrix in update, so the execution can suffer from L1 cache thrashing if the weights does not fit in the L1 cache. Therefore, we focus on coarser-grained interleaving with larger B.

For coarse-grained interleavings, we still overlap memory and compute operations, in two ways. First, within a single thread, the prefetch operations in the tiled loop iteration j *may*, depending on the prefetch distance, D, prefetch for the next loop iteration $j + 1$. Therefore,

Algorithm 4.2: Fused aggregation and update.

input : graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, input feature matrix $\mathbf{h}^{k-1}$, task size T, block size B, prefetch distance D

output : output feature matrix $\mathbf{h}^k$

```
for i = 0 to |V| − 1 step T · B in parallel do
    for j = 0 to T · B − 1 step B do
        for m = 0 to B − 1 do
            v = V_{i+j+m}
            a_v^k = AGGREGATE(h_u^{(k−1)} | ∀u ∈ N(v) ∪ {v})
            v' = V_{i+j+m+D}
            PREFETCH(h_{u'}^{(k−1)} | ∀u' ∈ N(v') ∪ {v'})
        for m = 0 to B − 1 do
            v = V_{i+j+m}
            h_v^k = UPDATE(a_v^k)
```

during an update phase, the hardware prefetches features needed by the next aggregation phase. Second, when we consider all of the threads, aggregation and update operations may happen simultaneously for disjoint subsets of threads.

Figure 4.3 illustrates how the two phases on three cores overlap. Critically, we do not synchronize threads within the parallel loop; thus, we do not force or encourage threads to be out of phase with respect to each other, but expect this to happen naturally. In the figure, the aggregation phases on different cores take varied latency, so the subsequent updates start at different times and can overlap with the aggregation phases on other cores.

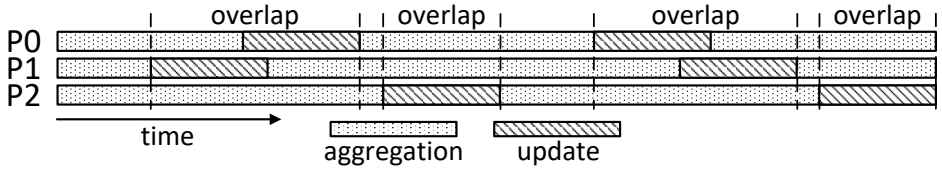
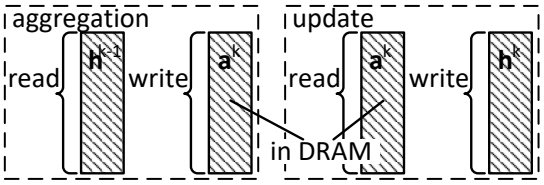
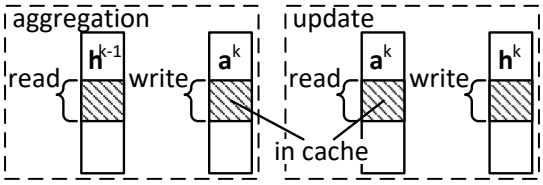


Figure 4.3: Aggregation and update on different cores can overlap without synchronizations.

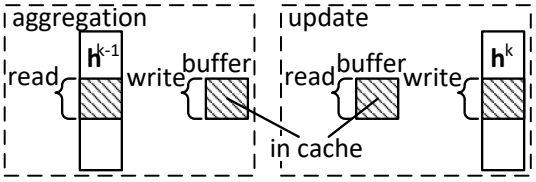
Besides compute-memory overlap, layer fusion also reduces DRAM traffic and/or footprint, as shown in Figure 4.4. Without fusion, when a^k is much larger than the cache, the aggregation phase writes the entire $\mathbf{a}^k$ to DRAM, and the subsequent update phase fetches $\mathbf{a}^k$ from DRAM again (Figure 4.4a). In contrast, the fused implementation produces less DRAM traffic in both training and inference. In each tiled loop iteration j, the aggregation phase produces a block of $\mathbf{a}^k$, which is then consumed by the subsequent update phase. With a proper B, the $\mathbf{a}^k$ block resides in cache between the two phases (Figure 4.4b). Additionally,



(a) The basic implementation writes the whole $\mathbf{a}^k$ to DRAM in the aggregation and reads it back in the update.



(b) The fused training kernel keeps the $\mathbf{a}^k$ working set in the cache between an aggregation and the subsequent update iteration.



(c) The fused inference kernel uses a single buffer to hold the $\mathbf{a}^k$ block used by an aggregation and the subsequent update iteration.

Figure 4.4: Layer fusion produces less main memory traffic and/or footprint than the basic implementation.

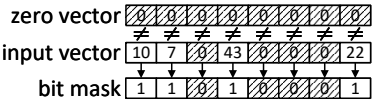
in inference, the fused implementation does not need to keep the entire $\mathbf{a}^k$ for all vertices. Instead, we only need a reusable buffer to hold the block of $\mathbf{a}^k$. We can discard the buffer's content after an update phase, and use it for the next $\mathbf{a}^k$ block (Figure 4.4c). In training, the entire $\mathbf{a}^k$ is needed for back-propagation, so this footprint reduction is inapplicable.

4.3.3 Feature Compression

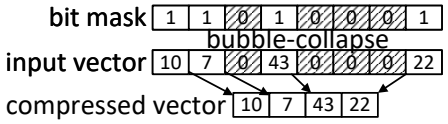
The feature matrix, h, may be moderately sparse because of ReLU and/or dropout. Since the aggregation draws so much DRAM bandwidth, we can improve performance by avoiding transferring zero-valued elements in the feature vectors. However, as discussed in Section 1.3, using a traditional compressed format such as CSR would be counterproductive for h, so we need a more space/time-efficient compression technique for the purpose.

Modern CPU vector extensions such as x86's AVX-512 provide mask-based (de)compression instructions. The compression instruction takes a vector and a bit mask as input. It uses the set-bits in the mask to select the active elements from the source vector to compress into a contiguous destination vector. The decompression instruction uses a mask to expand elements from a contiguous input vector to a sparse destination vector. Figure 4.5 shows how we can utilize the above instructions to compress a sparse vector into contiguous memory and later restore it back to the sparse vector. The example assumes the hardware vector

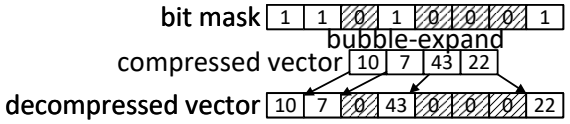
length $V = 8$.



(a) Compression step 1: produce a bit mask to mark non-zero elements using vector comparison.



(b) Compression step 2: bubble-collapse the input vector according to the set-bits in the mask.



(c) Decompression: bubble-expand the compressed vector according to the bit mask.

Figure 4.5: Examples of a mask based (de)compression.

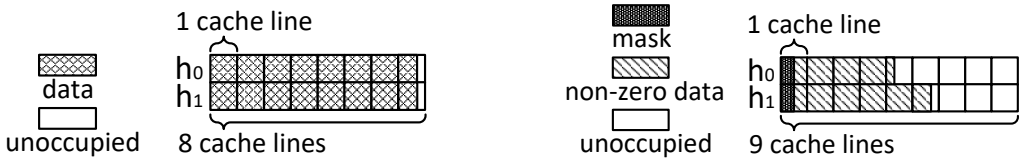
To compress a sparse vector, the first step is to compare it with a vector of zeros, using a vector compare instruction, to produce a mask (Figure 4.5a). Each of the set bits in the mask indicates the position of a non-zero element in the sparse vector. The second step executes the compression instruction with the mask (Figure 4.5b). It collapses all bubbles in the original vector and compacts it to a dense vector. We need to store both the mask and the compacted vector for later decompression. To restore the sparse vector from the compacted form, we execute the decompression instruction with the mask generated during compression (Figure 4.5c). This expands the dense vector back to the sparse vector by inserting zeros at the positions indicated by the mask.

The only meta data of the compression scheme is the mask. Each feature element requires one bit, so for example, if each feature has 32 bits, the space overhead is $1/32 = 3.125\%$ of the uncompressed feature matrix, regardless of the sparsity level. For moderate sparsity, as we expect, this overhead is small. For example, when 32-bit features are 50% sparse, the traffic from reading/writing the features is efficiently reduced by $50\% - 3.125\% = 46.875\%$.

While one could reduce the memory footprint of h via compression, we do *not* do this. Fast random accesses to vectors are critical, and variable-sized vectors would harm this. Therefore, we maintain a constant-sized storage for each vector, and simply use only a fraction of it, depending on its sparsity. Our purpose in compressing feature vectors is purely to save DRAM bandwidth when reading and writing them, which is achieved with this scheme.

Figure 4.6 shows examples of uncompressed and compressed storage. Each example contains two feature vectors stored in row-major order. Figure 4.6a shows the uncompressed

data layout. In the example, each vector is just under 8 cache lines in length. For higher access performance, we align each feature vector to a cache line boundary; this may leave some unoccupied space in the last cache line of each feature vector.



(a) Uncompressed. (b) Compressed.

Figure 4.6: Example data layouts of the feature vectors before and after the compression.

Figure 4.6b shows the compressed data layout. Each vector starts with its mask. We allocate enough space to hold the mask and the uncompressed features for each feature vector, to ensure dense vectors will fit. In the example, this is 9 cache lines. Combining the masks and the compressed features into the same working set is superior to storing them separately. During decompression, the masks are loaded first; thus, with combined storage, hardware prefetchers can help stream the features as the masks are being loaded.

We confirm that compressing the features at a realistic sparsity level can successfully reduce the DRAM bandwidth pressure. With compression, we have more free L1D fill buffer entries during the execution. Consequently, we are able to increase the amount of software prefetch to further improve the performance.

4.3.4 Temporal Locality Improvement

We can also reduce DRAM bandwidth pressure by reducing the reuse distance of each feature vector. In aggregation, each vertex gathers its neighbors' feature vectors. If we process two vertices with a common neighbor close together in time, that common neighbor's feature vector will have a small reuse distance, and is thus likely to be cached for the second access. Thus, the processing order of vertices influences the temporal locality in GNN workloads.

We access a given vertex's feature vector a number of times equal to the degree of the vertex. Thus, to minimize overall reuse distance, we choose to prioritize the ordering of accesses to high-degree vertices. Algorithm 4.3 describes a method to do this. In the algorithm, L is a collection of $|\mathcal{V}|$ sets. Each set $\mathcal{L}_v$ is intended to contain vertices that read vertex v's feature vector during aggregation. By building up the sets of high degree vertices at the expense of low degree vertices, we decrease reuse distance of high degree vertices. Each

set is initially empty (Lines 1). We populate the sets using a greedy algorithm (Lines 2-7). For each vertex v, we assign it to $\mathcal{L}_{u'}$, where u' is the vertex with the highest degree among $\mathcal{N}(v) \cup \{v\}$. After all vertices are assigned, we generate a processing order $\mathbf{M}$ in Lines 8-12. During aggregation, vertex $\mathbf{M}_{i+1}$ is processed after $\mathbf{M}_i$.

Algorithm 4.3: Compute a processing order of vertices to improve the temporal locality in aggregation.

```
   input  : graph G = (V, E)
   output : processing order M
 1 L_v = ∅ | ∀v ∈ V
 2 for v ∈ V do
 3 |   u' = v
 4 |   for u ∈ N(v) do
 5 |   |   if D_u > D_u' then
 6 |   |   |   u' = u
 7 |   L_u' = L_u' ∪ {v}
 8 i = 0
 9 for v ∈ V do
10 |   for u ∈ L_v do
11 |   |   M_i = u
12 |   |   i = i + 1
```

The time complexity of the algorithm is $O(|\mathcal{E}| + |\mathcal{V}|)$. For inference, the overhead can exceed the benefit. However, for training, we reuse graphs, so the cost of the algorithm is easily amortized.

4.4 EXPERIMENTAL SETUP

We implement our aggregation kernel with the xbyak JIT assembler [54]. We use GEMM libraries to execute the update phase. We use Intel MKL for the implementation without layer fusion. With layer fusion, because the matrix multiplication in each update phase has a small size, we use libxsmm [109], which is optimized for small matrix multiplications.

Our baseline uses the state-of-the-art DistGNN [97] for the aggregation and MKL's GEMM for the update. DistGNN employs optimizations such as cache blocking and vectorization. It has recently been incorporated in DGL. We refer to this baseline as *DistGNN*. Because the aggregation can be computed with sparse-dense matrix multiplication (SpMM), we also compare our approach with an implementation that uses MKL's SpMM for the aggregation and MKL's GEMM for the update. We call this implementation *MKL*.

The platform for the evaluation is a 28-core Intel Cascade Lake server CPU with AVX-512 vector extensions. Each core has a 32KB L1 data cache, a 1MB private L2 cache, and a 1.375MB slice of a non-inclusive shared L3 cache. The CPU is clocked at 2.7GHz without dynamic frequency scaling. The maximum DRAM bandwidth is 140.8GB/s. We disable simultaneous multithreading (SMT) and run 28 threads.

We evaluate our approach with the full-batch, non-sampled training and inference of the GNN layers from GCN and GraphSAGE. We experiment with 4 input graphs from medium to large size. Table 4.3 lists the detailed sizes of the graphs.

Table 4.3: List of dataset configurations

Name	$\|\mathcal{V}\|$	$\|\mathcal{E}\|$	H_{input}	H_{hidden}
ogbn-products [27]	$2,449,029$	$123,718,280$	100	256
wikipedia [110]	$3,566,907$	$45,030,389$	128	256
ogbn-papers [27]	$111,059,956$	$1,615,685,872$	256	256
twitter [108]	$61,578,415$	$1,468,364,884$	256	256

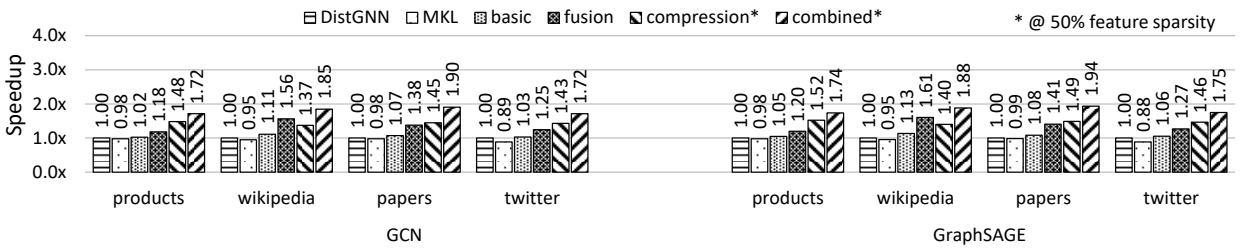
All graphs except *ogbn-products* are directed. For *ogbn-products*, the number of edges in the table is twice its number of undirected edges. *ogbn-products* and *ogbn-papers* have predefined input feature vector lengths, H_{input}. However, *wikipedia* and *twitter* do not attach vertex features. We synthetically set their H_{input} to 256. For each dataset, we set the hidden features vector length, H_{hidden}, to 256.

4.5 EVALUATION

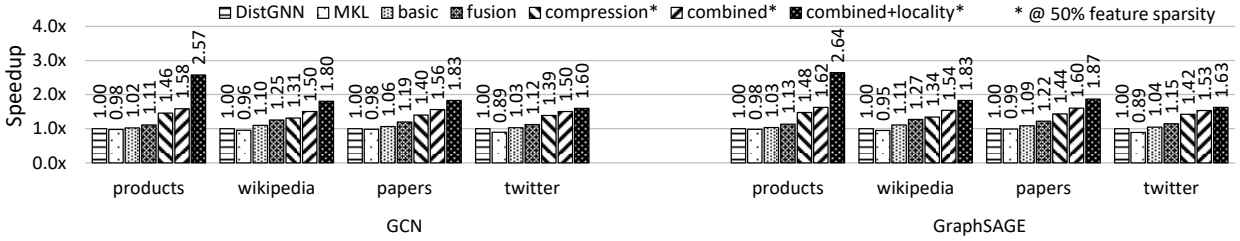
4.5.1 Performance

We evaluate the performance of our implementations with different techniques enabled. Figure 4.7 shows the speedup from our implementations and *MKL* over the *DistGNN* baseline. In the figure, besides *MKL*, *basic* is our algorithm from Section 4.3.1 for aggregation and the MKL's GEMM for update. *fusion* is the layer-fused implementation from Section 4.3.2. *compression* is *basic* plus feature vector compression from Section 4.3.3. *combined* incorporates both *fusion* and *compression*. For training only, *locality* is the performance of *combined* on the input graphs that have been optimized for locality from Section 4.3.4.

compression, *combined*, and *locality* all incorporate feature compression. In Figure 4.7, we report their performance when operating on 50% sparse features. This is conservative. We profiled a 20-epoch training of a 3-layer GraphSAGE on *ogbn-products*. Figure 4.8 presents the level of sparsity in the features as the training progresses. We see that ReLU sparsifies



(a) Inference.



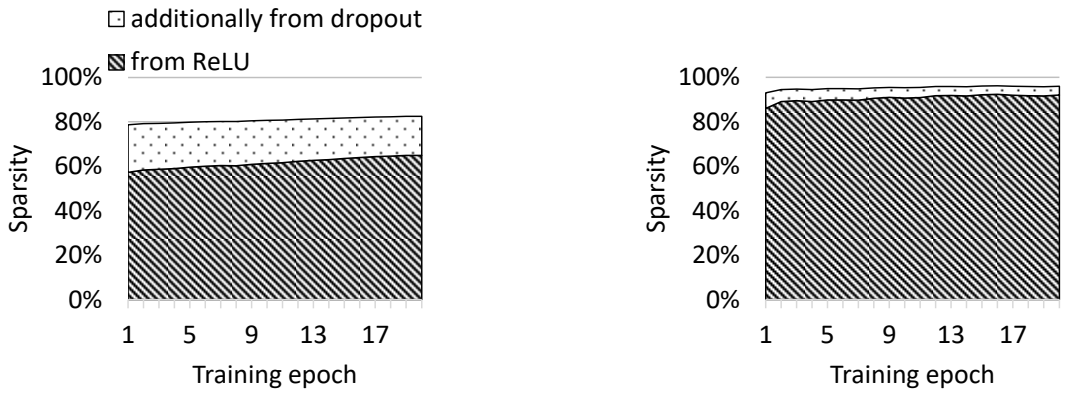
(b) Training.

Figure 4.7: The speedup from MKL and our implementations with different techniques over the DGL-DistGNN baseline.

the input features to the second layer by over 60%, and dropout further sparsifies them to over 80%. The sparsity of the input features to the third layer is even higher, reaching over 90%. Therefore, feature compression may improve the performance more than reported.

Our implementations outperform both the baseline and *MKL* significantly. Figure 4.7a shows the speedup in inference. Performance here is determined primarily by memory behavior, which is the same for the two GNNs, so we see similar performance results on the two models. *MKL* is slightly slower than the baseline. *basic* already outperforms the baseline on all datasets. The other variations of our implementation are faster than *basic* on all datasets. *wikipedia* benefits more from layer fusion than from feature compression, while the other datasets benefit more from feature compression than from layer fusion. We will explain why layer fusion accelerates *wikipedia* more in Section 4.5.3. The combination of the techniques always performs the best. Compared with the baseline implementation of GCN (GraphSAGE), *combined* is 1.72-1.90x (1.74-1.94x) faster.

Figure 4.7b presents the speedup in training. The execution time of each implementation includes both forward and the backward propagation. Forward propagation is similar to inference, except when layer fusion is applied, we do not reduce the footprint of $\mathbf{a}^k$ as discussed in Section 4.3.2. Backward propagation computes the gradients of $\mathbf{h}^{k-1}$, $\mathbf{a}^k$, $\mathbf{W}^k$, and $\mathbf{b}^k$. It has one more GEMM than the forward propagation. In training, *MKL* is again slightly slower than the baseline. *basic* outperforms the baseline on all datasets. Layer



(a) Layer 2 input feature sparsity.

(b) Layer 3 input feature sparsity.

Figure 4.8: Feature sparsity during the training of GraphSAGE.

fusion is less effective in training than in inference because in training, it does not shrink the memory footprint of the aggregation feature vectors, and therefore being less effective in reducing the memory traffic. Nevertheless, both *fusion* and *compression* perform better than *basic* in all cases. By combining layer fusion and feature compression, *combined* outperforms the baseline in GCN (GraphSAGE) training by 1.50x-1.58x (1.53x-1.62x).

The locality optimization further improves training performance. *locality* uses the same kernel as *combined* but with pre-processed inputs, where the order of vertices is changed to improve cache hit rates. The speedup from *locality* over the baseline in GCN (GraphSAGE) training reaches 1.60x-2.57x (1.63x-2.64x).

Since the performance of GCN and GraphSAGE are similar, we focus on GCN in the rest of the evaluation for simplicity.

4.5.2 Memory Performance Characterization

GNN workloads are often memory bound. We achieve speedup mainly by improving memory performance. Table 4.4 quantitatively demonstrates the memory performance enhancement from our techniques. The table lists key metrics collected with the Intel VTune Profiler that contribute to the performance difference among the *DistGNN* baseline, *MKL*, *combined*, and *locality*, in GCN training. The characteristics in inference are similar.

In the table, *Retiring* is the fraction of pipeline slots utilized by useful work. Increasing it results in higher instructions-per-cycle (IPC). *Mem. Bound* is the fraction of pipeline slots stalled due to incomplete memory requests. For our workloads, this is primarily from stalls on loads that miss in cache. *L2*, *L3*, *DRAM BW*, and *DRAM Lat.* are the fraction of cycles where execution is impacted by L1 miss/L2 hit, L2 miss/L3 hit, DRAM bandwidth limit,

Table 4.4: Memory performance characterization of GCN training. *combined* and *locality* are profiled with 50% feature sparsity.

Input Graph	Implementation	Pipeline Slots On		Cycles Bound By				Cycles When
		Retiring	Mem. Bound	L2	L3	DRAM BW	DRAM Lat.	L1 FB Full
ogbn-products	DistGNN	9.8%	75.2%	1.5%	2.4%	78.8%	5.3%	100%
	MKL	11.2%	71.8%	0.0%	0.5%	74.4%	5.2%	100%
	combined	18.8%	58.1%	0.8%	1.9%	62.8%	13.4%	100%
	locality	28.7%	39.3%	2.7%	4.7%	40.8%	19.1%	31.3%
wikipedia	DistGNN	23.2%	49.0%	2.4%	3.5%	47.9%	8.5%	100%
	MKL	23.1%	47.7%	0.1%	1.2%	45.4%	10.0%	100%
	combined	33.9%	30.6%	1.5%	2.9%	29.8%	12.6%	42.7%
	locality	34.1%	30.3%	1.5%	1.9%	28.3%	9.6%	39.1%
ogbn-papers	DistGNN	13.5%	75.7%	1.5%	3.5%	77.1%	7.2%	100%
	MKL	13.4%	76.7%	0.0%	0.8%	77.1%	7.0%	100%
	combined	24.5%	58.9%	1.0%	1.8%	60.6%	13.1%	100%
	locality	28.9%	52.0%	1.3%	3.2%	53.4%	15.3%	93.6%
twitter	DistGNN	12.4%	77.2%	2.4%	3.9%	79.1%	7.5%	100%
	MKL	12.3%	78.8%	0.0%	0.9%	79.2%	8.5%	100%
	combined	19.2%	64.3%	1.1%	2.7%	67.3%	16.7%	100%
	locality	22.6%	60.1%	1.4%	3.4%	62.4%	14.9%	100%

and DRAM latency, respectively. *L1 FB Full* is an estimate of how often the L1D fill buffers, a.k.a. the miss status holding registers (MSHR), are fully occupied. This scenario prevents additional L1D miss memory access requests from being issued. A high value is a symptom that the core is starved for data from memory.

DistGNN and *MKL* exhibit similar traits in most metrics. The two implementations are both heavily memory bound on *ogbn-products*, *ogbn-papers*, and *twitter*. On these datasets, they are memory bound in over 70% of the pipeline slots, and only 9.8-13.5% of the pipeline slots are doing useful work. They suffer mainly from the DRAM bandwidth limit. The executions are DRAM bandwidth bound in over 75% of the cycles. On *wikipedia*, the stress on the memory subsystem is lessened, and 23.1-23.2% of the pipeline slots are doing useful work. Nonetheless, L1 fill buffers are always full on all datasets. The difference between the two implementations is that *DistGNN* has more L2 and L3 bound cycles than *MKL*. This implies that the baseline captures more locality in the cache than *MKL* does, which explains why the baseline performs slightly better than *MKL*.

combined incorporates both layer-fusion and feature compression. It is much less memory bound than *DistGNN* and *MKL* on all datasets. The fraction of memory bound pipeline slots are reduced to 30.6-64.3%, and the fraction of useful pipeline slots increases to 18.8-33.9%. Overall, *combined* is less DRAM bandwidth bound than *DistGNN* and *MKL* but more DRAM latency bound than them.

locality further optimizes the memory performance. It lowers the fraction of memory bound pipeline slots to 30.3-60.1% and raises the fraction of retiring pipeline slots to 22.6-

34.1%. Compared with *combined*, generally more data reuse is captured in the cache, and fewer cycles are stalled by DRAM bandwidth.

As discussed in Section 4.3.1 and 4.3.3, we use software prefetches to exploit the spare L1D fill buffer entries. The approach in general can leverage all the fill buffers on the large scale graphs. However, the fill buffers are underutilized on the medium scale graphs, where the the memory access characteristics are significantly improved such that our current software prefetches do not saturate DRAM bandwidth. Although *locality* has already achieved impressive performance, free fill buffer entries suggest that adding more aggressive software prefetches may yield additional speedup.

In summary, our techniques are effective and significantly improve the memory performance of GNN workloads.

4.5.3 Layer Fusion

We now take a closer look at the effectiveness of layer fusion. Figure 4.9 compares *basic* and *fusion* on GCN's hidden layers in inference and the forward propagation in training. Hidden layers have the same input and output feature vector length, which is important for this evaluation. The execution time in the figure is normalized to *basic*. The execution time of *basic* is broken down into aggregation and update times.

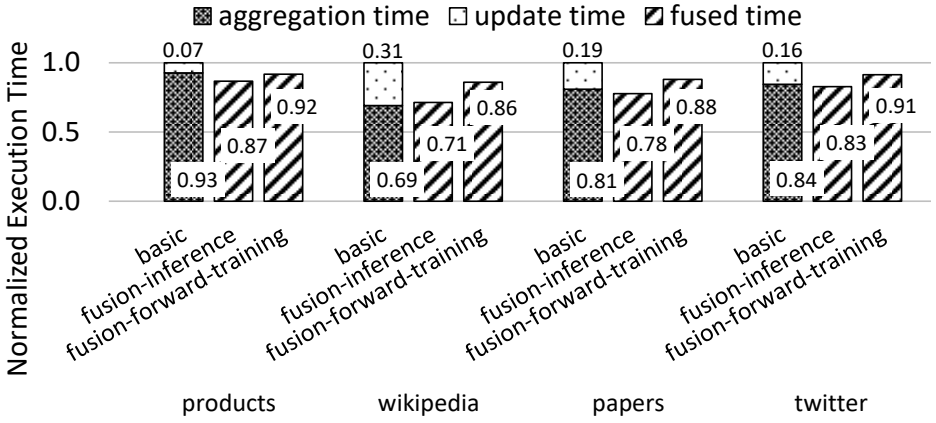


Figure 4.9: The execution time breakdown of *basic* and *fusion* on GCN's hidden layers in inference, normalized to *basic*.

Layer fusion provides more benefit when a larger fraction of *basic*'s execution time is spent on update. Since update is 31% of *basic*'s execution time on *wikipedia*, *fusion* is able to accelerate it by 1.40x in inference. In contrast, *basic* on *ogbn-products* only spends 7% of the time on update, so *fusion* merely outperforms it by 1.15x in inference.

The amount of DRAM traffic for *fusion* in inference and *basic*'s aggregation is similar — they have the same input and output sizes in our evaluation, and although *fusion* accesses additional data for update, those data are expected to be cache-resident. This gives us the opportunity to assess the effectiveness of the compute-memory overlap by comparing the execution time of *fusion* in inference and *basic*'s aggregation. We see that on all datasets, *fusion* in inference takes similar time as *basic*'s aggregation. This implies that for *fusion*, the compute in update is almost fully hidden.

The only difference between *fusion* in inference and in the training forward propagation is that the latter has to keep $\mathbf{a}^k$. This suggests that the performance difference between the two is from the additional traffic of writing to $\mathbf{a}^k$.

4.5.4 Feature Compression

We perform a sensitivity study on the performance of feature compression with different levels of feature sparsity. Figure 4.10 shows the speedup from *compression* over *basic* in GCN inference at feature sparsity levels ranging from 10% to 90%. The behavior in training is similar.

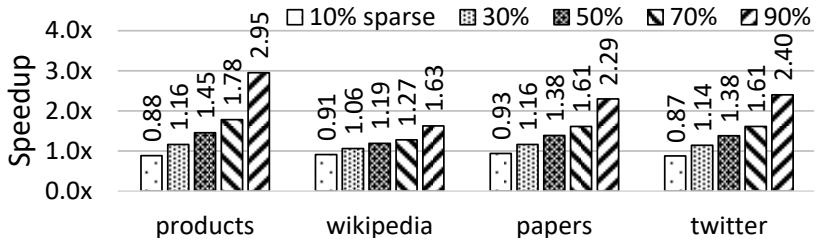
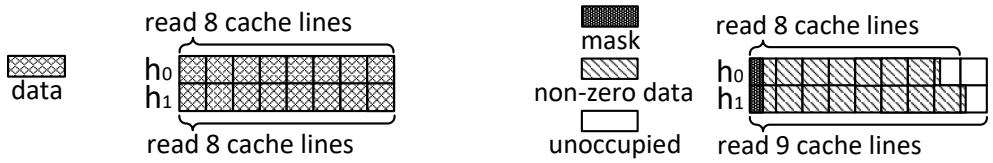


Figure 4.10: The speedup from *compression* over *basic* at different feature sparsity levels on GCN inference.

At 10% sparsity, *compression* is marginally slower than *basic*. At low sparsity, *compression* can generate more memory traffic than *basic*, even if the number of accessed bytes is no larger. This is because the granularity of the memory traffic is a cache line. Consider the example in Figure 4.11. When uncompressed, reading the two feature vectors accesses 16 cache lines (Figure 4.11a). At low sparsity, Figure 4.11b illustrates a case where the masks and non-zero data take less space than the uncompressed data, reading them accesses 17 cache lines.

At 30% sparsity, *compression* on all datasets surpasses the performance of *basic*. At 90% sparsity, which is realistic as shown in Figure 4.8, *compression* outperforms *basic* by 1.63x-2.95x. Thus, feature compression can be a major source of performance improvement.



(a) Uncompressed; read 16 lines. (b) Compressed; read 17 lines.

Figure 4.11: Examples of *compression* accesses more cache lines than *basic* at low sparsity.

4.6 RELATED WORKS

The alternating update and aggregation phases in GNN processing are challenging to both DNN libraries and graph processing frameworks [111, 112]. DNN libraries target regular computations, like the update phase, but perform poorly on the aggregation phase. Graph processing frameworks, on the other hand, manage irregular memory accesses well but lack support for optimized heavy compute operations in updates.

GNN-specific software frameworks and accelerators are emerging. Deep Graph Library (DGL) [104] and PyTorch Geometric (PyG) [113] are two of the prevailing libraries used by researchers. Other frameworks include Neugraph [114] and AliGraph [115]. Our single socket software optimizations can be incorporated into the compute kernels in these frameworks.

Software algorithms have been proposed to optimize GNNs on CPUs. FusedMM [116] fuses the sampled dense-dense matrix multiplication (SDDMM) for computing edge messages with the SpMM for computing vertex messages into a single operation. Their SpMM part performs comparably to MKL. DistGNN [97] is the state-of-the-art in full-batch GNN training on CPU clusters. It also provides single socket optimizations, which we use as our baseline. Dorylus [117] and FlexGraph [118] also focus on distributed training on CPUs, but they do not include single-socket optimizations. Works about distributed training on GPUs include ROC [105], P^3 [119], and DGCL [120].

Besides software, the community also explored custom hardware for GNN workloads. HyGCN [32] is built based on the insight that GNN's two alternating phases show significantly different computation needs and thus uses separate engines for the aggregation and update stages. Additionally, HyGCN manages the pipelined execution of aggregation and update with an inter-phase coordinator. The combination (a.k.a. update) engine utilizes a conventional systolic array to accommodate the huge computation demand, and the aggregation engine has an architecture to handle the irregular accesses with window sliding and shrinking. HyGCN is limited to GCN, and is not generalized to other types of GNN. EnGN [106], inspired by CNN accelerators, treats a GNN as a concatenated matrix multiplication

of feature vectors, adjacency matrices, and weights. With a single dataflow, EnGN is generalizable to many GNN variants. AWB-GCN[33] is motivated by the power-law distribution of most graphs, which means that some parts of the computation are dense and some are extremely sparse, which leads to the imbalance among process engines (PEs). AWB-GCN alleviates the workload imbalance via three balancing algorithms: distribution smoothing, remote switching and evil row remapping. The balancing algorithm is chosen at run-time based on the sparsity and PE's status. GRIP [34] leverages the abstraction of GReTA [121] to develop a general accelerator for any GNN variant. The hardware implementation in GRIP is similar to HyGCN, with specific hardware for vertex-centric and edge-centric computation.

Prior works that reduce memory traffic by compressing DNN sparse activations (features) include cDMA [16], which compresses the PCIe traffic between the GPU and the CPU, and ZCOMP, which compresses sparse features on CPUs by introducing dedicated instructions.

4.7 CONCLUSION

CPUs are good platforms for GNN workloads because of their high availability and high memory capacity. With potentially terabytes of memory, one can perform full-batch GNN computation on real-world large graphs on CPUs, which is impossible on memory-limited devices such as GPUs. However, GNN workloads on CPUs are often highly memory bound, which limits their performance.

In this chapter, we discuss our optimizations of full-batch GNN training and inference on multi-core SIMD CPUs. We mainly focus on the DRAM bandwidth problem that is partially caused by the irregularity from the sparse graph connections. Our proposed techniques include a layer fusion scheme that overlaps the memory-intensive aggregation phase and the compute-intensive updaye phase in a GNN layer, a feature compression scheme that reduces memory traffic by exploiting the sparsity in the vertex feature vectors, and an algorithm that changes the processing order of vertices to improve the temporal locality.

We evaluate our approach with GCN and GraphSAGE on large graphs up to 111 million vertices and 1.6 billion edges. Our techniques are effective. Our implementation outperforms a state-of-the-art GNN layer implementation by 1.72-1.94x on inference computations and 1.60-2.63x on training computations.

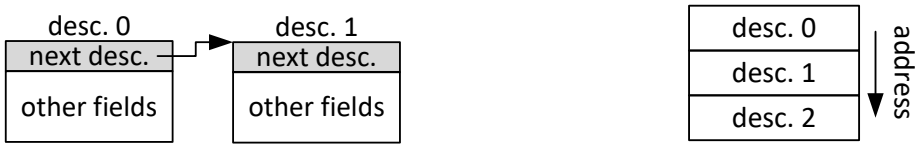
CHAPTER 5: FUTURE WORKS: AUGMENTED DMA ENGINE FOR OFFLOADING GNN AGGREGATIONS

5.1 INTRODUCTION

As discussed in Chapter 4, the aggregation phase of GNNs consists of each vertex gathering its neighbors' feature vectors and performing a simple reduction. Currently, machines execute this phase very inefficiently: processors spend substantial time fetching data from the lower levels of the cache hierarchy, then perform a simple computation, and then are unlikely to reuse the data from their caches because the data has poor locality.

One direction to improve efficiency is to offload the aggregation to a near-memory processor. However, these processors add significant hardware overhead and do not currently exist in commercial systems. On the other hand, we notice that Direct Memory Access (DMA) engines, such as various FPGA IPs [122, 123], do exist in current systems and are light weight. However, they support minimal functionality like a scatter-gather function. Consequently, there is an opportunity to augment DMA engines to implement hardware-assisted aggregation with relatively low cost and minimal intrusiveness.

Existing DMA engines usually employ a descriptor-based programming interface. A descriptor encodes the source and destination addresses as well as the size of the data block to be transferred. Scatter/gather operations are essentially batched data transfers. To describe a gather operation, the software needs to supply a chain of descriptors, where each one encodes the movement of a contiguous data block. The chain can be in the form of a linked list (e.g., Xilinx AXI [122] in Fig. 5.1a) or an array (e.g., Intel DSA [124] in Fig 5.1b).



(a) Linked-list based. (b) Array based.

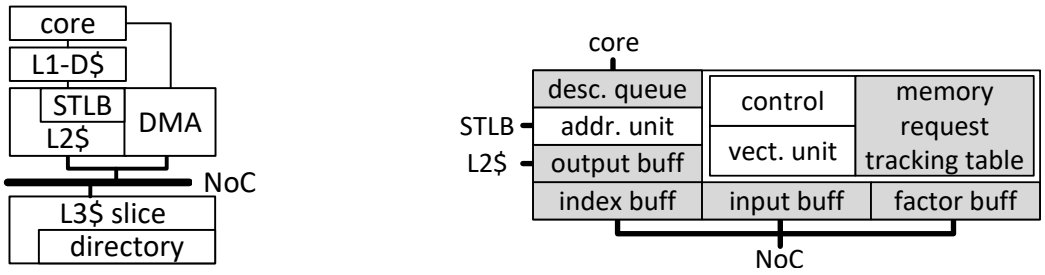
Figure 5.1: Scatter-gather DMA descriptor chain.

5.2 ENHANCED DMA ENGINE FOR GNN AGGREGATIONS

While the software techniques from Chapter 4 are effective at speeding-up GNN workloads, they still leave performance on the table: processor cores are often stalled waiting for data

during aggregation. To address this problem, we propose to offload aggregations from the cores to DMA engines.

In our design, we augment DMA engines that already include gather functionality. Fig. 5.2 shows a block diagram of the enhanced DMA engine. Fig. 5.2a is the top level diagram. It shows that each core is equipped with a DMA engine attached to its L2 cache. The engine takes commands from the core and shares the port to the network on chip (NoC) with L2.



(a) Top level.

(b) The DMA engine.

Figure 5.2: Our enhanced DMA engine.

Fig. 5.2b shows the components in the DMA engine, with storage components shaded. The engine works as follows. The core issues a command by enqueuing a descriptor through a dedicated instruction. When performing an aggregation, the control unit first fetches the indices of the inputs from memory to the index buffer. It then fetches the input data blocks to the input buffer, accordingly. It may also optionally fetch an array of factors to the factor buffer, as will be discussed later. It tracks all memory requests in a table. Next, it computes the reduction in a 4-lane vector unit, holding intermediate results in the output buffer. We choose the width of the vector unit such that the compute does not become a bottleneck. After all inputs are processed, the engine flushes the output buffer to the L2 cache. During the process, the engine performs address translation by looking up the second-level TLB.

We opt not to implement the feature compression in the DMA engine. This is because the compression hardware is expensive. Since models using neither ReLU nor dropout do not benefit from feature compression, the use case does not justify the hardware cost. Next, we will discuss the descriptor and the aggregation operation in detail.

5.2.1 The Aggregation Descriptor

Existing DMA designs use a chain of descriptors to encode a gather operation. Each descriptor in the chain describes a continuous block of data being gathered. This approach is suboptimal for typical GNN aggregations since the data blocks (in this case, the feature

vectors) are relatively small. For example, a 256-element single precision feature vector is only 1KB. Furthermore, rather than describing a set of arbitrarily-sized blocks, we need to only describe a set of fixed-size blocks. Thus, we encode the entire aggregation operation with a single, new descriptor.

Fig. 5.3 shows our proposed 64-byte descriptor and its fields. In the descriptor, *red_op* encodes the reduction operator. *bin_op* encodes the optional binary operator applied on the gathered feature vectors and the elements from a factor array. This is to support the feature processing function ψ described in Sec. 4.2.1. *idx_t* and *val_t* describe the data types of the index array elements and of the input/output, respectively.

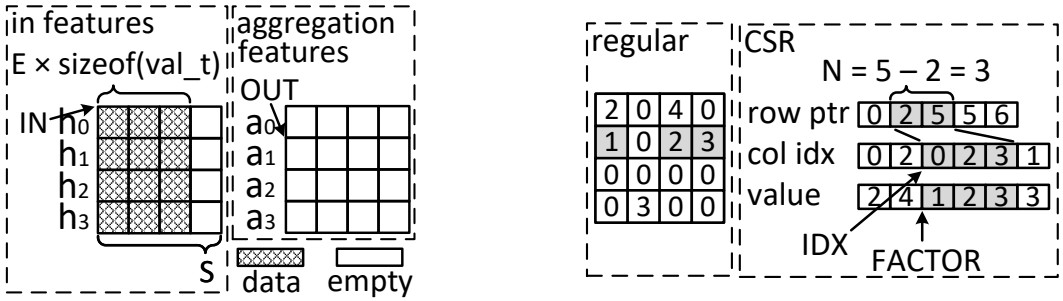
<table>
<tr><td>byte 7</td><td>byte 6</td><td>byte 5</td><td>byte 4</td><td>byte 3</td><td>byte 2</td><td>byte 1</td><td>byte 0</td><td>bytes</td></tr>
<tr><td>red_op</td><td>bin_op</td><td>idx_t</td><td>val_t</td><td colspan="4"># of values in each data block (E)</td><td>0</td></tr>
<tr><td colspan="4">padded size of each data block (S)</td><td colspan="4"># of input data blocks (N)</td><td>8</td></tr>
<tr><td colspan="8">index start address (IDX)</td><td>16</td></tr>
<tr><td colspan="8">input base address (IN)</td><td>24</td></tr>
<tr><td colspan="8">output start address (OUT)</td><td>32</td></tr>
<tr><td colspan="8">factor start address (FACTOR)</td><td>40</td></tr>
<tr><td colspan="8">completion record start address (STATUS)</td><td>48</td></tr>
<tr><td colspan="8">reserved</td><td>56</td></tr>
</table>

Figure 5.3: Proposed descriptor for the aggregation operation.

Field E contains the number of elements in each gathered data block. To enable a user to align data blocks, e.g., to cache line boundaries, the descriptor includes the S field, which encodes the padded size of each data block. Field N has the number of data blocks being gathered. *IDX* is the starting address of the index array. *IN* is the base address of the memory that contains all the data blocks being gathered. *OUT* is the starting address where the aggregation results are written to. *FACTOR* points to the optional factor array when performing a binary operation. Finally, the DMA engine writes the completion status of each operation to the completion record array *STATUS*.

Fig. 5.4a shows how the fields are set in an example aggregation. The example is for a graph with 4 vertices. Hence, the adjacency matrix $\mathbf{A}$ has a dimension of 4×4, and the input feature and the aggregation feature matrices each has 4 rows. Assume that each vertex has 3 features. Since we want to align each feature vector to 4-word cache lines, each feature vector is padded with one additional element. Fig. 5.4a shows the input feature matrix and the aggregation feature matrix before the operation is performed. We see the values of E and S.

Figure 5.4b shows the adjacency matrix $\mathbf{A}$ in both regular (left) and CSR (right) formats. It also shows that we are performing the aggregation operation for the second row. In this case, the number of data blocks is $N = 3$, which is the number of non-zeros in the second



(a) Feature vectors.

(b) Adjacency matrix.

Figure 5.4: Descriptor fields of an example aggregation.

row. N is easily derived from the CSR row pointers. IDX is set to the starting index of the second row in the CSR column indices. If the elements in A are the factors, $FACTOR$ points to the starting index of the second row in the CSR value array. Finally, as shown in Fig. 5.4a, IN points to the starting address of the input feature matrix and OUT points to the second row of the aggregation matrix, where the results will be written to.

5.2.2 The Aggregation Operation

Algorithm 5.1 describes the DMA-aggregation algorithm. In the algorithm, $\mathcal{B}$ is the output buffer. For each of the N inputs, the algorithm calculates the address of each of its E elements (Line 4). Then, it optionally applies the binary operator to the input element and the corresponding factor element (Line 5). Finally, it applies the reduction operator to the processed input element and the corresponding buffer element (Line 6). After an input data block is processed, it writes the completion status to the completion record (Line 7). If the status indicates a failure, the remaining operations are aborted. The algorithm omits this case for simplicity. After all input data blocks are processed in the loop in Lines 2-7, the algorithm flushes the buffer to the output in Lines 8-9. Note that, when *red_op* is "sum" while *bin_op* is "multiply", the algorithm essentially performs a dense-matrix sparse-vector multiplication.

The DMA engine fetches the indices, inputs, and optionally the factors from memory. For each address, it sends a request to the home directory of the address. That directory finds the data and replies to the engine. These fetches are parallelized. The number of entries in the index buffer, input buffer, factor buffer, and memory request tracking table determine the maximum number of fetch requests in flight. Besides structural limits, requests also obey dependences. Specifically, we need the indices first, to calculate the addresses of the inputs.

Fig. 5.5 is an example that illustrates how the DMA hardware performs concurrent fetches,

Algorithm 5.1: DMA-aggregation algorithm.

input : aggregation descriptor d
output : output feature vector OUT

$\mathcal{B}_i = 0 \mid i \in [0..d.E)$
for $i = 0$ **to** $d.N - 1$ **do**
  for $j = 0$ **to** $d.E - 1$ **do**
    $u = d.S \cdot d.\text{IDX}_i + j$
    $k = d.\text{bin\_op}(d.\text{IN}_u, d.\text{FACTOR}_i)$
    $\mathcal{B}_j = d.\text{red\_op}(\mathcal{B}_j, k)$
  $d.\text{STATUS}_i = \text{GET\_STATUS}()$
for $i = 0$ **to** $d.E - 1$ **do**
  $d.\text{OUT}_i = \mathcal{B}_i$

interleaves index and input data fetches, and gives priority to indices to make progress. The figure shows a timeline of the occupancy of a 2-entry Index buffer (top) and a 4-entry Tracking table (bottom). Each entry in the Tracking table is an outstanding request: when an address is first placed in an entry, the engine issues a request to memory, and when the data returns, the entry is freed. The entries in the Index buffer contain fetched indices, and can be reserved in advance. The figure assumes that each requested line contains either two index elements or half of a data block. The other buffers in the DMA engine are not a bottleneck. Also, idx k is required for calculating the address of input k.

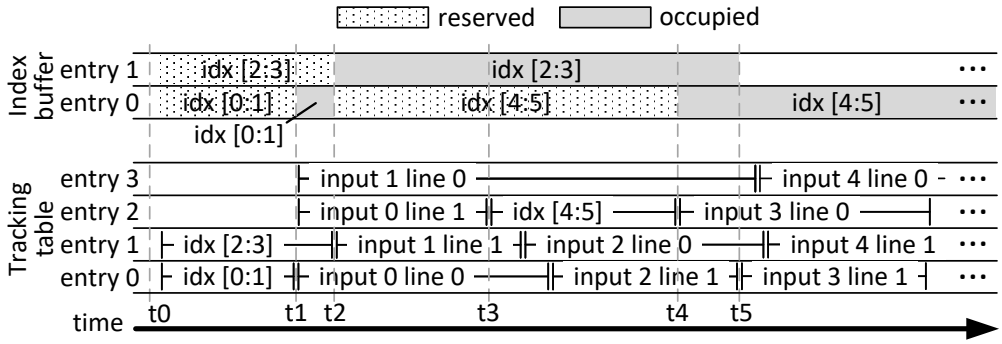


Figure 5.5: Example timeline of DMA requests.

At time t0, the DMA engine allocates Tracking table entries 0 and 1 to fetch indices idx[0:1] and idx[2:3], and reserves Index buffer entries 0 and 1 for when they are received. As idx[0:1] is received at t1, the Tracking table allocates entries for and fetches line 0 and 1 of input 0, and line 0 of input 1. There is no space for line 1 of input 1 until idx[2:3] arrives at t2. At that time, the Tracking table allocates an entry for and fetches line 1 of input 1, and the Index buffer frees-up idx[0:1] and reserves an entry for idx[4:5]. As soon as

a Tracking table entry is freed-up at t3, the table gives priority to allocate an entry for and fetch idx[4:5] over input data. The rest of the timeline proceeds as described.

It is possible that, because of dependences, some Tracking table entries are temporarily unused. Rather than underutilizing the memory bandwidth, the DMA engine simultaneously processes a second descriptor.

The output buffer that holds the intermediate results has a limited size. If the size of a feature vector is higher than this limit, the software can break down the aggregation to fit. For example, if the output buffer can fit 256 elements while each feature vector is 400 elements, the software first issues a DMA-aggregation to produce the first 256 elements and then a second one to compute the remaining 144 elements.

The aggregation feature vectors produced by this DMA operation are the input to the next phase: update. To facilitate the pipelining of the two phases, we opt to write the results of the aggregation to L2. When an aggregation begins, the DMA engine prefetches the output lines to L2 in Exclusive mode. After the aggregation results are produced, the engine writes to these lines. If they have not been evicted from L2, we save the latency of the writes missing in L2.

5.2.3 Software Algorithm Running on the Processor Core

DMA-aggregation is incompatible with feature compression. However, it is synergistic with layer fusion and orthogonal to the locality optimization. Importantly, during a DMA-aggregation, the processor core can work on the update phase, creating a perfect overlap.

Algorithm 5.2 shows the fused DMA-aggregation and update that runs on the processor core. It offloads the aggregation to the DMA engine and performs the update itself. The structure of the algorithm is like the software fusion in Algorithm 4.2. We use ping-pong buffers to pipeline the DMA-aggregation and the update. $\mathcal{Q}_t/\mathcal{Q}'_t$ are the current/previous ping-pong states on thread t respectively. $\mathcal{R}_t$ records the previous vertex tile number on thread t. We bookkeep the previous states for pipelining because we parallelize the main loop with OpenMP's *dynamic* scheduler (Line 2). Dynamic scheduling prevents us from calculating the previous states statically.

The algorithm tiles the computation in the same way as in Algorithm 4.2. Each thread alternates between sending the aggregation descriptors to the DMA engine for B vertices and updating B vertices. Each thread keeps $2 \cdot B$ descriptors. The aggregation phase builds and issues B descriptors (Line 7). The update phase first waits for the DMA-aggregations on the other B descriptors to finish (Line 10). It then performs updates on the aggregation feature vectors produced from those aggregations. After the two phases finish, we update

Algorithm 5.2: Pipelined fused DMA-aggregation and update running on the processor core.

input : graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, input feature matrix $\mathbf{h}^{k-1}$
constant: task size T, block size B, number of threads P
output : output feature matrix $\mathbf{h}^{k}$

```
1  Q_t = 0, Q'_t = ∅, R_t = ∅ | t ∈ [0..P)
2  for i = 0 to |V| − 1 step T · B in parallel do
3      t = ThreadID()
4      for j = 0 to T · B − 1 step B do
5          for m = 0 to B − 1 do
6              v = V_{i+j+m}
7              BUILD_AND_ISSUE(D_{t,Q_t,m}, v)
8          if Q'_t ≠ ∅ then
9              for m = 0 to B − 1 do
10                 WAIT(D_{t,Q'_t,m})
11             for m = 0 to B − 1 do
12                 v = V_{R_t+m}
13                 h_v^k = UPDATE(a_v^k)
14         R_t = i + j, Q'_t = Q_t, Q_t = (Q_t + 1) mod 2
15 for t = 0 to P − 1 in parallel do
16     for m = 0 to B − 1 do
17         WAIT(D_{t,Q'_t,m})
18     for m = 0 to B − 1 do
19         v = V_{R_t+m}
20         h_v^k = UPDATE(a_v^k)
```

the ping-pong states (Line 14). After the main loop finishes, we perform the trailing update (Lines 15-20).

5.3 SUMMARY

This chapter presents our future work on tackling the sparsity-induced memory performance issues in hardware. We observe that performing GNN aggregations in the processor core negatively impacts the performance and energy consumption of the private cache. To address this issue, we propose to augment the existing gather function in the DMA engines to execute the aggregations. With this approach, the low locality feature vectors gathered in the aggregation do not pollute the private cache. In addition, the processor core can compute the update while offloading the aggregation to the DMA engine.

CHAPTER 6: CONCLUSION

DNNs are some of the most popular workloads in recent years. DNN data structures are often sparse due to various reasons. State-of-the-art DNN implementations on CPUs either ignore the sparsity or suffer from performance degradation caused by the sparsity. This thesis studies ways to accelerate DNN workloads on CPUs by exploiting the sparsity as well as by mitigating the negative performance impact from the sparsity. Different types of sparsity in DNN workloads have distinctive characteristics. This thesis investigates their properties and proposes both software and hardware approaches to leverage them.

In the thesis, we explore two directions: to boost the compute performance and to improve the memory performance. In the first direction, we focus on compute-intensive DNN models that have moderately sparse working sets. The modest level of sparsity does not justify using a compressed representation to store the working sets. Our key idea is to skip ineffectual computations when operating on uncompressed data. An operation is ineffectual when it does not affect the end result. This can happen when a computation has zero-valued operand(s).

The thesis makes two contributions based on the idea. The first contribution is *Sparse-Train*, which is the first software-only CPU algorithm that speeds up both DNN training and inference by exploiting the dynamic and unstructured sparsity in the activations. *Sparse-Train* dynamically checks for zeros at run-time and branches over ineffectual computations when a zero is detected.

The second contribution is SAVE, the first sparsity-aware vector engine for CPUs. In a vector compute instruction, some input vector lanes can contain zeros and therefore render the computations on these lanes ineffectual. SAVE combines effectual vector lanes from multiple ready instructions and then issues a compacted vector computation. It is transparent to software and can accelerate legacy code. Although SAVE includes optimizations that targets GEMM-like workloads, it is general-purpose enough to benefit any vector workload that contains sparsity.

In the second direction, we focus on the memory-intensive GNNs. The input graphs to GNNs often have highly sparse connections. The adjacency matrices of these graphs are stored in a compressed format. The indirection required to operate on the compressed format induces irregular memory accesses, hampering the performance. To tackle the issue, we devise software methods to overlap the memory intensive phase and the compute intensive phase in each GNN layer, to reduce unnecessary accesses to the sparse features, and to improve the locality of the irregular memory accesses. In addition to the software optimiza-

tions, this thesis also discusses our future hardware work that aims to offload a GNN layer's memory intensive phase to an augmented DMA engine.

The thesis includes the evaluations of all the aforementioned contributions. The results prove that our techniques are effective. All contributions significantly outperform the state-of-the-art implementations in their respective areas.

APPENDIX A: AN EMPIRICAL STUDY OF THE EFFECT OF SOURCE-LEVEL LOOP TRANSFORMATIONS ON COMPILER STABILITY

A.1 INTRODUCTION

Two of the most important outcomes after sixty years of compiler research are powerful methods for program analysis and an extensive catalog of program transformations. Although there is room for improvement in these areas, we can say that today's compiler analysis and transformation technology rest on solid ground and are well understood. On the other hand, the process of program optimization, which guides the application of transformations to achieve good performance, is not well understood. This is why the standard way of selecting the best compiler command options is to use empirical methods [125, 126]. In addition, because of the lack of understanding of the optimization process, there is much room for improvement even under the best compiler command settings as testified by the numerous projects that apply source-to-source pre-passes to improve the quality of the target code [127, 128, 129, 130].

The benefits derived from tuning compiler options and applying source-to-source transformations are the results of sub-optimal compilers. A hypothetically "perfect" compiler would incorporate the best settings and transformation sequences in its optimization passes and would not need switch selection nor pre-passes. Such a "perfect" compiler would be "stable" in the sense that it would generate the same optimal target code for all semantically-equivalent versions of a loop nest. With near-optimal, stable compilers, programmers could focus on algorithm selection and program readability instead of having to twist the code with what often are obfuscating transformations to improve performance. The undecidability of program equivalence makes it impossible to develop compiler algorithms that generate the same code for all semantically-equivalent code sequences. However, there is no reason why they cannot generate target code that performs quite close to the optimum within a comfortable stability margin.

Although the compiler community is aware of the existence of instability, its magnitude has never been measured. In this appendix, we present the first quantitative study on compiler stability. Because instability implies that there is often a performance headroom of the target code, we also study this headroom and estimate the performance improvement that could result from manipulating the source code. Since the majority of the work is usually performed in loops for compute-intensive applications, we choose to investigate the stability and performance headroom of the loop optimization passes of three popular compilers: *GNU*

Compiler Collection (GCC), *Intel C++ Compiler (ICC)*, and *LLVM C Compiler (Clang)*. The focus is on the compilation of `for` loops because it is among the language constructs whose analyses and transformations are best understood.

In order to make our empirical study as representative as possible, we built an extensive collection of loop nests extracted from 13 benchmarks suites and other sources, such as software libraries and machine learning kernels. The source code of these loop nests as well as their performance results are available in the *LORE repository* [131] developed to serve as a resource for the evaluation of compilers. We implemented an *extractor* to separate `for` loop nests from the original applications and build standalone *codelets* that executes independently. The codelets contain operations that measure execution time and read a number of performance counters. Only loop nests consuming more than 1,000 processor cycles were included in the quantitative study of stability and performance headroom. As a result, out of 3,197 loops that we have extracted, between 1,175 and 1,266 loop nests were investigated, depending on the compiler.

Our methodology for estimating compiler stability and performance headroom is to apply source-to-source transformations to obtain numerous semantically-equivalent versions of each loop nest, called *mutations* in this appendix, and measure the variation in their execution time. Unlike the work mentioned above, our goal is not to improve the quality of the target code, but to study the effectiveness of today's compilers; therefore, we traverse the transformation space to create various loop structures without any performance target.

To generate the mutations, we developed an automated *mutator* that applies source-to-source transformations to the loop nests. These transformation sequences are combinations of five basic yet highly effective loop transformations: interchange, tiling, unrolling, unroll-and-jam, and distribution [132]. Before applying any transformation, the mutator computes the dependences and determines whether or not the transformation can be applied; hence, any transformation sequence applied are semantic-preserving. From the loop nests that we studied, a total of 64,928∼66,392 mutations, depending on the targeted compiler, were generated. The mutations were compiled by the three evaluated compilers. Because vectorization support is ubiquitous in modern processors and can have a significant impact on performance, the quality of a compiler's auto-vectorizer plays a significant role in the compiler's stability. Therefore, we also assessed the compilers' vectorization process by experimenting with different vectorization settings.

We quantified the stability of each compiler with an *intra-compiler stability score*. We found that the evaluated compilers are far from being stable and hence far from optimal. We also devised an *inter-compiler stability score* to measure the stability across multiple compilers, and we used it to confirm that source-level transformations, by moving the per-

formance closer to the optimum, narrow the performance gap between compilers.

Because the mutations are obtained by applying transformations that are widely used and can be easily implemented by any compiler, their effect is a good indication of the room for improvement. Our results show that even though these transformations are likely implemented by the investigated compilers, their availability is not enough, and that the decision on whether or not to apply the transformation and the selection of the right parameters can significantly impact the performance of the resulting code. Although we can only obtain a lower bound of the performance headroom by trying limited combination of transformations, we expect this result, together with a figure of merit for stability, to be a useful indication of the distance from optimality. And, by repeating the measurements along the years, these values could give us a measure of progress.

Our results indicate a significant performance headroom for each of the three compilers evaluated. The application of source-to-source transformations as a pre-pass alone results in 25.9∼36.6% of the loops studied seeing a performance improvement of 15% or more. By further tuning vectorization settings, the numbers rise to 35.7∼46.5%, and a loop nest can expect a 1.61x∼1.65x speedup on average if we manage to find a beneficial mutation and/or better vectorization setting for it.

We also analyzed how each of the five individual transformations applied by the mutator affects performance in order to find the deficiencies in the compilers that cause the instability. We used hardware performance counters and manual inspection for each transformation to establish a correlation between the transformation and execution behavior in terms of locality, number of instructions executed, and vectorization. To attenuate the cost of accessing the performance counters though system calls, we chose to consider only loops with an execution time longer than 10,000 cycles, reducing the number of loops considered in this part of the study to 768∼817 depending on the compiler. We discuss the effect of transformations on specific loops to illustrate the complex ways in which they affect compiler output and the magnitude of the challenge faced by compiler writers in developing stable optimization strategies. For two of the transformations, we also propose ideas that may help compilers increase stability against them. The capability of the compilers' vectorizers is further evaluated by measuring the accuracy of their profitability model and investigating how source-level transformations affect the success rate and effectiveness of vectorization.

The rest of the appendix is organized as follows. Section A.2 describes how we extract loop nests and generate mutations from them. Sections A.3 and A.4 present the experimental settings and quantitative results, respectively. Section A.5 analyzes how different transformations affect performance. Section A.6 explores how vectorization settings impact performance. Section A.7 discusses related work, and Section A.8 presents our conclusions.

A.2 LOOP EXTRACTION AND MUTATION

To study compiler stability and performance headroom, we apply a variety of source-to-source transformations to each `for` loop nest from an extensive collection. To carry out the transformations, the `for` loop in each of these nests is required to be able to transform into the canonical form:

```
for(i=lb;i<=ub;i+=step)
```
(A.1)

We refer to such canonical form as "`for` loop" or simply "loop" in the rest of the appendix unless specified otherwise. To build the collection of loops, an *extractor* outlines all qualified `for` loop nests from a variety of `C` language benchmark suites and libraries. Then, each loop nest is transformed by a *mutator* to generate various mutations. Finally, the execution time of each loop nest and of its mutations is measured, and the variation across mutations of each loop nest is computed.

We developed both the *extractor* and the *mutator* based on the *ROSE* source-to-source compiler infrastructure [133]. This section presents a short description of these two components. For more information, the reader is referred to [131].

A.2.1 The Extractor

The extractor encapsulates each `for` loop nest from an *original program* into a separate standalone program called a *codelet*. The extractor starts with finding all `for` loops in the original program by scanning the abstract syntax tree (AST). A `for` loop is skipped if it contains function call(s) other than standard math functions. If multiple `for` loops resemble a loop nest, the extractor identifies the outermost loop and generates a codelet for the entire loop nest only. Other types of loop such as `while` loops can be included as parts of a `for` loop's body, but our system does not processes them as the outermost loops of loop nests nor apply any transformation to them.

We choose to feed the codelets with the same data used in the original program to make both programs behave similarly as much as possible. To achieve that, the extractor instruments each loop nest to save the values of all read-only or write-after-read (WAR) variables in the loop right before executing the loop. Input data from global/static variables, heap, and stack are handled separately and will be restored to their corresponding locations when later executing the codelet. The loop bounds are recorded at run-time if their values are not constants. Finally, the instrumented original program is executed to create an input data file for each loop nest.

If a loop nest has multiple execution instances (e.g. inside a function that is invoked

multiple times), the extractor saves the data captured from one of the executions chosen using the reservoir sampling algorithm, which grants each execution equal chance to be selected [134].

To create the codelet, the extractor copies the source code of the loop nest from the original program and surrounds it with the following collection of operations:

1. Read the input data file and initialize variables and memory regions with the values they had during the execution of the loop nest in the original program.

2. Record time using the `RDTSCP` instruction, which allows accurate timing measurements with a resolution of just two instructions [135].

3. Read hardware performance counter values.

4. Use all the data that the loop nest produces to generate reduced values that are output to I/O so that the compilers do not remove operations as dead code. For a scalar variable, the codelet writes the variable value; for an integer array, the codelet writes the MD5 hash of the array; and for a floating point array, the codelet writes its sum reduction.

5. Repeatedly execute the loop nest 100 times and record the median of the execution time. Variables and memory regions are reinitialized before each re-execution.

The codelets are thus completely self-contained and ready for transformation by the mutator, compilation, and execution.

Although the source code and input for an extracted loop are replicated from the original program, the loop's behavior during the codelet execution may vary from its behavior during the original program execution because (I) the cache state will typically differ from the state during the execution of the original program. On the other hand, it will be consistent across re-executions of the loop in the codelet (except for the first execution); (II) some of the compiler's inter-procedural and/or inter-loop analysis may lead to changes in the optimization process. For example, the extractor outlines neighboring outermost loops into separate codelets and therefore disables any interaction between them, but a compiler may fuse them when compiling the original program. However, these differences are acceptable since our study focuses on how performance varies between the original codelet and its mutations. Thus, replicating the exact cache state and the optimizations applied on the original program is not essential because our goal is not to optimize the original program.

A.2.2 The Mutator

The extracted loop nests in the form of codelets are processed using the mutator to create semantically equivalent mutations. The mutator applies sequences of source-to-source loop transformations that are constructed from interchange, tiling, unrolling, unroll-and-jam, and distribution. These relatively easy to implement transformations are among the best-known loop transformations, so they can be effortlessly added to any compiler if they are not currently present.

We imposed limitations on the transformation sequences because the number of mutations of a single loop nest may grow exponentially with the number of transformations and the number of possible parameters to each transformation [128]. These limitations ensured the number of mutations generated remained reasonable.

First, the mutator does not explore the transformation space exhaustively. Instead, it applies sub-sequences of transformations of the following sequences:

$$interchange \rightarrow unroll\text{-}and\text{-}jam \rightarrow distribution \rightarrow unrolling$$
$$interchange \rightarrow tiling \rightarrow distribution \rightarrow unrolling$$

A sub-sequence can skip transformation(s) in the above sequences. For example, $interchange \rightarrow distribution$ is a valid sub-sequence. However, the order of transformation needs to be preserved. For instance, $distribution \rightarrow interchange$ is never applied. We chose this ordering to ensure that transformations that only operate on perfectly nested loops (i.e. all assignment statements are in the innermost loop) due to the limitation of our tool, namely interchange, tiling, and unroll-and-jam are not applied after any transformation that may render loop nests imperfect, namely distribution, unrolling, and unroll-and-jam. The maximum length of transformation sequence is 4.

Second, we limit the parameters to each transformation as shown in Table A.1. For interchange, we explore every possible permutation, and the parameter for it is a number denoting the permutation in lexicographical order. For tiling, we tile a single dimension only, and the parameters are the size used for strip mining plus the loop level that is strip-mined. For unrolling, we only unroll the innermost loop(s). If there are multiple loops at the innermost level, the mutator will unroll all of them the same number of times. For unroll-and-jam, we apply it at each non-innermost level, and the parameters are the loop level to be unrolled and the unroll factor. For distribution, we distribute statements in the innermost loop as much as possible based on dependence information; hence, distribution does not take any parameter. As presented in column 2, we only use selected tile sizes/unroll factors for tiling, unrolling, and unroll-and-jam. If the loop level being transformed has a

static trip count that is smaller than one of the tile sizes/unroll factors, the transformation with the corresponding parameter is skipped. For example, the mutator does not apply unroll by 8 times on the loop `for(i=0;i<=5;i+=1){...}`, which has a static trip count of 6.

Table A.1: Transformations and their parameters

Transformation	**Parameters**	**Maximum # of variation**
Interchange	Lexicographical permutation number	$depth! - 1$
Tiling	Loop level, tile size $\in \{8, 16, 32\}$	$depth \times 3$
Unrolling	Unroll factor $\in \{2, 4, 8\}$	3
Unroll-and-jam	Loop level, unroll factor $\in \{2, 4\}$	$(depth - 1) \times 2$
Distribution	N/A	1

Although these limitations confine the search space of transformation sequences to a manageable size for each loop nest, they also decrease the chance of finding the optimal transformation sequence for a loop nest. Therefore, in this study, we only aim at finding lower bounds for performance headroom and instability of the investigated compilers.

In addition to the imposed restrictions, the number of mutations is also limited by data dependence. Unrolling is the only employed transformation that is guaranteed to be semantics-preserving (although additional care needs to be taken when `continue`, `break`, and/or `goto` is present in the loop body). To ensure the legality of the other four transformations, the mutator analyzes dependence through *PolyOptC* [136], which provides an interface between *ROSE* and the polyhedral model based dependence analyzer *Candl* [137]. If a loop nest is incompatible with the polyhedral model, e.g. because of having non-affine array subscripts, the mutator applies only unrolling to it because *Candl* cannot analyze it. Therefore, all generated mutations are guaranteed to preserve the original semantics in theory. In practice, to counteract any potential bug in the mutator or the compilers, we also added a second layer sanity check. As described in Section A.2.1, a codelet writes the reduction of all of the loop's output to I/O. We compare the original loop nest and its mutations' outputs to verify semantic equivalence. Although this method may produce false positives or false negatives, we found it sufficient during our experiments.

The nesting depth and the dependence graph of a loop nest determine how many mutations are produced from it. Among the loops that we studied, up to 1,680 mutations and on average 60 mutations were created from a single loop nest.

A.2.3 Collection of Loop Nests and Their Mutations

We extracted 3,197 loop nests from various sources for this study, such as: benchmarks, audio/video codecs, and machine learning kernels. The first column in Table A.2 lists the

sources that we extracted the loop nests from. For the benchmarks, we used the default dataset during extraction except for SPEC and NPB. We used the "ref" dataset for SPEC and the "CLASS=B" dataset for NPB. For the libraries, we used the test data that they provided. In total, we produced 100,219 mutations from the 3,197 codelets; however, we only study the results from loops whose execution time exceeded 1,000 cycles. The final number of loops and mutations is presented in Table A.2 and discussed in Section A.4.

Table A.2: The numbers of loop nests and their mutations included in the study

Benchmark	**# of loops (# of mutations)**		
	GCC	**ICC**	**Clang**
ALPBench [138]	24 (72)	22 (66)	31 (129)
ASC Sequoia [139]	22 (350)	21 (347)	22 (350)
Cortexsuite [140]	60 (1060)	57 (791)	62 (1042)
FreeBench [141]	38 (242)	31 (141)	39 (245)
Parallel Research Kernels (PRK) [142]	36 (286)	23 (189)	34 (261)
Livermore Loops [143]	53 (1443)	51 (1436)	57 (1612)
MediaBench II [144]	152 (773)	120 (532)	183 (1279)
Netlib [145]	25 (207)	21 (195)	24 (204)
NAS Parallel Bench. (NPB) [146]	196 (52259)	195 (52244)	198 (52350)
Polybench [147]	90 (3574)	91 (3589)	91 (3589)
SPEC 2000 [148]	122 (1263)	125 (1272)	129 (1337)
SPEC 2006 [149]	102 (421)	103 (425)	129 (907)
Extended TSVC [150]	149 (1955)	149 (1955)	149 (1943)
Machine learning kernels [151]	27 (177)	27 (177)	21 (123)
Libraries [152, 153, 154, 155, 156]	145 (1735)	139 (1569)	97 (1023)
Subtotal	1241 (65817)	1175 (64928)	1266 (66392)
Execute for over 1,000 cycles for all 3 compilers		1061 (63902)	

Different benchmark applications may have similar loops (e.g. matrix multiplication kernel). We did not attempt to group similar loops and pick one to represent the group, so the results may bias towards larger clusters of similar loops. However, we believe it is acceptable because it applies a natural weight on the loop types that are more common and thus need more attention.

A.3 EXPERIMENTAL SETUP

We used the loop nests and their mutations to evaluate recent versions of three widely used compilers: *GNU Compiler Collection (GCC)* 6.2.0, *Intel C++ Compiler (ICC)* 17.0.1, and *LLVM C Compiler (Clang)* 4.0.0.

The experiments were conducted on an Intel Xeon E5-1630 v3 processor (Haswell microarchitecture, 32KB/32KB private L1 data/instruction cache, 256KB private L2 cache, 10MB

shared L3 cache) with 32GB DDR4 2133 RAM. The CPU is equipped with an invariant time stamp counter (TSC) so that the readout from `RDTSCP` is accurate regardless of ACPI P-, C-, and T-states [157]. To achieve stable results, all executions were assigned to the same core with dynamic frequency scaling, Intel *Hyper-Threading*, C-State higher then one, and *TurboBoost* technologies disabled. The experimental results from a single machine setup are sufficient to provide an estimation of compiler stability and performance headroom since we believe similar traits may exist on other systems. However, doing experiments on a single hardware setup tends to add measurement bias towards certain compiler(s) [158]. As a result, the difference in the tested compilers' stability and performance headroom reported by our results should not be considered as a conclusive comparison of the compilers' quality because such difference may vary on other systems. In the future, conducting further experiments on additional machine setups may reduce the measurement bias, providing more comprehensive conclusions.

When compiling the loop nests and their mutations, we turned on the following switches in addition to `-O3`:

- *GCC*: `-ffast-math` allows breaking strict IEEE compliance so that floating point operations can be reordered; `-funsafe-loop-optimizations` tells the loop optimizer to assume that loop indices do not overflow, and that loops with nontrivial exit condition are not infinite; `-ftree-loop-if-convert-stores` allows if-converting conditional jumps containing memory writes;
- *ICC*: `-restrict` and `-ipo` help with inter-procedural alias analysis;
- *Clang*: `-ffast-math` has similar effects as in GCC; `-fslp-vectorize-aggressive` enables a second basic block vectorization phase.

We instructed all three compilers to optimize for the native architecture, which supports vector extensions up to AVX2, and let the compilers' default vectorization profitability models determine when to vectorize loops.

A.4 RESULTS

This section presents the main results of our study. In Section A.4.1, we report the performance of the code generated by the evaluated compilers from the original loop nests. We discuss the overall effect of the source-to-source transformations applied by the mutator on performance in Section A.4.2, discuss the performance impact from the length of transformation sequence in Section A.4.3, evaluate the effect of each transformation in Section A.4.4,

discuss the performance headroom of the loops from different benchmarks in Section A.4.5, and finally propose metrics to measure compiler stability and performance convergence in Sections A.4.6 and A.4.7, respectively.

A.4.1 Baseline Performance

Table A.2 lists, for each of the three compilers considered, the number of loops and and the number of mutations used for the part of the study presented in this section. Recall that we require the execution time of a loop nest to be at least 1,000 cycles; therefore, only between 1,175 and 1,266 loop nests are used for evaluating each of the three compilers.

To compare the effect of the compilers on the baseline loops, we only consider the 1,061 loops whose execution time is longer than 1,000 cycles for all three compilers. On average, the code generated by GCC and Clang is 1.06x and 1.27x slower respectively than that by ICC; however, they generate code that outperforms ICC's by at least 15% in 174 and 114 cases, respectively. Therefore, the optimal compiler for each loop varies. We chose a 15% threshold because it is a meaningful difference, even with experimental timing noise [150].

A.4.2 Overall Impact of Our Collection of Mutations on Performance

We calculate the speedup of a loop nest l's fastest mutation over its baseline by

$$speedup^{(l)} = \frac{min(t^{(l)}_{mutation[0]}, ..., t^{(l)}_{mutation[n^{(l)}-1]})}{t^{(l)}_{baseline}} \tag{A.2}$$

In the formula, $t^{(l)}_{baseline}$ and $t^{(l)}_{mutation[i]}$ are the execution times of a compiler's outputs of baseline loop l and its mutation $i \in [0, n^{(l)} - 1]$ respectively, and $n^{(l)}$ is the number of mutations generated from baseline loop l, which varies depending on the loop. We see that, on average, the fastest mutation of a loop is 1.11x, 1.05x, and 1.16x faster than the baseline for GCC, ICC, and Clang respectively, as shown in Table A.3 row 2. Also, the standard deviations of speedup are 1.02∼1.04, suggesting that the range of speedup is significant.

We report the average lower bound of performance headroom by applying source-to-source transformations in row 3, which is calculated by

$$headroom \geq ((\prod_{l=1}^{L} \frac{t^{(l)}_{baseline}}{min(t^{(l)}_{baseline}, t^{(l)}_{mutation[0]}, ..., t^{(l)}_{mutation[n^{(l)}-1]})})^{\frac{1}{L}} - 1) \times 100\% \tag{A.3}$$

When computing the lower bound of headroom, if all mutations are slower than the baseline,

Table A.3: General statistics of mutations' performance impact

		GCC	ICC	Clang
1	# of loops studied (L)	1241	1175	1266
2	μ_g (σ_g) of the fastest mutation to baseline speedup	1.11 (1.02)	1.05 (1.04)	1.16 (1.03)
3	average lower bound of performance headroom	16.7%	13.8%	20.9%
4	# (%) in L that have beneficial mutation(s)	402 (32.4%)	304 (25.9%)	463 (36.6%)
5	# (%) in L that have all mutations unfavorable	89 (7.2%)	188 (16.0%)	73 (5.8%)

we consider it to be 0%. On average, the performance of a loop nest has a headroom of at least 13.8%∼20.9%, depending on the compiler.

We assign categories to each mutation based on its impact on performance. We consider mutations that generate code 15% faster than the baseline to be *beneficial* and those that generate code that is 15% slower than the baseline to be *unfavorable*, the rest are considered to be *neutral*. As shown in row 4, the percentage of loops with at least one beneficial mutation ranges from 25.9% (ICC) to 36.6% (Clang). This suggests that Clang benefits more from source-level transformations than ICC, with GCC sitting somewhere between the two. On the other hand, as shown in the last row, the percentage of loops that only have unfavorable mutations is much higher for ICC at 16.0% vs. Clang at 5.8% and GCC at 7.2%.

Focusing on loops with beneficial mutations, we get the distribution of speedups shown in Figure A.1. The plot reveals that for all three compilers, although the majority of the speedups are below 2x, a number of loops receive an over 2x speedup. In fact, there are loops with speedups as high as 20x; however, we found that most speedups over 6x are due to pathological scalar optimization after unrolling. For example, after the mutator unrolls a loop from *TSVC* 8 times, Clang decides to further fully unroll the loop and pre-calculates most of the scalar operations at compile time, accelerating the loop by 20x. Nevertheless, there is a case where interchange facilitates better locality and vectorization to help a loop nest from *TSVC* gain 15x performance with Clang.

While ICC has fewer loops that are sped up by the source-to-source transformations, the number of loops that have an over 3x speedup is comparable to Clang's and is much greater than GCC's. Furthermore, both ICC and Clang on average obtain a 1.54x maximum speedup for the loops with beneficial mutation(s) whereas GCC on average can only attain a 1.46x maximum speedup for loops in that category.

Next, we consider loops where all mutations are unfavorable. Figure A.2 shows the distribution of loops at various slowdown ranges. The average slowdown for these cases are 1.56x, 1.54x, and 1.49x for ICC, GCC, and Clang, respectively. It turns out that most slowdown factors are less than 2x, but for several loops, slowdowns are greater than 4x and can be up

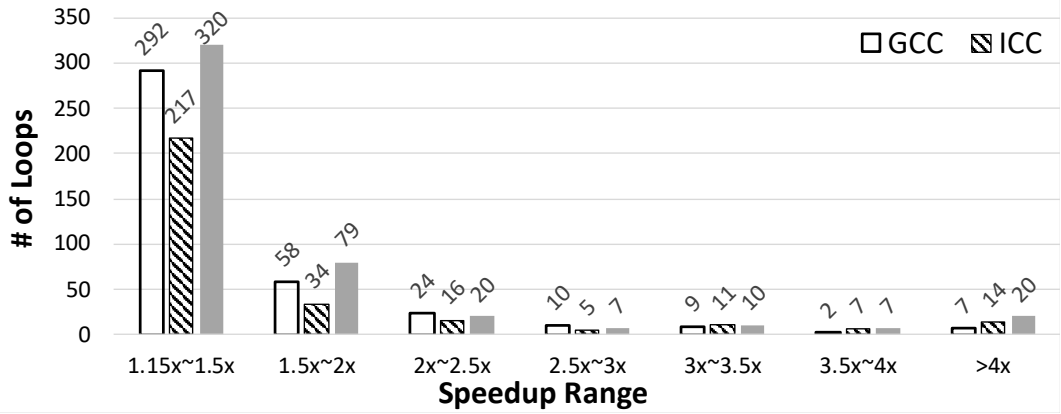


Figure A.1: Distribution of loops with beneficial mutation(s) and their speedup

to 14x in extreme cases. Considering that every loop has a mutation as simple as unrolling by two, these results are surprising. After examining the extreme cases, we learned that large slowdowns are often tied to a sharp increase in instruction count, implying that the compilers generate inefficient code when faced with harmful mutations.

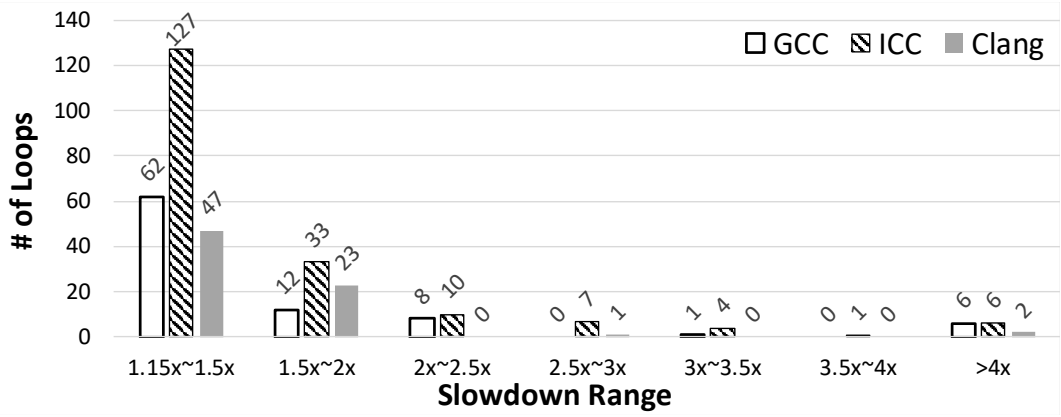


Figure A.2: Distribution of loops with all mutations unfavorable in different slowdown range

A.4.3 Performance Impact from Different Transformation Sequence Lengths

Table A.4 shows the statistics of the performance impact from different transformation sequence lengths. If a given loop nest has mutation(s) that are transformed by s number of transformations, the performance of the fasted mutation among them is used in the calculation of the statistics for sequence length s.

The first set of columns lists the average speedup per sequence length for each compiler. It reveals that for sequence length of 1∼3, on average there is at least one mutation that

Table A.4: Statistics of the performance impact of the length of transformation sequence

Seq. length	μ_g **of speedup**			**% of loops benefited (B)**			μ_g **of speedup in B**		
	GCC	**ICC**	**Clang**	**GCC**	**ICC**	**Clang**	**GCC**	**ICC**	**Clang**
1	1.07	1.02	1.11	29.2%	22.0%	32.0%	1.45	1.55	1.51
2	0.99	0.95	1.06	26.5%	21.1%	35.1%	1.46	1.47	1.55
3	1.01	0.98	1.10	26.6%	22.8%	31.9%	1.54	1.52	**1.79**
4	**0.53**	**0.59**	**0.61**	**2.2%**	**0.0%**	**6.4%**	**1.18**	N/A	**2.06**

performs on par with the baseline for each affected loop nest. However, for sequence length of 4, the fastest mutation is expected to only have 0.53x∼0.61x of the baseline's performance depending on the compiler, as highlighted in the table. This is because as the number of transformations applied increases, the structure of the loop nest becomes more complicated, therefore making the compilers hard to analyze and optimize. The second set of columns lists the percentage of loops affected by a given sequence length that have beneficial mutation(s) of that sequence length. When the sequence length is lower than 4, the percentages are relatively stable for all three compilers, but for sequence length of 4, as highlighted, none of the mutations are beneficial for ICC, and the percentages for GCC and Clang are also very low at 2.2% and 6.4% respectively. The last set of columns lists the average speedup that the loops with beneficial mutations of a given sequence length get from the fastest mutation of that sequence length. When the sequence length is lower than 4, the expected speedup is around 1.5x for all three compilers except when the sequence length is 3, Clang obtains a notably higher speedup of 1.79x. When the sequence length is 4, GCC on average only receives a 1.18x speedup, but surprisingly for Clang, the expected speedup is as high as 2.06x.

The results suggest that (I) by applying a single transformation to a loop nest, we can already expect comparable speedup with applying multiple transformations; (II) Clang may receive more benefit from longer transformation sequences than ICC and GCC do; (III) increasing the sequence length beyond 4 is unlikely to yield better results.

A.4.4 Performance Impact from Each Transformation

Table A.5 shows the statistics of the performance impact from each loop transformation. If a transformation T is legal for a given loop nest, the performance of the mutation that is the result of applying only T with the parameters that produce the highest performance is used in the calculation of the statistics for T. Note that the fastest variant of a transformation may still be slower than the baseline. The table reports the geometric mean and

standard deviation of speedup as well as the percentage of the loop nests that have at least a beneficial variant of the transformation. In general, all three compilers react to the same transformation similarly with some exceptions that are highlighted in the table, which are: (I) while GCC and Clang on average show a speedup from unrolling, ICC shows a slowdown; (II) distribution is able to help ICC much more than it can help the other two compilers; (III) unroll-and-jam and tiling can speed up significantly more loops for Clang than for ICC and GCC; (IV) compared with GCC and Clang, ICC has less loops that benefit from interchange, unrolling, and unroll-and-jam, but more loops that benefit from distribution.

Table A.5: Statistics of speedup from different transformations

Transformation	μ_g **of speedup**			σ_g **of speedup**			**% of loops benefited**		
	GCC	**ICC**	**Clang**	**GCC**	**ICC**	**Clang**	**GCC**	**ICC**	**Clang**
Interchange	0.66	0.69	0.63	1.06	1.05	1.07	9.0%	**6.4**%	9.0%
Tiling	0.83	0.91	0.94	1.03	1.03	1.05	9.5%	9.2%	**16.2**%
Unrolling	**1.06**	**0.97**	**1.09**	1.03	1.04	1.05	25.8%	**18.0**%	29.6%
Unroll & jam	1.01	1.02	1.10	1.02	1.06	1.04	22.7%	**16.2**%	**32.4**%
Distribution	1.12	**1.25**	1.05	1.06	1.11	1.04	27.9%	**34.0**%	27.0%

While unrolling, unroll-and-jam, and distribution increase performance on average, the other two transformations, interchange and tiling, produce a slowdown on average. The intuitive reason is that most loops are already written with good locality, so altering the loop shape may lead to sub-par results from unstable compilers. Nonetheless, 6.4%∼9.0% of the interchanged mutations and 9.2%∼16.2% of the tiled mutations were beneficial, depending on the compiler.

A.4.5 Performance Headroom of the Loops from Each Benchmark

Figure A.3 shows that loops from different benchmarks have varied performance headroom from applying source-level transformations. Among the benchmarks, loops from *polybench* and the libraries have over 20% headroom with all three studied compilers. Source-to-source transformations can accelerate loops from *polybench* with ease because this benchmark is designed for polyhedral compilers, which often are also source-to-source, to optimize the loop nests inside them. Loops from the libraries, on the other hand, have higher headroom because their loop types may be different than those in well-known benchmarks that are intensively studied by the compiler developers.

Among the studied compilers, ICC has the lowest headroom while Clang has the highest headroom for most of the benchmarks. This is mainly because ICC is more aggressive in optimization compared with GCC and especially with Clang. It affects the results in two

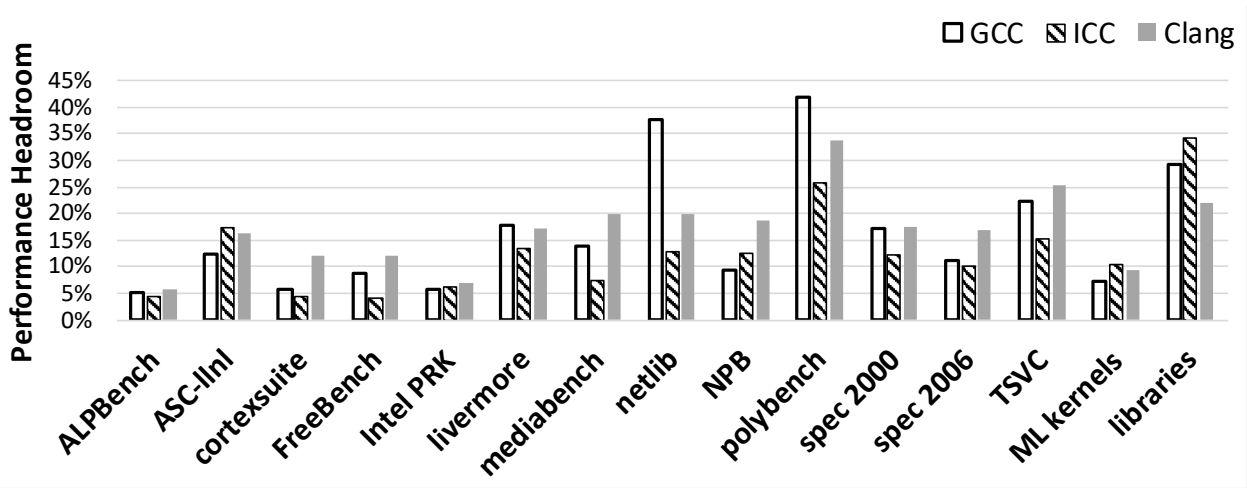


Figure A.3: Average performance headroom available from applying simple source-level transformations

ways: (I) heavy optimization may leave less room for improvement; (II) ICC may sometimes undo source-level transformations. There are cases where it re-rolls or permutates the loop back after we apply unrolling or interchange on a loop. Nevertheless, there are notable exceptions: for the libraries, ICC's headroom (34%) is significantly higher than Clang's (22%); for *netlib*, GCC's headroom (38%) is much higher than ICC's (13%) and Clang's (20%); for *polybench*, GCC also has a headroom (42%) noticeably higher than ICC's (26%) and Clang's (34%).

A.4.6 Intra-compiler Stability

We expect a *perfect* compiler to undo unfavorable transformations and apply beneficial transformations to any mutation of a loop. We call such a compiler *stable* since it would produce the same performance for any mutation of a given loop. We do not expect a perfectly stable compiler to be built in the near future, and perhaps it will never be built. However, we hope that it will be possible to get very close to the perfect compiler, and the first step towards it is to quantitatively measure the stability of compilers. Therefore, we devised the following *intra-compiler stability score* S_{intra}:

$$
\begin{aligned}
S_{intra}^{(l)} &= C_v(t^{(l)}) = \frac{\sigma(t_{baseline}^{(l)}, t_{mutation[0]}^{(l)}, \ldots, t_{mutation[n^{(l)}-1]}^{(l)})}{\mu(t_{baseline_l}^{(l)}, t_{mutation[0]}^{(l)}, \ldots, t_{mutation[n^{(l)}-1]}^{(l)})} \\
S_{intra} &= \frac{1}{L}\sum_{l=1}^{L} S_{intra}^{(l)}
\end{aligned}
\tag{A.4}
$$

We compute $S^{(l)}_{intra}$, the intra-compiler stability of a loop l, as the coefficient of variation of the execution time of its n mutations and its baseline. The coefficient of variation is calculated by scaling the standard deviation of execution time with the average execution time; thus, it approaches 0 if a compiler produces perfectly stable performance for a given loop semantics. We further calculate the intra-compiler stability score of a set of L loops by taking the mean of $S^{(l)}_{intra}$ for all $l \in L$. The stability score has no absolute meaning by itself. Instead, it can be used to compare the stability of different compilers or to track the change in stability between different versions of a given compiler.

The intra-compiler stability score reflects a compiler's ability to recognize optimization opportunities. For example, a stable compiler would interchange back a loop that was interchanged by the mutator if this reversal improves memory access patterns and/or creates vectorization opportunities, a trait that we did occasionally observe from the studied compilers. Table A.6 presents the S_{intra} and the highest $S^{(l)}_{intra}$ calculated from all transformation sequences as well as from individual transformations. Because all transformations are not valid for all loops, the table lists the number of loops included for calculating the stability score for each transformation. In each row, the compiler that produces the highest S_{intra} is highlighted. When considering all transformation sequences, the scores are 0.195, 0.182, and 0.169 for ICC, GCC, and Clang respectively. By definition, a higher stability score reflects greater instability; therefore, among the compilers we tested, ICC is the most unstable overall and Clang is the most stable, with GCC somewhere in between them.

Although the stability score suggests that Clang is more stable than the other two compilers, earlier in Section A.4.2 we showed that source-to-source transformations are, on average, more beneficial to Clang than to ICC and to GCC. Intuitively, this may be because Clang is younger than the other two compilers, so it has a less aggressive yet also less brittle optimization process. This means that it is more stable for the limited set of transformations that we applied in general; however, it benefits more from better structured source code because of its relatively naïve optimization process. Consequently, the mutations compiled by Clang may have a lower performance variation but skew more towards speeding up the original loop.

More interesting observations are made from the stability scores of individual transformations. For interchange, the scores from all three compilers are much higher than their overall scores, indicating that interchange produces significant performance variations. Also, Clang is the least stable toward interchanged loop nests, and ICC is the most stable one against interchange. We believe it is due to ICC having a higher tendency to permute the loop nests, as mentioned in Section A.4.5. For tiling, the stability scores for ICC and GCC are comparable, but the score for Clang is noticeably lower. For unrolling, unroll-and-jam,

Table A.6: Stability scores for different transformations with the most unstable compiler being highlighted

Transformation	**# of loops included**	S_{intra} (**highest** $S^{(l)}_{intra}$) **GCC**	**ICC**	**Clang**
All possible sequences	1061	0.182 (1.189)	**0.195 (1.243)**	0.169 (1.105)
Interchange	169	0.355 (0.866)	0.348 (0.879)	**0.385 (0.881)**
Tiling	430	**0.209 (1.036)**	0.208 (1.169)	0.190 (1.176)
Unrolling	1061	0.097 (1.175)	**0.123 (0.893)**	0.099 (0.788)
Unroll & jam	177	0.112 (0.511)	0.107 (0.524)	**0.137 (0.571)**
Distribution	106	0.111 (0.508)	**0.140 (0.681)**	0.099 (0.523)

and distribution, the stability scores for all three compilers are significantly lower than their overall scores; thus, these three transformations cause less performance variations than the other two. Nevertheless, ICC appears to be less stable than the other two compilers against unrolling and distribution, and Clang is less stable when dealing with unroll-and-jam. The per-transformation results suggest that different compilers may have strengths and weaknesses in terms of stability when facing different source-level transformations.

To demonstrate the instability, we plot in log scale the ratio of the execution time of each loop's fastest mutation and its slowest one (Figure A.4). The performance differences are taken from the 1,061 loops that are shared by all three compilers and then sorted for each compiler separately; thus, the data points at the same x-axis location do not necessarily represent the same loop. The plot clearly shows that ICC typically has the highest performance difference while Clang has the lowest when taking all transformation sequences into consideration.

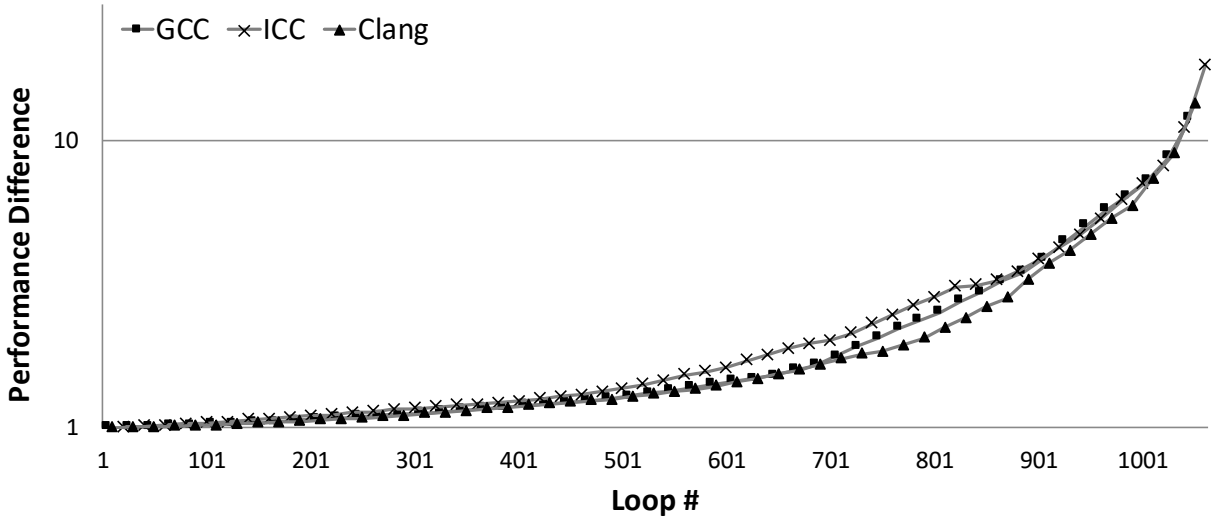


Figure A.4: Performance difference between each loop's best-to-worst mutations

Our results show that all three compilers have significant best-to-worst performance difference for the mutations of most loops, and the difference can be as high as 29x. Therefore, the compilers still have a long way to go before being stable.

A.4.7 Inter-compiler Stability

In Section A.4.1, we mentioned that the performance of the code generated by the three compilers from the same loop may vary, meaning that apart from the intra-compiler stability that we discussed earlier, there is an additional inter-compiler stability to be concerned about. To quantify it, we propose an *inter-compiler stability score* S_{inter} to measure how close the effectiveness of the optimization processes of a set of compilers is, as given by,

$$S_{inter} = \frac{1}{L}\sum_{l=1}^{L}\frac{\sigma(t_C^{(l)})}{\mu(t_C^{(l)})} \quad C \in \{GCC, ICC, Clang\} \tag{A.5}$$

In the formula, $t_C^{(l)}$ is the execution time of baseline loop l compiled by compiler C. Like the S_{intra}, S_{inter} is an average of coefficient of variation. $S_{inter}^{(l)}$ of an individual loop l is computed by first taking the standard deviation of the execution time of the code generated by each compiler and then scaling it using their average execution time, so it measures how close the performances of a set of compilers' generated code are. S_{inter} of a set of loops L is the average of $S_{inter}^{(l)}$ where $l \in L$. The score is meaningful only when used in comparison, and a lower S_{inter} means closer performance. By comparing S_{inter} of different generations of the same set of compilers, one can track the progression of compiler stability in a bigger picture. However, here we study only a single version of each compiler, so we use the score for another purpose: to investigate how source-to-source transformations impact inter-compiler stability. We define the post-transformation inter-compiler stability score S'_{inter} as:

$$\begin{aligned} t_C'^{(l)} &= min_C(t_{baseline}^{(l)}, t_{mutation[0]}^{(l)}, \ldots, t_{mutation[n^{(l)}-1]}^{(l)}) \\ S'_{inter} &= \frac{1}{L}\sum_{l=1}^{L}\frac{\sigma(t_C'^{(l)})}{\mu(t_C'^{(l)})} \quad C \in \{GCC, ICC, Clang\} \end{aligned} \tag{A.6}$$

When calculating $S'^{(}l)_{inter}$ for a single loop, instead of using the execution time of the baseline loop, we use the execution time of the fastest mutation if it is faster than the baseline. Note that the fastest mutation of a given loop can vary when compiled by different compilers. For example, while interchange helps GCC to achieve the best performance for a

given loop, unroll-and-jam can be the key performance enhancer for ICC for the same loop. Therefore, S'_{inter} calculates the inter-compiler stability as if the compilers have incorporated the proper transformation.

From our results, the baseline S_{inter} is 1.39, and the post-transformation S'_{inter} is 0.96. The lower S'_{inter} tells that by applying simple source-to-source transformations, the performance gap among the code generated by various compilers is narrowed; hence, source-level transformations can help the compilers that lag behind catch up. We call this phenomenon *the convergence effect*.

A.5 EFFECT OF TRANSFORMATIONS

To aid compiler developers in the design of more stable compilers, we studied the effects of source-level transformations on the studied compilers. In this section, we investigate how each of the five source-to-source transformations impacts performance by computing the correlation coefficient between performance change and each of a number of hardware performance counter readouts. If a strong correlation between performance and a metric is discovered for a transformation, we can derive the dominant effect produced by the transformation and therefore pinpoint the deficiency in the compilers that causes the instability. In addition, we also present a few case studies illustrating the complexity of the interaction between the transformations applied by the mutator and the compilers. We focused on the individual transformations because performing correlation analysis on all possible transformation sequences that the mutator generates is impractical.

A.5.1 Computing Correlation Coefficients

We compute the correlation coefficient between (a) the change in value of a performance metric and (b) the change in execution time, where "change" refers to the difference between the original loop and the transformed loop. Each performance metric is obtained by reading a hardware performance counter or by computing from multiple hardware counters, as is the case for cache miss rate. Specifically, the correlation coefficient for a transformation T is computed as follows: (I) For each loop nest l, we determine the fastest mutation, m, that is an application of T. Recall that transformations can produce multiple mutations since they are controlled by parameters (Table A.1). (II) Compute the ratio of the execution time of the original loop nest and the execution time of the mutation m: $1/S = t_m/t_{original}$. This is the inverse of the speedup of m over the original loop. (III) Compute the ratio of the values for a performance metric P for the original loops and the mutation m: $R = P_{original}/P_m$.

(IV) Calculate the Pearson correlation coefficient between the performance ratio $1/S$ and the metric ratio R, denoted as $\rho_{1/S,R}$ for all loop nests that can be transformed by T. The values obtained from the metric are inaccurate for loops with short execution time since the hardware counter reading process, which involves system calls, is also partially included in the measurement. Hence, to attenuate the inaccuracy, we only include loops with baseline execution time higher than 10,000 cycles. As a result, depending on the compiler, 768∼817 loops are included in this part of the study. Because the general performance statistics and the counter correlation analysis are decoupled, it is acceptable to use a subset of the loops for the latter.

Note that $\rho_{1/S,R} \in [-1, 1]$, and -1, 1, and 0 represent a perfect negative correlation, a perfect positive correlation, and no correlation, respectively. When the absolute value $|\rho_{1/S,R}|$ is high for a metric P, we may expect transformation T to affect performance mainly in a way that relates to the factors measured by P. On the other hand, if $|\rho_{1/S,R}|$ is insignificant for any of the performance metrics, either the transformation has multiple reasons that impact performance, or the reason is not captured by the limited set of performance metrics that we gather, so its effects are less clear.

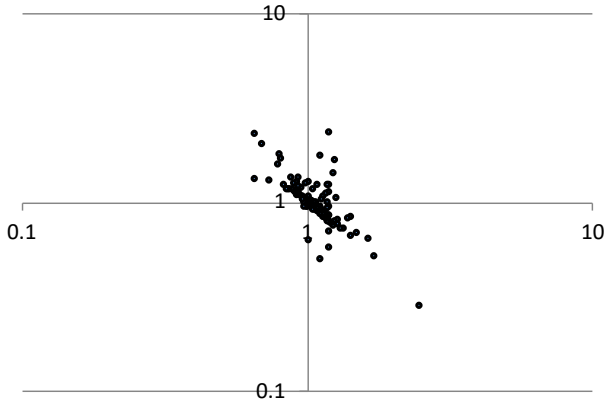
Figure A.5 illustrate these ideas. It plots the ratio of performance factor values R (y-axis) against speedup S (x-axis) in logarithmic scale. For a perfect correlation (i.e. $\rho_{1/S,R} \in \{-1, 1\}$), all points on the plot are expected to be on a $R = kS, k \neq 0$ line. Figure A.5 (a) is a plot with high negative $\rho_{1/S,R}$ value (-0.79), and the points resemble a line $R = kS$ with $k < 0$ with a few noises. Figure A.5 (b), on the other hand, presents a mid-range positive correlation $\rho_{1/S,R} = 0.27$. We can still see a $R = kS$ with $k > 0$, but the points are more scattered. Finally, Figure A.5 (c) plots a $S - R$ relationship with a close to 0 $\rho_{1/S,R}$. In this case, the figure does not manifest a visual correlation. In the rest of the section, we consider $|\rho_{1/S,R}| \in [0.2, 0.5)$ as moderate correlation, and $|\rho_{1/S,R}| \in [0.5, 1]$ as high correlation.

The full list of performance metrics used in our study contains 59 entries. For clarity, we only present the descriptions of performance metrics that show interesting correlations in the following sections in Table A.7, where derived metrics are highlighted.

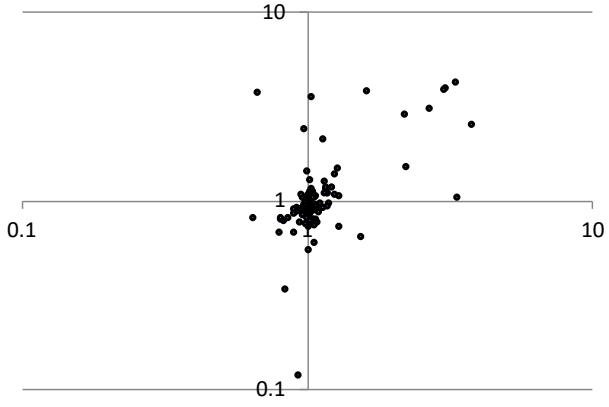
A.5.2 Interchange

Loop interchange can convert non-unit-stride array access to unit-stride, which improves spatial locality. In addition, such a conversion may remove the need of gather-scatter instructions when vectorizing the loop.

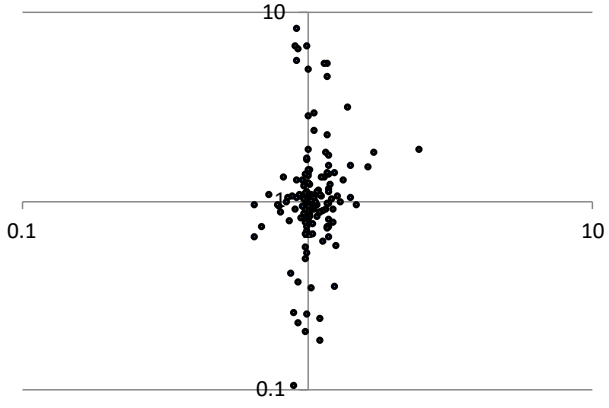
The correlation values indicate that interchange is correlated with cache performance metrics for all three compilers. For ICC and Clang, we found high negative correlations ($\rho_{1/S,R} <$



(a) $\rho_{1/S,R} = -0.79$



(b) $\rho_{1/S,R} = 0.27$



(c) $\rho_{1/S,R} = 0.03$

Figure A.5: Example visualization of different $\rho_{1/S,R}$ range

-0.9) with performance metrics that reflect change in L1 cache behavior (`l1d.replacement`, `l1d_pend_miss.pending`, `l1d_pend_miss.pending_cycles`), as well as in L2 cache behavior (`l2_trans.l2_wb`, `l2_lines_in.all`,`l2_demand_rqsts.wb`). Besides, for ICC there is a high negative correlation with L3 performance metric `%l3_miss`. For GCC, we also see moderate

Table A.7: Top performance metrics that correlate to execution time

Metric	Description
inst_retired.any	Counts the number of instructions retired from execution.
l1d.replacement	Counts when new data lines are brought into the L1 Data cache, which cause other lines to be evicted from the cache.
l1d_pend_miss.pending	Increments the number of outstanding L1D misses every cycle.
l1d_pend_miss.pending _cycles	Cycles with L1D load Misses outstanding.
mem_load_uops_retired .l1_miss_ps	Counts retired load uops in which data sources missed in the L1 cache.
l2_demand_rqsts.wb_hit	Not rejected writebacks that hit L2 cache.
l2_trans.l2_wb	L2 writebacks that access L2 cache.
l2_lines_in.all	Counts the number of L2 cache lines brought into the L2 cache.
l2_rqsts.miss	All requests that missed L2.
dtlb_load_misses .miss_causes_a_walk	Misses in all TLB levels that cause a page walk of any page size.
dtlb_load_misses.stlb_hit	Number of cache load STLB hits. No page walk.
%l1/l2/l3_hit/miss	L1/L2/L3 hit/miss rate
inst_rate	Instruction rate that measures instruction level parallelism (ILP)
l2_rw_rate	L2 read/write rate calculated by dividing the total number of L2 requests by the cycle count

to high negative correlations (-0.7∼-0.4) with these cache related metrics. Moreover, interchange also affects TLB performance indicated by the -0.7∼-0.6 negative correlations with TLB related metrics (`dtlb_load_misses.miss_causes_a_walk`, `dtlb_load_misses.stlb_hit`). The locality improvements subsequently contribute to an increase in instruction execution rate suggested by moderate positive correlations (0.2∼0.5) with `inst_rate`.

In addition, the performance of both GCC and Clang appears to be influenced by the change in dynamic instruction count (`inst_retired.any`) with $|\rho_{1/S,R}|$ ranging from $0.4 \sim 0.6$. This phenomenon implies that interchange can help these two compilers accomplish the same amount of work with fewer instructions. For ICC, we also found that performance change is often accompanied by change in instruction count in the opposite direction. By hand analyzing the instruction mix of affected loops, we corroborated that vectorization contributes to the reduction of instruction count.

Listing A.1 is a case from the *Livermore Loops* that demonstrates how interchange affects instruction count differently for each of the three compilers. ICC gives the best performance for the original loop nest; Clang and GCC's outputs are respectively 1.13x and 1.45x slower than ICC. Based on manual inspection, we conclude that neither GCC nor Clang vectorize the loop due to a non-unit stride access. However, Clang manages to generate a more efficient address calculation than GCC, so Clang's scalar code executes 268K instructions while GCC's executes 384K instructions. On the other hand, ICC vectorizes the loop using gather-scatter and its output executes 288K instructions. However, the speedup from

```
for(k = 0; k < 25; k++) {
  for(i = 0; i < 25; i++) {
    for(j = 0; j < Inner_loops; j++) {
      Px[j][i] += Vy[k][i] * Cx[j][k];
}}}
/* interchanged */
for(j = 0; j < Inner_loops; j++) {
  for(k = 0; k < 25; k++) {
    for(i = 0; i < 25; i++) {
      Px[j][i] += Vy[k][i] * Cx[j][k];
}}}
```

Listing A.1: Original and interchanged *Livermore Loops* code

vectorization is modest due to the gather scatter operations.

After the mutator interchanges the loop nest as shown in Listing A.1, the accesses to `Px` and `Vy` become unit-stride, and `Cx` becomes a loop constant of the innermost loop. As a result, all three compilers vectorize the mutation without gather-scatter. ICC's output now executes only 62K instruction, and is 2.8x faster than its baseline; GCC's numbers are 153K/2.4x, and Clang's are 33K/4.7x. Hence, apart from affecting locality, interchange can enable and/or increase the effectiveness of vectorization. Additionally, after interchange, Clang's mutation takes the lead and becomes 1.46x faster than ICC's output and 2.39x faster than GCC's.

We confirmed that the compilers perform loop interchange in some cases by scrutinizing the assembly. However, the high correlations with locality related metrics suggest that overall they rarely apply it, implying that these compilers fail to do interchange properly is due to inaccurate profitability models and/or not seeking interchange opportunities most of the time. In fact, some extracted loops have embarrassingly bad permutations that can be manually spotted instantly. For example, the loop nest in Listing A.2 from *NPB* has an obviously problematic memory access pattern, and it does not contain loop carried dependence. However, none of the three compilers interchanges it to a better shape, and the properly interchanged mutation becomes 2.6x∼3.2x faster depending on the compiler. Such scenario is common in real world when the programmer does not pay attention to performance, or when the source code is generated by tools such as f2c [159]. Hence, it would be helpful if compilers prioritize the analysis for interchange opportunities.

A.5.3 Unrolling

Loop unrolling is a technique traditionally used to obtain better performance at the expense of program size. Unrolling may improve performance by reducing control overhead (e.g. advancing the iterator and testing exit conditions), exposing more instruction level

```
double dt, rhs[65][65][65][5];

for(j = 1; j <= y - 2; j += 1)
  for(k = 1; k <= z - 2; k += 1)
    for(m = 0; m < 5; m += 1)
      for(i = 1; i <= x - 2; i++)
        rhs[i][j][k][m] = rhs[i][j][k][m] * dt;

/* interchanged */
for(i = 1; i <= x - 2; i++)
  for(j = 1; j <= y - 2; j += 1)
    for(k = 1; k <= z - 2; k += 1)
      for(m = 0; m < 5; m += 1)
        rhs[i][j][k][m] = rhs[i][j][k][m] * dt;
```

Listing A.2: Original and interchanged *NPB BT* code

parallelism (ILP), and/or by enabling scalar optimizations (e.g. common sub-expression elimination, constant folding, etc). Compilers may undo a source level unrolling by applying loop re-rolling.

The correlation results demonstrate that unrolling does reduce dynamic instruction counts. All three compilers show moderate to high negative correlations (-0.7∼-0.3) with the metric `inst_retired.any`. We also see moderate positive correlations (0.3∼0.4) with `l2_rw_rate`, suggesting that although unrolling does not effectively reduce L1 miss rate, its ability to reduce instruction count and increase ILP can better utilize L2 throughput when the locality is captured at the L2 level.

By further examining the assembly code, we observed that the compilers apply unrolling in some cases. Sometimes they may even fully unroll a loop when the trip count is relatively low, potentially exploding the code size. Yet, the compilers fails to apply unrolling in many cases in which it would be beneficial. In particular, compilers may not unroll all loops with small bodies. Unrolling is particularly effective for these loops because they have high overhead due to loop bookkeeping. In addition, since the operation counts are low for a small loop body, unrolling does not excessively inflate the code size. It is surprising that the compilers decide not to unroll these loops, so there is a clear need to improve the profitability model for unrolling by significantly bias towards loops with small bodies.

Unrolling occasionally facilitates vectorization; however, in some cases it can also prevent vectorization. Vectorization is traditionally done by strip-mining the inner loop by the length of the vector registers and then replacing the original loop body with a vector equivalent. This means that fully unrolled loops cannot be vectorized and that partially unrolled loops could lead to inefficient vectorization. For example, ICC and Clang are both able to vectorize a loop from *SPEC 2006* by default, but unrolling the loop twice causes the compilers to

produce scalar code, which leads to 12x and 14x slowdown respectively. In fact, compilers sometimes re-roll a source-level unrolled loop in order to vectorize it [150]. However, the newer basic block vectorization techniques benefit from unrolling. This technique first unroll the loop by a certain factor and then try to assemble isomorphic statements (statements that contain the same operations in the same order) in the unrolled loop body into vector instructions. Such vectorization is also referred to as superword-level parallelism (SLP) [160].

```
for (i = 0; i < 32000; i++) {
  x = a[32000 - i - 1] + b[i] * c[i];
  a[i] = x - 1.0;
  b[i] = x;
}
/* unrolled 8 times */
for (i = 0; i < 32000; i += 8) {
  x = a[32000 - i - 1] + b[i] * c[i];
  a[i] = x - 1.0;
  b[i] = x;
  /* iteration 2~7 are omitted */
  x = a[32000- (i + 7) - 1] + b[i+7] * c[i+7];
  a[i+7] = x - 1.0;
  b[i+7] = x;
}
```

Listing A.3: Original and unrolled *TSVC s281* code

Compared with strip-mining based vectorization, basic block vectorization can partially vectorize a loop more easily. Although we witnessed numerous cases where unrolling helps the compiler to fully vectorize the loop, we present the case of loop *s281* from *TSVC*, as shown in Listing A.3, because it exhibits the interesting trait of partial vectorization. The original loop is not well-optimized by any of the the three compilers that we evaluated. After unrolling the loop 8 times, Clang creates two temporary vectors, denoted as `tmp[0:7]` and `x[0:7]`. The former vector holds the intermediate results from 8 instances of sub-expression `b[i]*c[i]` while the latter vector is an extension of scalar `x`. Then, `tmp[0:7]=b[i:i+7]*c[i:i+7]` and `b[i:i+7]=x[0:7]` are vectorized because they do not have loop carried dependences. Since operations on `a[]` have loop carried dependences, they remain scalar. By partially vectorizing this loop, Clang gains a 1.7x speedup. While compilers have implemented basic block vectorization, this case demonstrates that they may miss vectorization opportunities until the loops are manually unrolled.

Because unrolling can impact vectorization both negatively and positively, it is difficult for programmers and source-to-source optimizers to predict its effect on performance. To avoid this inconsistency, we propose that compilers could first determine whether or not the source code of the loop is unrolled and then choose the appropriate vectorization pass.

Alternatively, compilers could apply an unrolling or a re-rolling pass before the basic block based or strip-mining based vectorization pass respectively, and if the vectorization pass does not succeed, discard the re-rolled/unrolled results (if they are not profitable by themselves).

A.5.4 Unroll-and-Jam

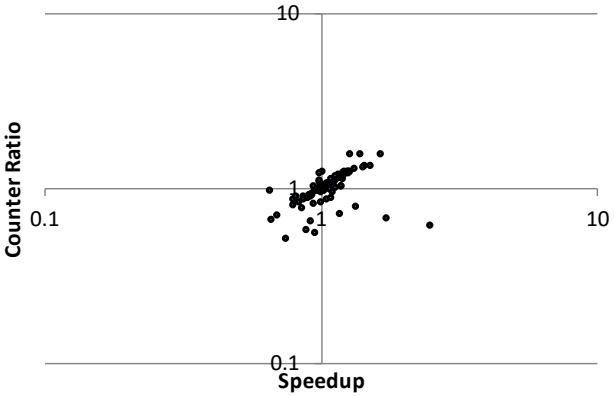
Unroll-and-jam is primarily employed to facilitate data reuse by improving register usage, which decreases the number of memory accesses. In addition, it may enable vectorization on the outer loop without performing interchange [161]. Finally, while not as important, unroll-and-jam may reduce control overhead and/or increase ILP similarly to unrolling. Compilers may reverse a source level unroll-and-jam by applying an interchange → re-rolling → interchange sequence accordingly.

The correlation results for unroll-and-jam are rather interesting. ICC has a high negative correlation (-0.9) with `l1d.replacement` and has a moderate positive correlation (0.4) with `%l1_hit`, indicating that an improvement in L1 hit rate is a major factor for the performance gain. GCC and Clang, on the other hand, have lower negative correlations (-0.6∼-0.5) with `l1d.replacement`; however, they also have negative correlations (-0.6∼-0.5) with `inst_retired.any`. The main difference between ICC and GCC/Clang is the correlations with `l2_rw_rate`. For this metric, ICC exhibits a high negative correlation (-0.7) while GCC and Clang both have moderate positive correlation (0.3). These values imply that unroll-and-jam has a different effect on ICC compared to GCC/Clang.

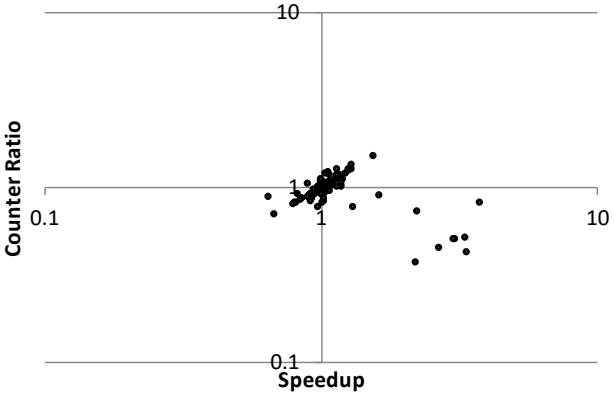
Figure A.6 contains plots of speedup (x-axis) vs. change in `l2_rw_rate` (y-axis) for all three compilers under the influence of unroll-and-jam. From the figure, we see that at low speedup/slowdown, all three compilers show positive correlations with the metric. In fact, 75% of ICC's data points land in quadrant 1 and 3, suggesting a positive correlation. However, for high speedup cases, ICC shows decrease in `l2_rw_rate`, represented by the points in quadrant 4. Since the Pearson correlation biases towards data points with higher values, the correlation coefficient becomes negative. In order to understand ICC's opposite correlation with `l2_rw_rate` at different speedup ranges, we inspected other metrics and found that:

1. At high speedup range, `l1d.replacement` is significantly reduced, which means that the speedup is achieved from better data reuse and thus fewer L1 eviction.

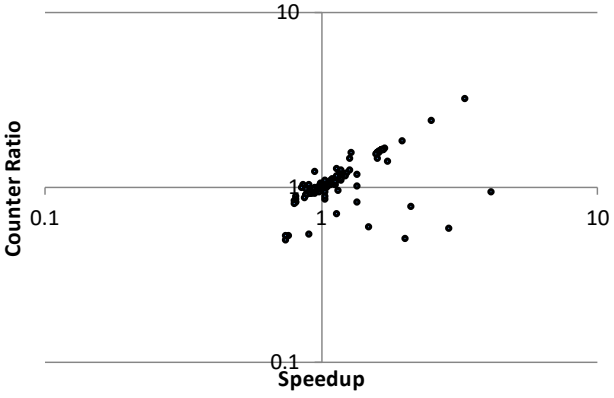
2. At low speedup/slowdown range, the performance change is due to other factors such as lower control overhead and/or higher ILP.



(a) GCC



(b) ICC



(c) Clang

Figure A.6: Speedup vs. change in `l2_rw_rate` due to unroll-and-jam

On the contrary, GCC and Clang have fewer points in quadrant 4 and more points showing positive correlation. More notably, Clang is able to obtain high speedup with a positive correlation with `l2_rw_rate`. By digging into other metrics, we learned that a majority of the speedup corresponds to the negative correlation with `inst_retired.any`, and the source

of the reduction in dynamic instruction count is largely related to vectorization. For GCC, in the 42 cases where unroll-and-jam is beneficial ($>$ 1.15x speedup), 12 loops are not vectorized initially, and 4 of them become vectorized after unroll-and-jam. To further analyze how vectorization is affected, we define *dynamic vectorization rate* as:

$$dynamic\ vectorization\ rate = \frac{dynamic\ vector\ instruction\ count}{dynamic\ instruction\ count} \tag{A.7}$$

We see a general increase in vectorization rate. From the 30 loops that are already vectorized at the beginning, 21 loops receive at least a 15% vectorization rate increase. For Clang, in the 55 cases where unroll-and-jam is beneficial, 9 loops are not vectorized initially, and 4 of them become vectorized afterwards. Note that these 4 loops contain the top 2 speedup that Clang attains through unroll-and-jam, achieving 4.1x and 3.6x respectively. Unroll-and-jam also improves the vectorization rate for 8 loops.

While unroll-and-jam helps ICC's performance, we also observed that it seemingly reduces the effectiveness of vectorization. Of the 28 loops where unroll-and-jam is beneficial, 4 loops are not vectorized initially, and the transformation does not help ICC succeed in vectorizing any of them. Instead, there are 5 loops that are vectorized initially but become not vectorized after unroll-and-jam. Nonetheless, unroll-and-jam manages to speedup these loops by 2.2x to 3.3x. Furthermore, unroll-and-jam reduces the vectorization rate of 7 loops by at least 15%. Two major factors contribute to this situation. First, the benefit from vectorization is overshadowed by a worse memory access pattern. For the cases where scalar mutations outperform vectorized baselines, we always see sharp reduction in L1 miss rate after unroll-and-jam. Second, unroll-and-jam may eliminate performance unfriendly patterns from the vectorized baseline, such as gather-scatter, and generate more efficient vector code, despite having lower vectorization rates.

Listing A.4 contains a loop nest from *Polybench*'s linear algebra workload *gemver* that illustrates how unroll-and-jam helps ICC's vectorizer with a deceiving reduction in vectorization rate. ICC manages to vectorize the original loop nest, yet in an inefficient manner. It first unrolls the inner loop by 32 times. It then transforms the unrolled iterations into 8 vector operation sessions. In each session, 4 elements of `A` are gathered from far apart addresses to assemble a vector, and another vector of 4 `y` elements are directly loaded from consecutive `y`. Then, the two vectors together with a third vector of copies of `beta` are multiplied together. The result vectors from all 8 session are later added together and reduced to a single value that is stored to `x[i]` afterwards. This vectorization is very inefficient in terms of both gather-scatter overhead and locality.

Fortunately, unroll-and-jam provides a better vectorization approach. ICC is able to

```
for(i = 0; i < n; i++) {
  for(j = 0; j < n; j++) {
    x[i] = x[i] + beta * A[j][i] * y[j];
  }
}
/* unroll-and-jammed 4 times*/
for(i = 0; i < n - fringe; i += 4) {
  for(j = 0; j < n; j++) {
    x[i] = x[i] + beta * A[j][i] * y[j];
    x[i+1] = x[i+1] + beta * A[j][i + 1] * y[j];
    x[i+2] = x[i+2] + beta * A[j][i + 2] * y[j];
    x[i+3] = x[i+3] + beta * A[j][i + 3] * y[j];
  }
  /* residue loop is omitted */
}
```

Listing A.4: Original and unroll-and-jammed *Polybench linear-algebra-blas-gemver* code

vectorize the inner loop body with the basic block vectorization technique after the transformation. It first broadcasts `y[j]` to a 256-bit vector register. Then, it loads another vector register with `A[j][i:i+3]` from consecutive addresses. Afterwards, it multiplies the two vectors with a vector of `beta`. Each iteration the result of the multiplication is accumulated onto that from the previous iteration, and after the inner loop exits, the results are written to `x[i:i+3]` with a vector store instruction. Clearly, the new approach is superior since it completely eliminates gather-scatter and has improved locality. In the end, the mutation is 3.7x faster than the baseline. However, the mutation contains 46% vector instructions, whereas the baseline contains 64%, so in this case the vectorization rate is misleading.

A.5.5 Tiling

Tiling is applied to a loop nest primarily when the workset is reused multiple times but is too large to fit in cache. By tiling the loop nest with appropriate block size, blocks of the original workset can be reused without re-fetching from the lower memory, which improves performance. Unlike the other transformations, it is harder for general purpose compilers to un-tile a loop nest.

Due to the limitations described in Section A.2.2, we are only able to examine the effects of 1-D tiling. The correlation results confirm that tiling mainly affects locality. All three compilers exhibit different amount of correlations ($0.3 \sim 0.6$) with metrics related to various cache levels, such as L1 (`l1d_pend_miss.pending_cycles`, `%l1_hit`, `l1d_pend_miss.pending`, `l1d.replacement`), L2 (`l2_rqsts.miss`, `l2_lines_in.all`), L3 (`%l3_hit`), as well as L1 DTLB (`dtlb_load_misses.stlb_hit`).

The hardware counter values also suggest that tiling often increases dynamic instruction count, which stems from the additional address calculations and iterator operations added by the new loop nest level. Therefore, tiling can cause slowdowns if the access patterns do not benefit from tiling or the benefits do not outweigh the overhead.

Furthermore, we found rare cases where tiling enables vectorization for loops with variable bounds. After tiling or strip-mining, the tiled innermost loop no longer has variable bounds. This helps the compilers whose vectorization model is confused by the original variable bounds.

A.5.6 Distribution

Loop distribution separates data streams, which may improve locality and/or prefetching behavior. Compilers may reverse a source level distribution by applying loop fusion. The correlation results confirm its utility. All three compilers have high positive correlations ($0.7 \sim 0.9$) with `l2_rw_rate`, indicating that distributed code may utilize L2 throughput better, likely because of better prefetching behavior. We also see moderate positive correlations ($0.3 \sim 0.5$) with `inst_rate`, which implies that higher L2 access throughput also helps increasing the overall instruction throughput.

A.6 VECTORIZATION

In Section A.5, we have already discussed that vectorization plays a major role in the performance variation. If a loop is not originally vectorizable but, after undergoing a transformation sequence, becomes vectorized, it may receive a sizable performance boost. On the other hand, a loop's performance may suffer greatly if a transformation sequence disables vectorization. We also noticed that there are scenarios where scalar code outperforms vector code due to the overhead introduced during the vectorization process and/or locality difference. Such performance variation caused by vectorization contributes to compilers' instability significantly. Therefore, we took one step further to investigate how different vectorization settings may influence the performance of loops.

We compiled and profiled the loop nests and their mutations with 4 more vectorization settings. We refer to the compiler settings described in Section A.3 as the reference settings, and the additional settings are the reference settings with added switches. They are: generating scalar code only, using only SSE, using SSE and AVX, and using up to AVX2. For the three vector configurations, we disabled the compilers' vectorization profitability analysis if possible so that the compilers vectorize a loop with the corresponding vector extension

whenever possible, regardless of the predicted profitability. Note that Clang 4.0.0 does not provide switches to turn off its profitability analysis.

Because vectorization does not guarantee speedup, instead of looking at a compiler's vectorization report to determine whether a loop is vectorized, we define that a loop has *effective* vectorization if the vector code is at least 15% faster than the scalar code [150]; specifically, we claim that a vectorization attempt is effective if $t_{scalar}/\min(t_{SSE}, t_{AVX}, t_{AVX2}) > 1.15$ where t_s is the execution time of setting s. Since the compiler flags for SSE, AVX, and AVX2 are identical to those for scalar, except for enabling various vector extensions, the performance difference is expected to be mainly from vectorization. Furthermore, a mutation's scalar performance may be significantly lower than that of the baseline. If so, the mutation might not be beneficial overall even if it has effective vectorization. Therefore, for this study, we are only interested in mutations that are both beneficial to the baseline while being vectorized effectively.

A.6.1 Effect of Transformations on Vectorization

Let's first investigate how source-level transformations affect vectorization. The first row of Table A.8 lists the total number of loops that we studied for each compiler, denoted as L. The second row has the number and percentage of loops in L whose baseline are not effectively vectorized, denoted as N. It shows that ICC's vectorizer is the most effective among the three because it fails to vectorize the least percentage of L (59.6%). On the contrary, Clang's vectorizer is the least effective in the sense that it only manages to vectorize 21.1% of L effectively. The next row presents the number and percentage of loops in N that have beneficial mutations, denoted as B. Note that the percentages in this row (39.6%∼47.5%) are much higher than the percentages of loops with beneficial mutation in L, which are 25.9%∼36.6% (second row in Table A.3). This phenomenon indicates that loops that are not originally vectorized have higher chance to receive speedup from source-level transformations. Finally, the last row contains the number and percentage of loops in B whose beneficial mutations are effectively vectorized. It demonstrates that 36.1%∼38.1% of the beneficial mutations are vectorized effectively while their baselines are not; thus, applying the correct source-level transformations can boost compilers' vectorization success rate.

A.6.2 Vectorization Settings

We compiled each mutation with various vectorization settings to assess compilers' effectiveness in (I) deciding whether to vectorize a vectorizable loop and (II) choosing the best

Table A.8: Statistics of effective vectorization

		GCC	ICC	Clang
1	# of loops studied (L)	1241	1175	1266
2	# (%) in L without effective vectorized baseline (N)	866 (69.8%)	700 (59.6%)	999 (78.9%)
3	# (%) in N that has beneficial mutation (B)	373 (43.1%)	277 (39.6%)	475 (47.5%)
4	# (%) in B whose baseline is not vectorized but has vectorized beneficial mutation(s)	141 (37.8%)	100 (36.1%)	181 (38.1%)

vector extension for the task. Table A.9 contains the number of loops that are improved by at least 15% via changing vectorization settings. Row 2 to 4 focus on the benefit by bypassing the profitability model. Row 2 shows that 6.0%∼8.1% of the loops can be sped up over 15% by forcing the compilers to yield scalar code. This situation occurs when the overhead introduced by vectorization (e.g. gather/scatter) is higher than the benefit of vectorization yet the profitability model fails to assess it. Row 3 is for the opposite scenario when the loop is vectorizable and profitable, but the profitability model deems otherwise. Because we were not able to turn off Clang's vectorization profitability model, only GCC and ICC produce this situation, and 2.6%∼3.1% of the loops fall in this category for them. Row 4 is the subtotal of the above two situations. Row 5 counts the loops whose performances increase when vectorized with an older vector extension, i.e. SSE or AVX. In this category, vectorization profitability model is disabled for GCC and ICC but enabled for Clang. This time, 10.0%∼12.1% additional loops receive benefit. Finally as shown in the last row, 18.1%∼21.5% of the loops can receive a sizable performance boost by changing vectorization settings only and without undergoing any transformation. Note that although Clang has the lowest value in the last row, its vectorization profitability model is not necessarily better than those of the other two compilers; in fact, Clang having the highest value in row 2 suggests otherwise. We believe Clang's low percentage in the last row heavily attributes to the inability to disable its profitability model.

Table A.9: Statistics of loops having speedup by changing vectorization settings

		GCC	ICC	Clang
1	# of loops studied (L)	1241	1175	1266
2	# (%) in L best improved with forced scalar	85 (6.8%)	71 (6.0%)	102 (8.1%)
3	# (%) in L best improved with forced vectorization	32 (2.6%)	36 (3.1%)	N/A
4	Subtotal	117 (9.4%)	107 (9.1%)	102 (8.1%)
5	# (%) in L best improved with older vector ext.	150 (12.1%)	138 (11.7%)	127 (10.0%)
6	Total	267 (21.5%)	245 (20.9%)	229 (18.1%)

Figure A.7 plots the speedup distribution of loops that are sped up by only changing the vectorization setting during compilation. While most speedups are below 2x, a number of loops gain speedups of 3x or above. Hence, a more accurate vectorization profitability model and a better understanding on the characteristics of different vector extensions can potentially help compilers to generate much faster results.

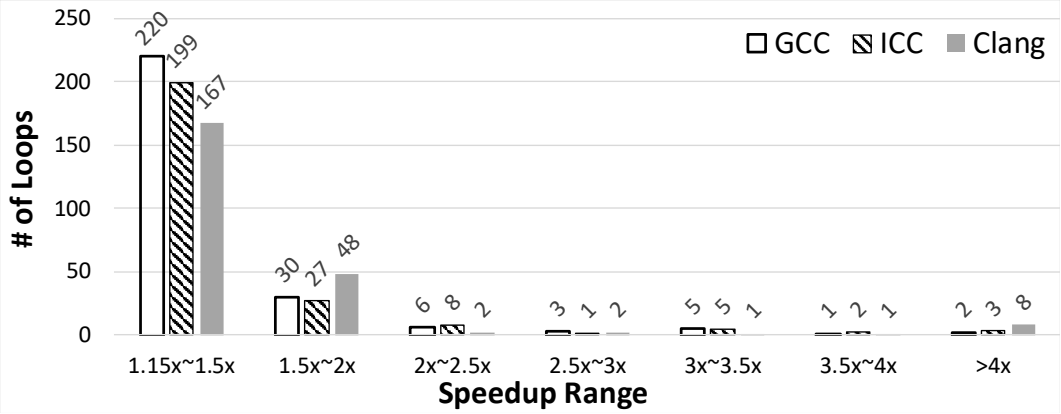


Figure A.7: Distribution of loops that are improved by changing vectorization settings when compiling the original loop

By combining the efforts of applying sequences of source-to-source transformations and searching for the best vectorization setting, we are able to accelerate 579 (46.7%), 420 (35.7%), and 589 (46.5%) loops by over 15% for GCC, ICC, and Clang respectively, and the average speedup of these beneficial cases is 1.61x∼1.65x depending on the compiler. The average lower bound of performance headroom by applying source-to-source transformations as well as searching for the best vectorization setting is 23.7% for GCC, 18.1% for ICC, and 26.4% for Clang.

A.6.3 Vectorization Profitability Case Study

In order to gain insight into the complex factors that affect the profitability of vectorization, we study the case in Listing A.5, which is taken from NPB LU benchmark and whose scalar version outperforms its vectorized counterparts. ICC's vectorized reference compilation of the codelet is 2.2x slower than the scalar compilation. After investigating the assembly code, we discovered that the anomaly may be attributed to the following two reasons.

First, the reference code is vectorized at a length of 2. Instead of packing consecutive elements in the array `ce` to a vector register, the compiler unrolls the loop by a factor of 2 and then picks two adjacent elements in a column, e.g. `ce[0][0]` and `ce[1][0]` to form the

```
double ce[5][13], rsd[64][65][65][5];

for(i = 0; i < nx; i++) {
  iglob = i;
  xi = iglob /(nx0 - 1);
  for(j = 0; j < ny; j++) {
    jglob = j;
    eta = jglob /(ny0 - 1);
    for(k = 0; k < nz; k++) {
      zeta = k /(nz - 1);
      for(m = 0; m < 5; m++) {
        rsd[i][j][k][m] = ce[m][0] + ce[m][1] * xi + ce[m][2] * eta + ce[m
           ][3] * zeta + ce[m][4] * xi2 + ce[m][5] * eta2 + ce[m][6] *
           zeta2 + ce[m][7] * xi3 + ce[m][8] * eta3 + ce[m][9] * zeta3 +
           ce[m][10] * xi4 + ce[m][11] * eta4 + ce[m][12] * zeta4;
}}}}
```

Listing A.5: NPB LU code

vector. Since one vector instruction can process operations from multiple loop iterations in the source code, one would expect the vector code's assembly loop trip count to be less than that of the scalar one. However, the trip count of the scalar loop is instead half of that of the vectorized loop. By scrutinizing the assembly, we learned that the compiler fully unrolls the scalar code's innermost loop. It turns into fewer iterations and improves performance by eliminating the end-of-loop test. Meanwhile, the compiler aggressively schedules instructions for the scalar code after unrolling as there is no data dependence. This enhances instruction pipelining with the help of better ILP.

Second, this loop nest tends to have many write cache misses since the `rsd` array does not fit into L1 and even L2 cache. Vector code is usually supposed to stress memory more than scalar code since it is more likely to complete computations faster. But it turns out that The scalar code surprisingly manages to keep the memory much busier in this case. For example, we found that the scalar code is able to fetch two cache lines concurrently for `rsd` over 14% of the execution time while the vector code is essentially fetching one cache line at a time. Moreover, the scalar code keeps fetching at least one cache line over 70% of the execution time while the vector code keeps the memory busy for only 31% of the execution time. Since the most expensive factor is write cache misses, and the scalar version manages to process it more aggressively, it runs faster than the vector code. We observe that the vector code has more L1 hitting loads from `ce` in between write missing stores to `rsd`. These loads fill up the load buffer, causing the processor to stalls and prevent the processor from executing the stores more aggressively.

Consequently, the fact that compilers may fail to accurately predict the outcome of vec-

torization due to complex factors interfering with each other reduces the compilers' stability.

A.7 RELATED WORK

[150] studied the effectiveness of vectorization in compilers as well as how transformations aid vectorization. Their study demonstrated, for vectorization, some of the instability effects discussed in this appendix. They applied transformations on a smaller collection of loop nests by hand and focused only on vectorization. In comparison, we conducted our study on a large number of programmatically generated mutations from an extensive collection of loop nests, and we also investigate other components of the optimization process besides vectorization. Most importantly, the main perspective that this appendix studies: stability, is not explicitly considered by them.

Exploring the space of program variants as a mechanism to test for correctness of compilers has been studied extensively. A variant is Equivalence Modulo Inputs (EMI) [162], which transforms source programs to generate versions that are semantically-equivalent not for all inputs but for a specific set of program inputs. A compiler defect is detected when the target code from two different versions produce different outputs. The *GLFuzz* technique [163] is similar to our approach in that it applies semantic-preserving transformations to check the correctness of GLSL compilers. Other similar techniques to find performance bugs is discussed by [164]. They call the strategy *Performance Metamorphic Testing*. The work reported in this appendix can be considered as a member of this class of performance testers.

[165] used the performance counter values gathered from executing a loop as the input to a machine learning model that predicts the best polyhedral transformations for the loop. We instead use the correlation between performance counters and performance to investigate the effect of source-level transformation.

Source-to-source transformations have been used as a compiler pre-pass in prior researches in order to improve code performance. [128, 129, 130] adopted the technique in iterative compilation. [128] used the *CHiLL* framework [166] to perform source-level transformations, and they pointed out that the transformation search space grows exponential in size as the number of tuning parameters increases. [129] searched the transformation space that consists of three transformations: array padding, loop unrolling, and loop tiling. [130] searched polyhedral transformation space in their iterative optimization approach. Polyhedral compilers such as *PLUTO* [167] and [168] also work as a source-to-source pre-pass to the back-end compiler. The polyhedral transformations that they apply can be viewed as sequences to loop transformations. However, both iterative compilation and polyhedral compilation only aim to find a good shape of a given loop nest within reasonable time. Therefore, iterative

compilation uses search algorithms to converge to high performance within limited number of steps, and polyhedral compilation applies a single compound transformation based on the polyhedral model. On the other hand, our work purposefully cover a large transformation space in order to evaluate the stability of compilers and to understand the effect of source-level transformations.

Aimed at accelerating performance evaluation of programs, a few prior works also proposed to extract the hotspots from applications and save them as stand-alone codelets. [169] isolated code at the Intermediate Representation (IR) level using LLVM framework. In contrast, our extractor is implemented as a separate component of the *ROSE* compiler to isolate loop nests at the source level. [128, 170] also employed *ROSE* to develop their extractor. While they mainly focused on outlining the kernels of the target application at the function level for automatic kernel tuning and specialization, our extractor concentrates on isolating `for` loops. In addition, the goal of our extractor is to provide stand-alone codelets for loop transformation.

A.8 CONCLUSION

This appendix presents the first quantitative study on compiler stability – the measure of the variation in performance of the target code generated by a compiler from semantically equivalent yet differently structured source code. In this study, we investigated the stability of GCC, ICC, and Clang's compilation processes of C language `for` loop nests. In addition, we also estimated the performance headroom of the said processes.

We measured the compilers' stability and performance headroom by profiling an extensive collection of loop nests extracted from benchmarks and libraries along with their semantically equivalent mutations generated by applying sequences of semantic-preserving transformations.

We quantified compiler stability by introducing a pair of *stability scores*. The *intra-compiler stability score* measures the average variation in performance of semantically equivalent mutations compiled by a given compiler. The score indicates that the three studied compilers are far from being stable. The *inter-compiler stability score* measures the average variation in performance of the target code generated by multiple compilers from the same loop semantics. The score reveals a noticeable performance gap among the compilers. We also used the score to confirm that source-to-source transformations are able to narrow the gap. We believe that the two stability scores are useful tools for compiler developers to evaluate the stability of their compilers as well as for the community to track the progress to compiler stability over time.

To understand the reasons that cause the instability in the compilation process, we analyzed the major effects of the transformations on performance by correlating the performance variation with the change in performance counter readings and derived metrics. Using the correlation and by manually inspecting the assembly code of interesting cases, we were able to suggest improvements on compiler design that may increase the compilers' stability. We also found that the effect of source-level transformations is sometimes difficult to predict. For example, unrolling may either help or harm vectorization depending on the vectorization technique employed. Because different compilers may react to the same transformation in different ways, it is even harder for a programmer to write a loop structure that can be optimized well by multiple compilers.

Because vectorization has a significant impact on performance and stability, we also investigated how different loop structures affect vectorization as well as how effective the compilers' vectorization profitability models are. We observed that when the vectorization profitability model fails, the performance of a loop can be severely harmed. Also, the newest vector extension, despite having longer vector length and more features than the older ones, can be outperformed by the older ones.

With the combined effort of applying source-to-source transformations and tuning vectorization settings, 35.7∼46.5% of the loops, depending on the compiler, exhibit a performance headroom of over 15%. Depending on the compiler, the average performance headroom of these significantly improved loops is 61.4%∼65.3%, and the average performance headroom of all studied loops is 18.1%∼26.4%. The results are the lower bound of potential performance improvement. We believe that the actual headroom may be significantly higher. By expanding the experiment with more loops and transformations in the future, we can gradually raise the lower bound and eventually get a sense of the actual performance headroom and instability of the compilers.

REFERENCES

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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/

[7] 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

[8] 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

[9] 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.

[10] 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

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

[12] 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.

[13] 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/

[14] 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

[15] 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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *arXiv preprint arXiv:1812.08434*, 2018.

[26] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[27] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *arXiv preprint arXiv:2005.00687*, 2020.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Hygcn: A gcn accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 15–29.

[33] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt et al., "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 922–936.

[34] K. Kiningham, C. Re, and P. Levis, "Grip: a graph neural network accelerator architecture," *arXiv preprint arXiv:2007.13828*, 2020.

[35] 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.

[36] Z. Gong, H. Ji, C. W. Fletcher, C. J. Hughes, S. Baghsorkhi, and J. Torrellas, "SAVE: Sparsity-Aware Vector Engine for Accelerating DNN Training and Inference on CPUs," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, October 2020.

[37] 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.

[38] S. Sen, S. Jain, S. Venkataramani, and A. Raghunathan, "Sparce: Sparsity aware general purpose core extensions to accelerate deep neural networks," 2017.

[39] 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.

[40] 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.

[41] 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.

[42] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[43] 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.

[44] 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.

[45] Intel, "Intel(R) Math Kernel Library for Deep Neural Networks (Intel(R) MKL-DNN)," https://github.com/intel/mkl-dnn, 2019.

[46] 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.

[47] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 981–991.

[48] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[49] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[50] 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.

[51] ARM, "ARM Compiler Version 5.06 armcc User Guide," 2016.

[52] A. Kerr, T. Liu, M. Hagog, J. Demouth, and J. Tran, "Programming Tensor Cores: Natice Volta Tensor Cores with CULTLASS," Mar 2019. [Online]. Available: https://developer.download.nvidia.cn/video/gputechconf/gtc/2019/presentation/s9593-cutensor-high-performance-tensor-operations-in-cuda-v2.pdf

[53] L. Wang, Z. Chen, Y. Liu, Y. Wang, L. Zheng, M. Li, and Y. Wang, "A Unified Optimization Approach for CNN Model Inference on Integrated GPUs," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[54] "Xbyak: JIT assembler for x86(IA32), x64(AMD64, x86-64) by C++," https://github.com/herumi/xbyak.

[55] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," ser. CVPR'16.

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

[57] H. Zhang, Y. N. Dauphin, and T. Ma, "Fixup Initialization: Residual Learning Without Normalization," *International Conference on Learning Representations*, 2019.

[58] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz, "Mixup: Beyond Empirical Risk Minimization," *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=r1Ddp1-Rb

[59] A. Krizhevsky and G. Hinton, "Learning Multiple Layers of Features from Tiny Images," Citeseer, Tech. Rep., 2009.

[60] K. Vincent, K. Stephano, M. Frumkin, B. Ginsburg, and J. Demouth, "On Improving the Numerical Stability of Winograd Convolutions," 2017.

[61] 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.

[62] R. Sheikh, J. Tuck, and E. Rotenberg, "Control-Flow Decoupling: An Approach for Timely, Non-speculative Branching," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2182–2203, 2015.

[63] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," ser. ICLR'17.

[64] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," ser. ICLR'17.

[65] X. Sun, X. Ren, S. Ma, and H. Wang, "meProp: Sparsified Back Propagation for Accelerated Deep Learning with Reduced Overfitting," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 70, International Convention Centre, Sydney, Australia, 2017, pp. 3299–3308.

[66] J. Zhang, F. Franchetti, and T. M. Low, "High Performance Zero-Memory Overhead Direct Convolutions," *arXiv preprint arXiv:1809.10170*, 2018.

[67] M. Mathieu, M. Henaff, and Y. LeCun, "Fast Training of Convolutional Networks Through FFTs," *arXiv preprint arXiv:1312.5851*, 2013.

[68] X. Liu, J. Pool, S. Han, and W. J. Dally, "Efficient Sparse-Winograd Convolutional Neural Networks," *CoRR*, vol. abs/1802.06367, 2017.

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

[70] 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

[71] 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.

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

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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).

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

[79] 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.

[80] 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.

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

[82] 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, June 2017.

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

[84] 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.

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

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 1025–1035.

[92] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[93] A. Sanchez-Gonzalez, N. Heess, J. T. Springenberg, J. Merel, M. Riedmiller, R. Hadsell, and P. Battaglia, "Graph networks as learnable physics engines for inference and control," in *International Conference on Machine Learning*. PMLR, 2018, pp. 4470–4479.

[94] T. Hamaguchi, H. Oiwa, M. Shimbo, and Y. Matsumoto, "Knowledge transfer for out-of-knowledge-base entities: A graph neural network approach," *arXiv preprint arXiv:1706.05674*, 2017.

[95] A. Fout, J. Byrd, B. Shariat, and A. Ben-Hur, "Protein interface prediction using graph convolutional networks," in *Advances in neural information processing systems*, 2017, pp. 6530–6539.

[96] V. Garcia and J. Bruna, "Few-shot learning with graph neural networks," *arXiv preprint arXiv:1711.04043*, 2017.

[97] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas, A. Heinecke, D. Kalamkar, N. K. Ahmed, and S. Avancha, "Distgnn: Scalable distributed training for large-scale graph neural networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3480856

[98] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[99] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 974–983.

[100] G. Li, M. Müller, A. Thabet, and B. Ghanem, "Deepgcns: Can gcns go as deep as cnns?" in *The IEEE International Conference on Computer Vision (ICCV)*, 2019.

[101] Y. Rong, W. Huang, T. Xu, and J. Huang, "Dropedge: Towards deep graph convolutional networks on node classification," *arXiv preprint arXiv:1907.10903*, 2019.

[102] M. Chen, Z. Wei, Z. Huang, B. Ding, and Y. Li, "Simple and deep graph convolutional networks," in *International Conference on Machine Learning*. PMLR, 2020, pp. 1725–1735.

[103] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[104] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[105] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, "Improving the accuracy, scalability, and performance of graph neural networks with roc," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 187–198, 2020.

[106] S. Liang, Y. Wang, C. Liu, L. He, L. Huawei, D. Xu, and X. Li, "Engn: A high-throughput and energy-efficient accelerator for large graph neural networks," *IEEE Transactions on Computers*, 2020.

[107] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World Wide Web*, S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, Eds. ACM Press, 2011, pp. 587–596.

[108] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[109] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "Libxsmm: Accelerating small matrix multiplications by runtime code generation," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 981–991.

[110] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[111] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[112] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016, pp. 1–12.

[113] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[114] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, "Neugraph: parallel deep neural network computation on large graphs," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 443–458.

[115] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "Aligraph: a comprehensive graph neural network platform," *arXiv preprint arXiv:1902.08730*, 2019.

[116] M. K. Rahman, M. H. Sujon, and A. Azad, "Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 256–266.

[117] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim, and G. H. Xu, "Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/thorpe pp. 495–514.

[118] L. Wang, Q. Yin, C. Tian, J. Yang, R. Chen, W. Yu, Z. Yao, and J. Zhou, "Flexgraph: A flexible and efficient distributed framework for gnn training," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3447786.3456229 p. 67–82.

[119] S. Gandhi and A. P. Iyer, "P3: Distributed deep graph learning at scale," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/gandhi pp. 551–568.

[120] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu, "Dgcl: An efficient communication library for distributed gnn training," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3447786.3456233 p. 130–144.

[121] K. Kiningham, P. Levis, and C. Ré, "Greta: Hardware optimized graph processing for gnns," in *Proceedings of the Workshop on Resource-Constrained Machine Learning (ReCoML 2020)*, 2020.

[122] Xilinx, "AXI DMA v7.1 LogiCORE IP Product Guide," June 2019.

[123] Lattice, "Scatter-Gather Direct Memory Access Controller IP Core User Guide," Mar 2015.

[124] Intel, "Intel Data Streaming Accelerator Architecture Specification," Sept 2019.

[125] Z. Pan and R. Eigenmann, "Fast, automatic, procedure-level performance tuning," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006, pp. 173–181.

[126] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO))*, 2003, pp. 204–215.

[127] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The structure of an advanced vectorizer for pipelined processors," in *Fourth International Computer Software and Applications Conference*. IEEE, 1980, pp. 201–218.

[128] A. Tiwari, J. K. Hollingsworth, C. Chen, M. Hall, C. Liao, D. J. Quinlan, and J. Chame, "Auto-tuning full applications: A case study," *The International Journal of High Performance Computing Applications*, vol. 25, no. 3, pp. 286–294, 2011.

[129] G. Fursin, M. F. O'Boyle, and P. M. Knijnenburg, "Evaluating iterative compilation," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2002, pp. 362–376.

[130] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, "Iterative optimization in the polyhedral model: Part ii, multidimensional time," in *ACM SIGPLAN Notices*, vol. 43, no. 6. ACM, 2008, pp. 90–100.

[131] Z. Chen, Z. Gong, J. Szaday, D. C. Wong, D. Padua, A. Nicolau, A. V. Veidenbaum, N. Watkinson, Z. Sura, S. Maleki, J. Torrellas, and G. DeJong, "Lore: A loop repository for the evaluation of compilers," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, 2017, pp. 219–228.

[132] R. Allen and K. Kennedy, "Optimizing compilers for modern architectures a dependence-based approach," 2001.

[133] D. Quinlan, "ROSE: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 215–226, 2000.

[134] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, Mar. 1985.

[135] G. Paoloni, "How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures," *Intel Corporation*, p. 123, 2010.

[136] L.-N. Pouchet, "Polyopt/C: A polyhedral optimizer for the rose compiler," http://web.cse.ohio-state.edu/~pouchet/software/polyopt, 2011.

[137] C. Bastoul and L. Pouchet, "Candl: the chunky analyzer for dependences in loops," tech. rep., LRI, Paris-Sud University, France, Tech. Rep., 2012.

[138] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes, "The ALPBench benchmark suite for complex multimedia applications," in *IEEE International Proceedings of the IEEE Workload Characterization Symposium (IISWC)*, 2005, pp. 34–45.

[139] LLNL, "Asc sequoia benchmark," https://asc.llnl.gov/sequoia/benchmarks/, 2008.

[140] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor, "Cortexsuite: A synthetic brain benchmark suite," in *IEEE International Proceedings of the IEEE Workload Characterization Symposium (IISWC)*, 2014, pp. 76–79.

[141] P. Rundberg and F. Warg, "The FreeBench v1.0 benchmark suite," 2002.

[142] R. F. Van der Wijngaart and T. G. Mattson, "The parallel research kernels." in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6.

[143] T. Peters, "Livermore loops coded in c," 1992.

[144] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf, "Mediabench ii video: Expediting the next generation of video systems research," *Microprocessors and Microsystems*, vol. 33, no. 4, pp. 301–318, 2009.

[145] S. Browne, J. Dongarra, E. Grosse, and T. Rowan, "The Netlib mathematical software repository," *D-Lib Magazine, Sep*, 1995.

[146] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber et al., "The NAS parallel benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[147] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," 2012.

[148] J. L. Henning, "Spec cpu2000: Measuring cpu performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[149] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[150] S. Maleki, Y. Gao, M. J. Garzar, T. Wong, D. A. Padua et al., "An evaluation of vectorizing compilers," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 372–382.

[151] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[152] xiph.org, "Codecs from Xiph.Org Foundation," https://www.xiph.org, 2017.

[153] LAME, "LAME MP3 Encoder," lame.sourceforge.net, 2017.

[154] TwoLAME, "TwoLAME - MPEG Audio Layer 2 Encoder," www.twolame.org, 2017.

[155] GAP, "GAP - Groups, Algorithms, Programming - a System for Computational Discrete Algebra," www.gap-system.org, 2007.

[156] Mozilla, "Mozilla JPEG Encoder Project," github.com/mozilla/mozjpeg, 2017.

[157] Intel, "Intel 64, and ia32 architectures software developer's manual, vol. 3a: system programming guide, part 1," *Intel Corporation, Denver, CO*, 2016.

[158] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" *ACM Sigplan Notices*, vol. 44, no. 3, pp. 265–276, 2009.

[159] S. I. Feldman, "A fortran to c converter," in *ACM SIGPLAN Fortran Forum*, vol. 9, no. 2. ACM, 1990, pp. 21–22.

[160] S. Larsen and S. Amarasinghe, *Exploiting superword level parallelism with multimedia instruction sets*. ACM, 2000, vol. 35, no. 5.

[161] D. Nuzman and A. Zaks, "Outer-loop vectorization-revisited for short simd architectures," in *Parallel Architectures and Compilation Techniques (PACT), 2008 International Conference on*. IEEE, 2008, pp. 2–11.

[162] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594334 pp. 216–226.

[163] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," *PACMPL*, vol. 1, no. OOPSLA, pp. 93:1–93:29, 2017. [Online]. Available: http://doi.acm.org/10.1145/3133917

[164] S. Segura, J. Troya, A. D. Toro, and A. R. Cortés, "Performance metamorphic testing: Motivation and challenges," in *39th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track, ICSE-NIER 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017. [Online]. Available: https://doi.org/10.1109/ICSE-NIER.2017.16 pp. 7–10.

[165] E. Park, J. Cavazos, L.-N. Pouchet, C. Bastoul, A. Cohen, and P. Sadayappan, "Predictive modeling in a polyhedral optimization space," *International journal of parallel programming*, vol. 41, no. 5, pp. 704–750, 2013.

[166] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," Citeseer, Tech. Rep., 2008.

[167] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model," in *International Conference on Compiler Construction*. Springer, 2008, pp. 132–146.

[168] D. Adamski, G. Jabłoński, P. Perek, and A. Napieralski, "Polyhedral source-to-source compiler," in *Mixed Design of Integrated Circuits and Systems, 2016 MIXDES-23rd International Conference*. IEEE, 2016, pp. 458–463.

[169] P. D. O. Castro, C. Akel, E. Petit, M. Popov, and W. Jalby, "CERE: LLVM-based codelet extractor and replayer for piecewise benchmarking and optimization," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 1, pp. 6:1–6:24, 2015.

[170] C. Liao, D. J. Quinlan, R. Vuduc, and T. Panas, "Effective source-to-source outlining to support whole program empirical optimization," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2009, pp. 308–322.