Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism

Jiecao Yu[1], Andrew Lukefahr[1], David Palframan[2], Ganesh Dasika[2], Reetuparna Das[1], Scott Mahlke[1]
[1]University of Michigan [2]ARM
{jiecaoyu, lukefahr, reetudas, mahlke}@umich.edu, dpalframan@gmail.com, ganesh.dasika@arm.com

ABSTRACT

As the size of Deep Neural Networks (DNNs) continues to grow to increase accuracy and solve more complex problems, their energy footprint also scales. Weight pruning reduces DNN model size and the computation by removing redundant weights. However, we implemented weight pruning for several popular networks on a variety of hardware platforms and observed surprising results. For many networks, *the network sparsity caused by weight pruning will actually hurt the overall performance despite large reductions in the model size and required multiply-accumulate operations.* Also, encoding the sparse format of pruned networks incurs additional storage space overhead. To overcome these challenges, we propose *Scalpel* that customizes DNN pruning to the underlying hardware by matching the pruned network structure to the data-parallel hardware organization. Scalpel consists of two techniques: *SIMD-aware weight pruning* and *node pruning*. For low-parallelism hardware (e.g., microcontroller), SIMD-aware weight pruning maintains weights in aligned fixed-size groups to fully utilize the SIMD units. For high-parallelism hardware (e.g., GPU), node pruning removes redundant nodes, not redundant weights, thereby reducing computation without sacrificing the dense matrix format. For hardware with moderate parallelism (e.g., desktop CPU), SIMD-aware weight pruning and node pruning are synergistically applied together. Across the microcontroller, CPU and GPU, Scalpel achieves mean speedups of 3.54x, 2.61x, and 1.25x while reducing the model sizes by 88%, 82%, and 53%. In comparison, traditional weight pruning achieves mean speedups of 1.90x, 1.06x, 0.41x across the three platforms.

CCS CONCEPTS

• **Computer systems organization** → **Parallel architectures**; • **Computing methodologies** → Parallel computing methodologies; Machine learning;

KEYWORDS

neural network pruning; hardware parallelism; single instruction, multiple data

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080215

ACM Reference format:
Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, Scott Mahlke. 2017. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 13 pages.
https://doi.org/10.1145/3079856.3080215

1 INTRODUCTION

Deep Neural Networks (DNNs) have become ubiquitous in various application domains including computer vision [15], natural language processing [11] and speech recognition [4]. Recent evidence reveals that the network depth is of crucial importance [22, 38]. Deeper models with more parameters greatly improve the accuracy of DNNs. Pioneering networks from the 1990's used less than 1M parameters for classification of handwriting digits [32]. Two decades later, AlexNet [29] and VGG [38] employ 61M and 138M parameters, respectively, to do classification on 1000 image categories of ImageNet. Large DNN models with lots of internal redundancy can achieve high accuracy, but at the cost of immense computation and energy requirements.

Weight pruning compresses DNN models by removing their internal redundancy. One such method, Deep Compression [19, 20], reduces the number of weights in AlexNet and VGG-16 by 9x and 13x, respectively. The compressed networks achieve 3-4x layerwise speedup on both CPUs and GPUs.

To investigate weight pruning more deeply, we reimplemented Deep Compression and measured the performance of five popular networks across three hardware platforms (ARM Cortex-M4 Microcontroller, Intel Core i7-6700 CPU, NVIDIA GTX Titan X GPU). We discovered surprising results. For 8/15 configurations, the performance of the networks after weight pruning was actually *worse* than before pruning. Pruning hurts performance despite removing an average of 80% of the weights. As an example, the execution time of AlexNet on the CPU increases by 25% even though 89% of its weights are removed. For the remaining configurations, a performance gain was observed with weight pruning, but that performance speedup lagged far behind the actual reduction in multiply-accumulate (MAC) operations. For instance, weight pruning can remove 76% of the MAC operations in LeNet-5 [32], but the execution time on the microcontroller is only reduced by 16%. For the tested hardware, a performance loss was consistently observed on the GPU while modest performance gains were observed on the microcontroller. The CPU yielded mixed results with some networks achieving a gain and others a loss.

To understand these counter-intuitive results, we need to examine weight pruning in more depth as well as the structure of the

networks and the interplay of both with the underlying hardware. DNNs consists of two types of layers: fully-connected layers and convolutional layers. They perform matrix-vector and matrix-matrix multiplication, respectively. Weight pruning techniques [19, 21, 31] measure the importance of each weight and remove those deemed unimportant, resulting in both memory storage and computation reductions. After weight pruning, redundant weights and related MAC operations are removed. The matrix computation in the pruned networks becomes sparse, thus the remaining weights are stored in a sparse matrix format.

The sparsity of pruned networks often leads to a performance decrease in DNN computation. Sparse weight matrices lose the regular structure of dense matrices, and sparse matrix multiplication needs extra computation to decode the sparse format. The performance impact of weight pruning is heavily intertwined with the underlying hardware. On microcontrollers, weight pruning consistently improves DNN performance. The simple architecture of microcontrollers cannot hide the memory access latency. The reduction in model size can, therefore, make up for the computation inefficiencies of sparse matrix multiplication. However, the reduction in execution time is still much lower than the computation reduction. For GPUs, weight pruning consistently loses performance. The sparse matrix computation cannot make optimal usage of the supported hardware, e.g. memory coalescing. Also, dense matrix optimizations, like matrix tiling, are less effective.

For CPUs, the effect of weight pruning varies for different networks and depends on the computation breakdown between fully-connected and convolutional layers. For fully-connected layers, weight pruning can improve performance because the total memory footprint is critical to matrix-vector multiplication. But for convolutional layers that perform matrix-matrix multiplication, the matrix data will be reused multiple times, and there is limited benefit from the memory footprint reduction. Therefore, the inefficiencies inherent in the sparse matrix format will hurt the performance of convolutional layers on CPUs. In addition to the performance decrease, another challenge for weight pruning is that a significant amount of data is necessary to record the sparse matrix structure. Each nonzero weight needs one extra column index to record its position. This extra overhead reduces the impact of weight pruning across all hardware platforms.

To address these challenges, we propose *Scalpel* to customize DNN pruning to the underlying data-parallel hardware structure. Scalpel consists of two methods: *SIMD-aware weight pruning* and *node pruning*. It creates a pruned network that can be efficiently executed on the target hardware platform. For hardware with low parallelism like microcontrollers, SIMD-aware weight pruning removes redundant weights but forces the remaining weights to be in groups. All groups are sized to the SIMD width, thereby, improving performance by ensuring the SIMD units are fully utilized. It also decreases the remaining model size since weights in the same group can share the same column index. For hardware with high parallelism like GPUs, node pruning removes redundant nodes in DNNs by using mask layers to dynamically find out and remove unimportant nodes. Removing nodes maintains the dense format of weight matrices, so the computation will not suffer from the sparsity caused by traditional weight pruning methods. For hardware platforms with moderate parallelism like desktop CPUs, SIMD-aware weight pruning and node pruning can be synergistically combined and applied together to reduce both execution time and model sizes. The pruned DNN models generated by Scalpel do not suffer a loss in prediction accuracy compared with the original models.

This paper makes the following contributions:

- We demonstrate that DNN weight pruning is not a panacea, but rather its impact is closely coupled to both the structure of the network (fully-connected vs. convolutional layers) as well as the data-parallel structure of the target hardware. The blind application of traditional weight pruning often results in performance loss, hence a closer examination of this topic is warranted.
- We propose Scalpel that creates a pruned network that is customized to the hardware platform that it will execute. Scalpel provides a method to improve the computation speed and reduce the model sizes of DNNs across processors ranging from microcontrollers to GPUs with no accuracy loss.
- SIMD-aware weight pruning is introduced as a refinement to traditional weight pruning. It puts contiguous weights into groups of size equal to the SIMD width. Extra data for recording the sparse matrix format is reduced along with providing higher utilization of SIMD units.
- A new method, node pruning, is proposed to compress the DNN model by removing redundant nodes in each layer. It does not break the regular structure of DNNs, thus avoiding the overheads of sparsity caused by existing pruning techniques.
- We compare the performance of Scalpel to prior DNN pruning techniques across three hardware platforms: microcontroller, CPU and GPU. Across these hardware platforms, Scalpel achieves geometric mean performance speedups of 3.54x, 2.61x, and 1.25x while reducing the model sizes by 88%, 82%, and 53%. In comparison, traditional weight pruning achieves mean speedups of 1.90x, 1.06x, 0.41x across the three platforms.

2 BACKGROUND AND MOTIVATION

Deep neural networks (DNNs) often include a significant amount of redundant weights. Weight pruning with retraining can safely remove those redundant weights with no reduction in the prediction accuracy.

2.1 DNN Weight Pruning

The fundamental building block of all DNNs is the neuron. A neuron combines its weighted inputs and the bias value to determine the output activation via the activation function. DNNs integrate convolutional (CONV) layers and fully-connected (FC) layers into an end-to-end multi-layer network [22]. Figure 1 shows a typical DNN used for image classification. It consists of two CONV layers followed by two FC layers. In FC layers, all input values are connected to every neuron. For CONV layers, as shown in Figure 1, they consist of a stack of 2D matrices named feature maps. The

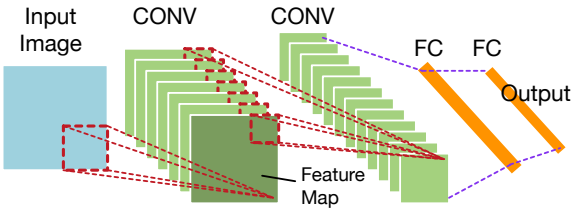

Figure 1: Deep Neural Networks (DNNs) structure. DNNs integrate convolutional layers (CONV) and fully-connected layers (FC) into an end-to-end structure.

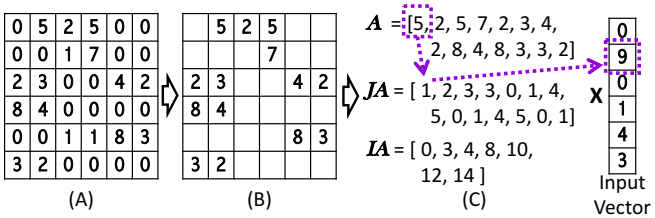

Figure 2: (A) Dense weight matrix; (B) Sparse weight matrix; (C) Compressed Sparse Rows (CSR) format for sparse matrices.

convolution operation is performed between the input feature maps and the weights to generate the output.

FC layers perform matrix-vector multiplication, and CONV layers perform matrix-matrix multiplication. Weights of each layer can be grouped into a weight matrix. For FC layers, the input values are stored in a 1D vector (input vector). Then, the weight matrix can be multiplied with the input vector to generate the output which is also a 1D vector. For each CONV layer, its input is effectively a 3D array. The image-to-column (im2col) function will rearrange the 3D array into a 2D matrix (input matrix). Then, the weight matrix will be multiplied with the input matrix to generate an output matrix which can be converted back to a 3D array for the next layer.

As an example, FC layers perform the computation

$$\mathbf{Y} = f(\mathbf{W} \cdot \mathbf{X}) \tag{1}$$

where $\mathbf{Y}$ is the output activation vector, $\mathbf{W}$ is the weight matrix and $\mathbf{X}$ is the input vector. f is the element-wise non-linear activation function. Bias values can be appended to weights matrix with corresponding input values equal to 1 and, therefore, are neglected.

As shown in Figure 2, weight pruning removes redundant weights and the dense weight matrix $\mathbf{W}$ (Figure 2(A)) is converted into a sparse matrix $\mathbf{W}^{sparse}$ (Figure 2(B)). Every output element $y_i \in \mathbf{Y}$ should be calculated as

$$y_i = f(\sum_{w_{ij}^{sparse} \neq 0, j \in [0,n-1]} w_{ij}^{sparse} x_j) \tag{2}$$

where multiply-accumulate operations with zero weight have been removed. n is the number of columns in $\mathbf{W}$.

After weight pruning, the input vector or input matrix is still dense for each layer. The FC and CONV layers need to perform sparse matrix-vector and sparse matrix-matrix multiplication, respectively.

Deep Compression [19, 20] provides a typical weight pruning technique. Weights with absolute values lower than the thresholds are removed, and the remaining network is retrained. The steps of removing weights and retraining are iteratively applied to generate the pruned DNN model.

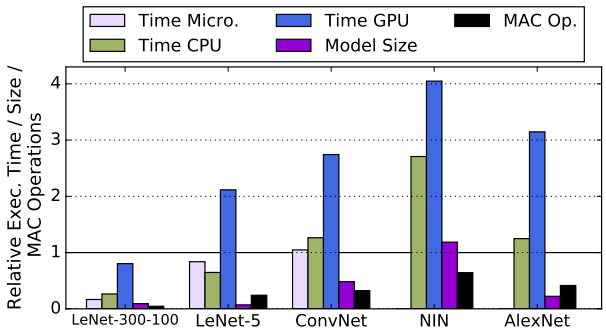

Figure 3: Relative execution time (Time Microcontroller/ CPU/ GPU), model sizes, and MAC operations of the networks pruned with the traditional pruning technique in Deep Compression. NIN and AlexNet are not tested on the microcontroller due to its limited storage size.

2.2 Challenges

Weight pruning can dramatically decrease DNN model sizes and the number of MAC operations. However, several challenges exist for traditional DNN pruning techniques.

The first challenge is that the sparse weight matrix spends too much extra data to record the sparse matrix format. For example, as shown in Figure 2 (C), the compressed sparse rows (CSR) format use three 1-D arrays (**A**, **IA**, **JA**) to store a $m \times n$ matrix **W**. **A** holds all the nonzero values. **IA** records the index into **A** of the first nonzero element in each row of **W**. **JA** stores the column indexes of the nonzero elements. Since the index array **JA** has the same size with the data array **A**, more than half of the data in the CSR format is used to store the sparse matrix format.

The second challenge is that weight pruning can hurt the DNN computation performance. Figure 3 shows the relative execution time, model sizes and MAC operations of network models pruned by the weight pruning method in Deep Compression with respect to the original DNN models. The first three bars show the relative execution time on the microcontroller, CPU and GPU. As shown in the figure, the relative execution time is much higher than the relative model sizes and MAC operations. Weight pruning hurts the performance of LeNet-5 (on GPU), ConvNet, NIN and AlexNet, which causes an execution time increase for these networks.

We generate a breakdown of the execution time for the dense and the sparse DNN models on the CPU, as shown in Figure 4. The expected execution time of the sparse network (Expected) shows the relative MAC operations remaining, which ignores the overheads of the sparse storage implementation. Execution times are presented relative to the unpruned baseline (Dense). The real execution time of the FC layers is significantly reduced by traditional weight pruning, which is similar to the results shown in the previous work [19]. For both FC and CONV layers, there is a large gap between the real execution time of the pruned networks and the expected values. The performance gains are lagging significantly behind the reduction in MAC operations. Lastly, weight pruning is ineffective for CONV

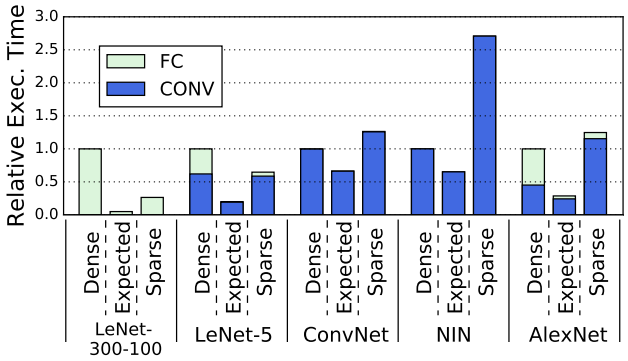

Figure 4: Execution time breakdown of the original networks (Dense) and the pruned networks (Sparse). The expected execution time of pruned networks (Expected) shows the relative MAC operations remaining. The second CONV layer (*conv2*) in AlexNet is tested.

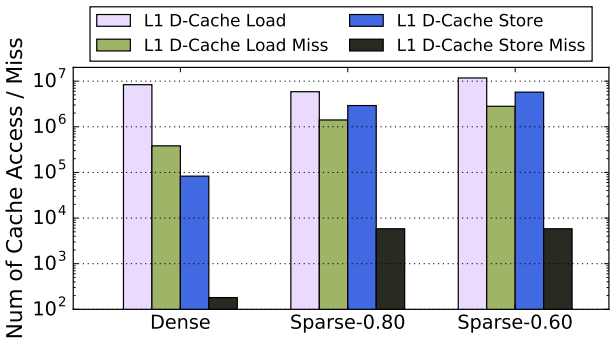

Figure 5: Numbers of cache access and cache miss in the computation of the second CONV layer (*conv2*) in AlexNet on Intel Core i7-6700 CPU. The original dense layer (Dense), the sparse layer with 80% (Sparse-0.80) and 60% (Sparse-0.60) of weights removed are tested. The matrices are randomly generated.

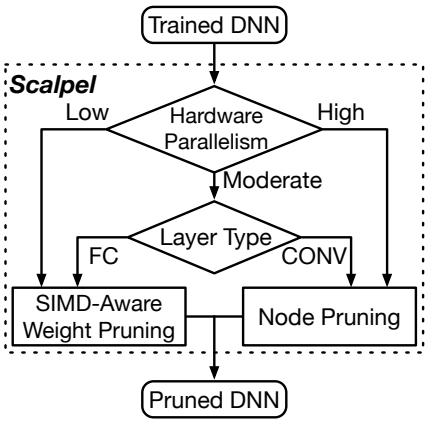

Figure 6: Overview of Scalpel.

layers, leading to substantial increases in execution time for ConvNet, NIN and AlextNet.

The gap between real and expected execution times occurs because the sparse weight matrices lose the regular structure of their dense counterparts. For example, all rows have the same size in dense matrices, which does not hold for sparse matrices. Assume the sparse weight matrix is stored in CSR format. To perform sparse matrix-vector multiplication, as the dashed lines in Figure 2 show, for each nonzero weight stored in array **A**, we first need to load the corresponding column index from **JA** and use that to load the input value from the input vector. This indexing of input values requires additional computation and memory accesses. Worse yet, since sparse matrices lose the regular structure, many optimizations applicable to dense matrices, e.g. matrix tiling, cannot be applied. For CPU computation, FC layers do not suffer as much from these reasons because they have a much lower computation/memory access ratio compared with CONV layers.

To understand the impact of pruning on memory access behavior in more detail, we consider a representative layer from AlexNet, the second layer (*conv2*). This layer performs a matrix-matrix multiplication because it is a CONV layer. Figure 5 shows the numbers of cache accesses and cache misses for this layer broken down by loads and stores. We measure three cases: original dense layer (Dense), sparse layer with 80% (Sparse-0.80) and 60% (Sparse-0.60) of the weights removed. We profile the computation with Callgrind. Sparse-0.60 represents the actual pruning rate for this layer, while Sparse-0.80 represents the pruning rate necessary to reach the break-even performance point. As shown in the figure, weight pruning leads to large increases in L1 D-Cache load misses, stores, and store misses due to the sparse representation. With 60% pruning, these overheads are not counter-balanced by the computation reduction, hence a net performance loss is observed. At 80% pruning, the combination of modestly lower memory access overheads and higher reductions in computations results in break-even performance. Pruning rates of more than 80% are necessary to overcome the memory access overhead for this layer and achieve a net performance gain.

3 SCALPEL

To address the challenges from traditional pruning techniques, we propose Scalpel in this paper. It consists of two methods: SIMD-aware weight pruning and node pruning. Scalpel customizes the DNN pruning for different hardware platforms based on their parallelism.

3.1 Overview

Figure 6 shows the overview of Scalpel. The first step of Scalpel is profiling and determining the parallelism level of the hardware platform. All general-purpose hardware platforms are divided into three categories based on their internal parallelism: low parallelism, moderate parallelism, and high parallelism.

For low-parallelism hardware, SIMD-aware weight pruning is applied. It prunes weights in groups and forces the remaining weights to be in aligned groups. All groups have the same size as the SIMD width and the weights in the same group share the same column index, reducing the overhead of the sparse format.

For high-parallelism hardware, node pruning is applied. It removes the DNN redundancy by removing redundant nodes instead of redundant weights. Removing nodes does not break the regular structure of dense weight matrices.

Table 1: Hardware platforms with different parallelism.

	Parallelism		
	Low	Moderate	High
Example	Micro-controller	CPU	GPU
Memory Hierarchy	No cache	Deep cache hierarchy	High bandwidth / long latency
Memory Size	~100KB SRAM	~8MB SRAM	2-12GB DRAM

For hardware with moderate parallelism, we use a combination of SIMD-aware weight pruning and node pruning. SIMD-aware weight pruning is employed for fully-connected layers and node pruning is applied to convolutional layers.

By customizing pruning technique for different hardware platforms, Scalpel can reduce both the DNN model size and execution time across all the general-purpose processors without accuracy loss for DNNs.

3.2 Hardware with Different Parallelism

We divide general-purpose processors into three main categories based on their parallelism level. Table 1 demonstrates the basic characteristics of these three categories.

Low Parallelism: Low-power processors like microcontrollers usually have a low parallelism. These processors contain in-order cores with a "shallow" pipeline and have no cache. They also have very limited storage. SIMD units are employed to accelerate the computation, but their width is still limited. As an example, ARM Cortex-M4 has a 3-stage in-order pipeline and a 2-way SIMD unit. The test board we use only has 128KB SRAM and 512KB flash.

Moderate Parallelism: Out-of-order processors, for example Intel Core i7-6700 CPU, can be classified as moderate-parallelism hardware. In addition to SIMD units, the instruction-level parallelism (ILP) and the memory-level parallelism (MLP) are utilized to accelerate the computation. To fully utilize ILP and MLP, moderate-parallelism processors require a deep cache hierarchy. They are usually connected to a large off-chip DRAM and, therefore, the storage size is sufficiently large to be considered unlimited.

High Parallelism: High-parallelism hardware like GPUs exploits thread-level parallelism (TLP) to further improve the parallelism to accelerate the computation. It focuses on high computation throughput and, therefore, DNN model sizes are not critical. Applications for high-parallelism hardware tend to be bandwidth sensitive, which requires a memory hierarchy with high bandwidth. Like moderate-parallelism hardware, the storage size can be considered unlimited.

Multiple inputs for the same DNN can be processed in a batch to reduce the memory access. The weight matrices can be loaded once and shared for the computation of multiple different inputs. Batch processing will increase the computation throughput but also the latency. On low-parallelism and moderate-parallelism hardware, we set the batch size to 1 because real-time applications need the DNN computation to be completed within a short latency. However, for high-parallelism hardware, the computation is throughput-driven, and DNN computation with a large batch size can still meet the latency requirement. In this case, we set the batch size to 50 for DNN computation on high-parallelism hardware. Han et al. [19] set batch size to 1 for GPU testing, which is actually unpractical.

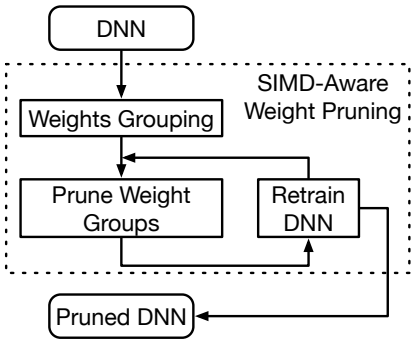

Figure 7: Main steps of SIMD-aware weight pruning.

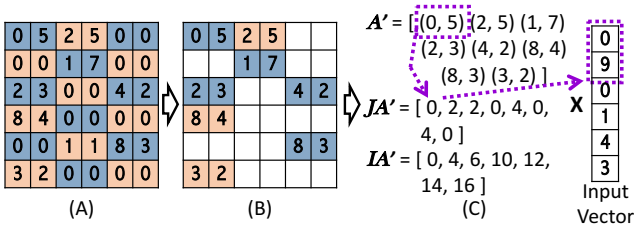

Figure 8: (A) Weights grouping; (B) Sparse weight matrix after pruning weight groups; (C) Modified CSR format for SIMD-aware weight pruning.

3.3 SIMD-Aware Weight Pruning

For low-parallelism hardware, we apply SIMD-aware weight pruning to DNNs. The main steps of SIMD-aware weight pruning are shown in Figure 7. We use ARM Cortex-M4 microcontroller as an example of low-parallelism hardware. It has a 2-way SIMD unit for 16-bit fixed-point numbers.

The first step is weights grouping. All the weights are divided into aligned groups with the same size equal to the supported SIMD width. Figure 8 (A) shows a simple example of weights grouping. All groups have a size of 2 which is the SIMD width of Cortex-M4.

The second step is pruning weight groups. We calculate the Root-Mean-Square (RMS) of each group and use it to measure the importance of weight groups. Groups with RMS value below a threshold are removed. Figure 8 (B) shows an example of the weight matrix after pruning weight groups. Then the pruned weight matrix will be retrained. The steps of pruning weight groups and retraining DNN are iteratively applied until the retrained DNN cannot keep the original accuracy.

SIMD-aware weight pruning works layer by layer. The execution time for each layer will be generated at the beginning, and the pruning process starts with the layer of the highest execution time. Every time after retraining the pruned DNN, the execution time of each layer will be updated. The new slowest layer will be pruned in the next iteration if the retrained DNN does not lose the original accuracy. We will not prune the layers which have low redundancy and cannot get a performance improvement through the SIMD-aware weight pruning.

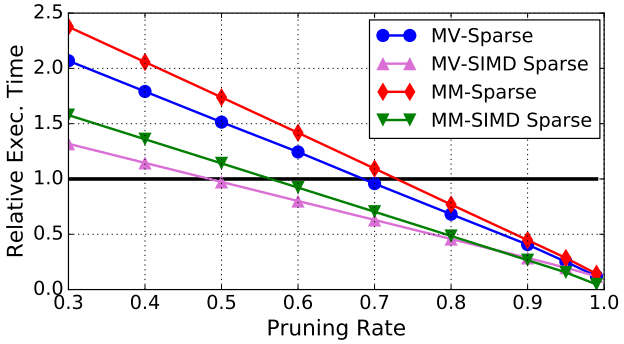

Figure 9: Relative execution time of sparse matrix multiplication on ARM Cortex-M4 with respect to the original dense matrix-vector or matrix-matrix multiplication. MV/MM-Sparse show the results for sparse Matrix-Vector (MV) and Matrix-Matrix (MM) multiplication, respectively. MV/MM-SIMD Sparse shows the corresponding performance with nonzero elements grouped and aligned as SIMD-aware weight pruning does. All matrices have the size of 100 x 100 and are randomly generated.

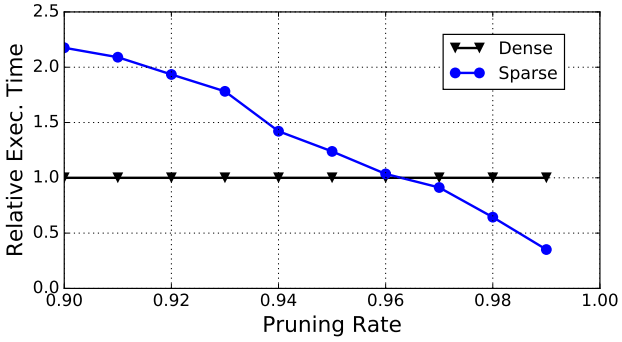

Figure 10: Relative execution time for sparse matrix-matrix multiplication (Sparse) on NVIDIA GTX Titan X with respect to original execution time (Dense). The matrices have the sizes of 4096x4096 and 4096x50.

During SIMD-aware weight pruning, we need to adjust the dropout ratio. Dropout is a widely used technique for preventing overfitting [39]. During network training, dropout is implemented by keeping a neuron active with some probability p, or setting it to zero otherwise. This procedure can be regarded as sampling the neural network, and only the sampled part of the network needs to be updated through this iteration of training. For next iteration, the network should be re-sampled. SIMD-aware weight pruning will remove connections and reduce the DNN model capacity. We use the same technique with Han et al. [20] to adjust the dropout ratio. Assuming for the layer i, C_i is the number of the connections where C_{io} is for the original network and C_{ir} is for the remaining network. We can adjust the dropout ratio as

$$D_r = D_o \sqrt{\frac{C_{ir}}{C_{io}}} \tag{3}$$

where D_o is the original dropout ratio and D_r is the adjusted dropout ratio for retraining the remaining network after pruning weight groups.

After SIMD-aware weight pruning, we use a modified CSR format to record the sparse weight matrices. The modified CSR format, shown in Figure 8(C), includes three 1-D arrays: $\mathbf{A}'$, $\mathbf{IA}'$ and $\mathbf{JA}'$. $\mathbf{A}'$ stores all the nonzero weight groups with the original order. $\mathbf{IA}'$ records the index into $\mathbf{A}'$ of the first nonzero element in each row of $\mathbf{W}$. $\mathbf{JA}'$ stores the column index of each group. Only the column index of the first element in each group is recorded. In real computation, as the dashed arrows in Figure 8 show, we can load the nonzero weights in the array $\mathbf{A}'$ in groups. Then only one index from array $\mathbf{JA}'$ is used to load the corresponding input values. Since the input values are now also in contiguous addresses, they can be loaded with a single SIMD instruction. With the input values and weights loaded, the SIMD unit then performs the computation.

SIMD-aware weight pruning can reduce both model sizes and execution time of DNNs on low-parallelism hardware. Using one column index for each weight group can dramatically reduce the storage size of the indexes array $\mathbf{JA}'$ and the entire model size. For DNN computation, loading multiple contiguous input values with one SIMD instruction can reduce the computation instructions. The reduction in model size can also reduce the memory footprint. Therefore, the DNN computation performance can be improved with SIMD-aware weight pruning.

Figure 9 shows the peak performance benefit from SIMD-aware weight pruning. The x-axis is the pruning rate which means how much weights we can remove from the weight matrix. For sparse matrix-vector (MV-Sparse) and matrix-matrix (MM-Sparse) multiplication, we need to remove more than 68% and 73% of the weight matrix to decrease the execution time, respectively. However, with SIMD-aware weight pruning (MV-SIMD Sparse / MM-SIMD Sparse), we only need to remove 48% and 56% of the weights.

3.4 Node Pruning

For hardware with high parallelism, node pruning is employed to remove the redundancy in DNNs. We use NVIDIA GTX Titan X GPU as an example of high-parallelism hardware.

Traditional weight pruning techniques will decrease the performance of all DNN layers on high-parallelism hardware. Figure 10 shows the relative execution time of sparse matrix-matrix multiplication on GPU against the pruning rate. The two matrices have the sizes of 4096x4096 and 4096x50. It estimates the performance of a fully-connected layer with 4096 inputs and 4096 outputs. The computation batch size is set to 50. As shown in the figure, more than 96% of the weights need to be removed to achieve a performance speedup. However, without a loss of accuracy, it is unpractical to remove that much weights from DNN layers. The matrix sparsity caused by weight pruning will hurt the computation performance of all layers.

To avoid this performance decrease, node pruning removes DNN redundancy by removing entire nodes instead of weights. It uses mask layers to dynamically find out unimportant nodes and block their outputs. The blocked nodes are removed after the training of mask layers. After removing all redundant nodes, mask layers are removed, and the network is retrained to get the pruned DNN model.

One neuron in the fully-connected layers or one feature map in the convolutional layers is considered as one node. Removing nodes in

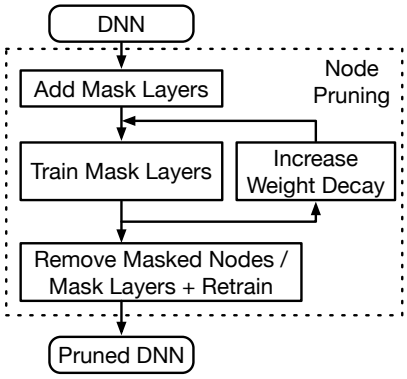

Figure 11: Main steps of node pruning.

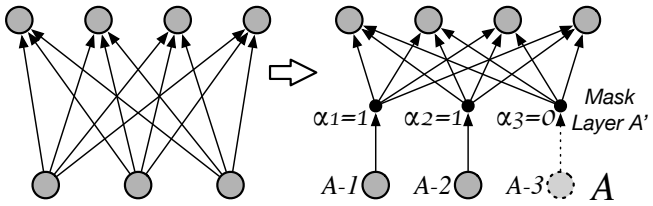

Figure 12: Mask layers. Node *A-3* with $\alpha_3 = 0$ can be removed. The whole mask layer A' will be removed after pruning all redundant nodes.

DNNs only shrinks the size of each layer but will not incur sparsity into the network. The remaining DNN model after node pruning keeps the regular dense DNN structure and will not suffer from the overheads of network sparsity.

Figure 11 shows the main steps of node pruning. First, we will add a mask layer for each DNN layer except the input and output layers. Figure 12 gives an example of a mask layer for fully-connected layers. The output values of layer A need to go through the mask layer A' before propagated to the next layer. Each node in the mask layer holds two parameters α and β. α is a boolean variable ($\alpha \in \{0, 1\}$) and β is a floating number between 0 and 1 ($0 \leqslant \beta \leqslant 1$). Let array $\mathbf{Y}$ and $\mathbf{Y}'$ to be the output activation array of the original layer A and the mask layer A'. For $y'_i \in \mathbf{Y}'$ and $y_i \in \mathbf{Y}$, we have

$$y'_i = \alpha_i \cdot y_i \tag{4}$$

With α_i set to 0, the corresponding node can be considered as removed because the output y'_i is fixed to 0.

For convolutional layers, the activations in the feature map i will go through the same mask node i and produce a masked feature map keeping the same size. Therefore, convolutional layers are pruned at a granularity of feature maps, and we are considering each feature map as one node.

The next step is to train the mask layers. For a single node i in the mask layer, α_i and β_i are both initialized to 1:

$$\alpha_i|_0 = 1, \quad \beta_i|_0 = 1.0 \tag{5}$$

In training iteration $k \geqslant 1$, α_i is calculated as

$$\alpha_i|_k = \begin{cases} 1, & T + \varepsilon \leqslant \beta_i|_k \\ \alpha_i|_{k-1}, & T \leqslant \beta_i|_k < T + \varepsilon \\ 0, & \beta_i|_k < T \end{cases} \tag{6}$$

where T ($0<T<1$) is a threshold shared by all the mask layers. ε is a small value to make training more stable that α_i will not "ping pong" between 0 and 1. β_i is updated through back-propagation and truncated to $[0, 1]$.

We use the L1 regularization to control the number of nodes got removed. Regularization is used to penalize the magnitude of parameters. For the mask layer, the penalty of each parameter β_i can be calculated as

$$R_{i,\,L1} = \lambda |\beta_i| = \lambda \beta_i \tag{7}$$

where λ is the weight decay (regularization strength). It forces β_i to be close to zero. If the corresponding node is not important, β_i will be decreased to be lower than threshold T and the node is temporarily removed. In case some removed nodes are found important, they will be retained through the DNN training. Since the threshold T is fixed in node pruning, increasing the weight decay λ will increase the penalty for β_i and decrease more parameters β to be lower than T. More nodes will, therefore, be removed.

Node pruning also needs to adjust the dropout ratio. Different from SIMD-aware weight pruning, the dropout ratio is dynamically updated during the step of training mask layers. In iteration k, the dropout ratio is calculated as

$$Dropout|_k = Dropout|_0 \times \frac{N|_k}{N|_0} \tag{8}$$

where $Dropout|_0$ is the initial dropout ratio, $N|_0$ is the initial number of nodes and $N|_k$ is the number of remaining nodes in iteration k.

In the step of training mask layers, the parameters of other layers are not fixed. Weights and biases in other layers are trained to fit the new DNN architecture with nodes removed.

After training mask layers, the weight decay of L1 regularization on the mask layers is increased. Then more nodes will be removed in the next iteration of training mask layers. The two steps of training mask layers and increasing weight decay will be iteratively applied until retraining cannot retain the DNN accuracy.

The last step of node pruning is removing masked nodes, removing mask layers, and retraining the network. All the nodes and feature maps with corresponding α value equal to zero are removed. For example, in Figure 12, the node *A-3* with $\alpha_3 = 0$ will be removed. The mask layers are then removed, and output activations of remaining nodes can be directly propagated to the next layer. The remaining network is retrained to get the final pruned DNN.

3.5 Combined Pruning

For hardware with moderate parallelism, for example Intel Core i7-6700 CPU, the SIMD-aware weight pruning and node pruning can be combined and applied to the same DNN. To limit the computation latency for real-time application, DNN computation usually has a small batch size on moderate-parallelism hardware. Here the batch size is fixed to 1, and a small batch size will give similar results.

DNNs can be split into two parts: fully-connected layers and convolutional layers. With a batch size of 1, fully-connected layers perform matrix-vector multiplication, and convolutional layers perform matrix-matrix multiplication. In this case, Scalpel applies SIMD-aware weight pruning to fully-connected layers and node pruning to convolutional layers.

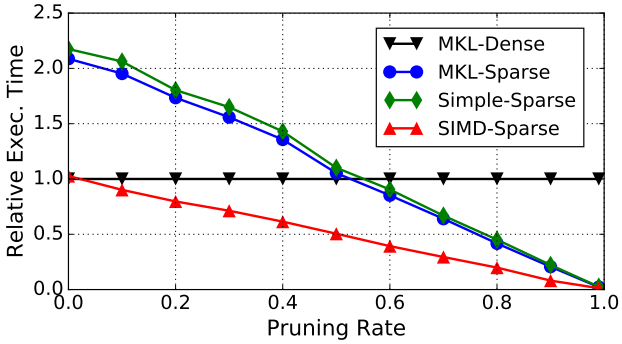

Figure 13: Relative execution time for sparse matrix-vector multiplication (FC layers) on Intel Core i7-6700. The matrix size is 4096 x 4096 and the vector size is 4096. MKL-Dense/Sparse show the results of dense and sparse weight matrix with the Intel MKL library.

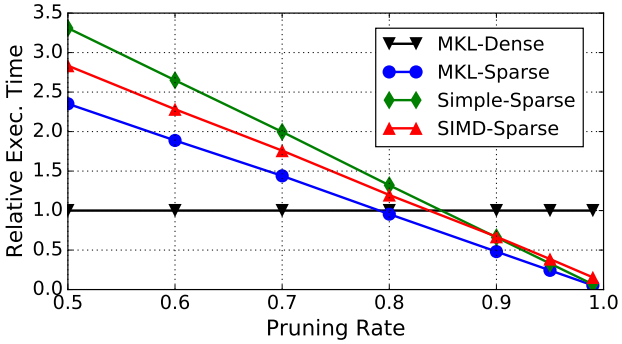

Figure 14: Relative execution time for sparse matrix-matrix multiplication (CONV layers) on Intel Core i7-6700. The weight matrix and input matrix have the size of 128 x 1200 and 1200 x 729, respectively.

Figure 13 and Figure 14 profiles the relative execution time for sparse matrix-vector and matrix-matrix multiplication on Intel Core i7-6700 CPU, respectively. MKL-Dense and MKL-Sparse show the results of dense and sparse weight matrix with the Intel MKL library. Simple-Sparse is our implementation of the sparse library. SIMD-Sparse shows the results with nonzero elements grouped and aligned as in SIMD-aware weight pruning.

Figure 13 shows the relative execution time reduction of matrix-vector multiplication with SIMD-aware weight pruning. Intel i7-6700 has an 8-way SIMD unit for 32-bit floating-point numbers. Therefore, SIMD-aware weight pruning removes weights in groups of 8. Comparing to the sparse matrix-vector multiplication with the Intel MKL library (MKL-Sparse), the SIMD-aware weight pruning (SIMD-Sparse) can dramatically improve the computation performance. The percentage of weights need to be removed for performance speedup decreases from 52% to 3%. Therefore, for fully-connected layers which perform matrix-vector multiplication, the SIMD-aware weight pruning can be applied to improve the performance and reduce model sizes.

SIMD-aware weight pruning can dramatically decrease the execution time on moderate-parallelism hardware for three reasons. First, by reducing the number of indexes, the memory footprint for the computation decreases. Second, weights are grouped and aligned with SIMD-aware weight pruning, which increases the spatial locality of reading weights and corresponding inputs. Third, the number of computation instructions decreases since we only need to load one index for each group and the corresponding input values can be loaded with SIMD instructions.

Figure 14 is the corresponding relative execution time reduction for matrix-matrix multiplication. Compared to the sparse matrix-matrix multiplication with the Intel MKL library (MKL-Sparse), SIMD-aware weight pruning (SIMD-Sparse) cannot improve the execution performance. To achieve a performance speedup with traditional pruning technique or SIMD-aware weight pruning, at least 79% of the weights need to be removed. However, without accuracy loss, it is difficult to remove that much weights from convolutional layers for DNNs running on moderate-parallelism hardware. Convolutional layers have much less redundancy compared with full-connected layers since all weights need to be reused in the convolution operations. Therefore, weight pruning will hurt the computation performance and cannot be applied to convolutional layers. In this case, Scalpel applies node pruning to convolutional layers since it keeps the regular structure of weight matrices to avoid the performance decrease caused by weight pruning.

To apply both SIMD-aware weight pruning and node pruning to the same network, we will first use node pruning to remove redundant nodes in the convolutional layers. Then the convolutional layers are fixed, and the SIMD-aware weight pruning is employed to prune redundant weights from the fully-connected layers.

4 EXPERIMENT METHODOLOGY

Hardware platforms We test Scalpel on three different general-purpose processors: microcontrollers, CPU and GPU. They are representatives of hardware with low, moderate and high parallelism, respectively.

1) Microcontroller - low parallelism. We use ARM Cortex-M4 microcontroller which has a 3-stage in-order pipeline and 2-way SIMD units for 16-bit fixed-point numbers. The test board has 128KB SRAM and 512KB flash storage. To run the benchmarks, we use libraries directly from ARM for the dense matrix-vector/matrix multiplication. Libraries for sparse matrix-vector/matrix multiplication are written in-house.

2) CPU - moderate parallelism. We use Intel Core i7-6700 CPU, which is a Skylake class core. It supports 8-way SIMD instructions for 32-bit floating-point numbers. To run the benchmarks, we use MKL BLAS GEMV/ GEMM to implement the original dense model and the convolutional layers with node pruning. MKL Sparse BLAS CSRMV/ CSRMM is used for the sparse models generated by existing weight pruning techniques. Libraries of sparse matrix multiplication for SIMD-aware weight pruning are written in-house.

3) GPU - high parallelism. We use NVIDIA GTX Titan X which is a state-of-the-art GPU for deep learning and included in the NVIDIA Digits Deep Learning DevBox machine [1]. We use cuDNN to implement the original dense model and the convolutional layers with node pruning. cuBLAS GEMV/ GEMM is used for profiling the

Table 2: DNN benchmarks.

Networks	Num of Layers		Test Dataset	Error Rate
	CONV	FC		
LeNet-300-100	0	3	MNIST	1.50%
LeNet-5	2	2		0.68%
ConvNet	3	1	CIFAR-10	18.14%
NIN	9	0		10.43%
AlexNet	5	3	ImageNet	19.73% (top-5)

Table 3: Results overview.

Hardware	DNNs	Speedup	Relative Size
Micor-controller	LeNet-300-100	9.17x	6.93%
	LeNet-5	3.51x	6.72%
	ConvNet	1.38x	40.95%
CPU	LeNet-300-100	6.86x	7.08%
	LeNet-5	4.15x	5.20%
	ConvNet	1.58x	44.28%
	NIN	1.22x	81.16%
	AlexNet	2.20x	13.06%
GPU	LeNet-300-100	1.08x	66.83%
	LeNet-5	1.59x	11.67%
	ConvNet	1.14x	45.40%
	NIN	1.17x	81.16%
	AlexNet	1.35x	76.52%

performance of the dense matrix-vector/matrix multiplication. cuSPARSE CSRMV/ CSRMM is used for the sparse model generated by existing weight pruning techniques.

Benchmarks. We compare the performance and the model size of five DNNs: LeNet-300-100 [32], LeNet-5 [32], ConvNet [28], Network-in-Network (NIN) [34] and AlexNet [29].

Table 2 shows the DNN benchmarks and their structures. NIN consists of 9 convolutional layers (CONV) and no fully-connected layers (FC). Recent DNN designs [22, 34, 40] contain no or only small-size fully-connected layers to reduce the model size. NIN is chosen as an example to test the performance of Scalpel on those networks. For NIN and AlexNet, we get the original models from Caffe Model Zoo [26]. Also, due to their large sizes, NIN and AlexNet are not tested on ARM Cortex-M4 microcontroller.

All pruned DNN models generated with Scalpel have *no accuracy loss* compared with the original DNNs. Caffe [26] is used for DNN training and pruning.

Experiment baselines. We use the original dense DNN models as the baseline. Performance speedups and model sizes shown in section 5 are relative values with respect to the corresponding original DNN models.

We compare Scalpel to two weight pruning techniques:

1) Traditional pruning. The pruning technique proposed by Han et al. [20] is implemented as the traditional pruning.

2) Optimized pruning. After traditional pruning, for layers with more than 50% of the weights remaining, we revert them back to the original dense format. We keep the nonzero weights and add zeros to the sparse weight matrix to convert it back into a dense matrix. This technique combined with traditional pruning is considered as

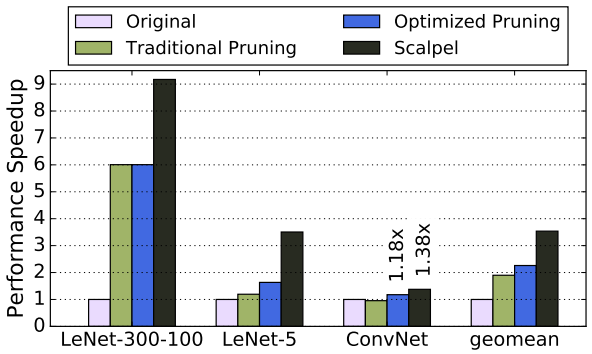

Figure 15: Relative performance speedups of the original models (original), traditional pruning, optimized pruning and Scalpel on ARM Cortex-M4 microcontroller. NIN and AlexNet are not tested due to the limited storage size of the microcontroller.

the optimized pruning. Recording the layers with more 50% weights remaining into a sparse format will increase the model size. Also, not enough weights are removed to achieve a performance improvement for these layers. Therefore, optimized pruning can reduce both the model size and the execution time compared with the traditional pruning.

5 EVALUATION RESULTS

Table 3 is the overview of the results. Scalpel achieves mean speedups of 3.54x, 2.61x, and 1.25x on the microcontroller, CPU and GPU, respectively. It also reduces the model sizes by 88%, 82%, and 53% on average.

5.1 Microcontroller - Low Parallelism

Scalpel is first tested on ARM Cortex-M4 microcontroller which is considered as the low-parallelism hardware. SIMD-aware weight pruning is applied to LeNet-300-100, LeNet-5, and ConvNet. Since Cortex-M4 has a 2-way SIMD unit, the size of weight groups is set to 2.

Figure 15 and Figure 16 show the relative performance speedups and relative model sizes of the original models, traditional pruning, optimized pruning and Scalpel.

For all tested networks, Scalpel achieves better performance and lower model sizes than traditional pruning and optimized pruning. The performance speedup can be up to 9.17x, and the model size is reduced by up to 93.28%.

Traditional pruning and optimized pruning have the same performance speedups and model sizes on LeNet-300-100. This is because LeNet-300-100 is a fully-connected network and all layers can have more than 50% of weights removed. But for ConvNet, the first convolutional layer has a relatively small size of 2400 weights. There is little redundancy inside this layer, and few weights can be removed. Therefore, it is not pruned in the optimized pruning, which helps improve the performance and reduce the model size.

How to measure the importance of each group is important for SIMD-aware weight pruning since weight groups with low importance will be removed in the step of pruning weight groups. We

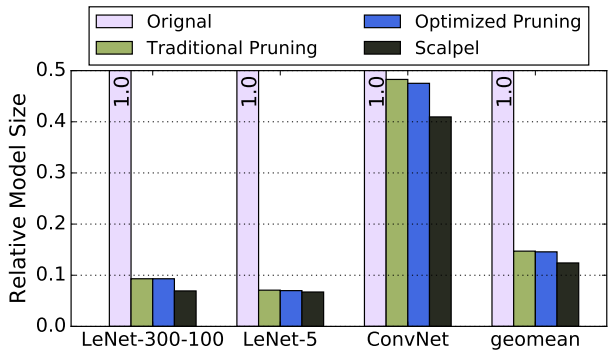

Figure 16: Relative model sizes of the original models, traditional pruning, optimized pruning and Scalpel for ARM Cortex-M4 microcontroller.

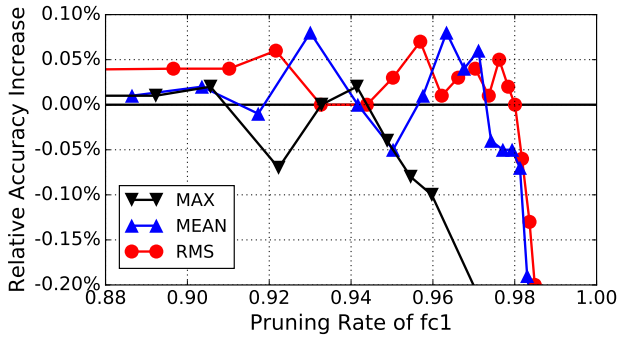

Figure 17: Relative accuracy against pruning rate with different metrics for importance measurement: maximum absolute value (MAX), mean absolute value (MEAN) and root-mean-square (RMS).

test three different types of metrics for importance measurement: maximum absolute value (MAX), mean absolute value (MEAN) and root-mean-square (RMS).

In these metrics, MAX only takes the weight with highest absolute value into consideration, but all weights in the group should be considered to determine the group importance. For MEAN, large weight values and small weight values have the same impact on the group importance. However, in real DNN, larger weight tend to be more important and should have a larger impact. RMS considers all the weight values, and larger weight value will have a larger impact on the group importance. Therefore, compared to the other two metrics, RMS is expected to help SIMD-aware weight pruning remove more redundant weights.

We apply SIMD-aware weight pruning on the first layer of LeNet-300-100 (*fc1*) with these three different metrics to test the real effect. The group size is chosen to be 2. Figure 17 shows the curves of relative accuracy against pruning rate for the three metrics. For each line, every dot is the result of one iteration for the SIMD-aware weight pruning since we are applying the two steps of pruning weight group and increasing weight decay iteratively. As expected, without accuracy loss, using RMS for the SIMD-aware weight pruning has the highest pruning rate of 98.0%.

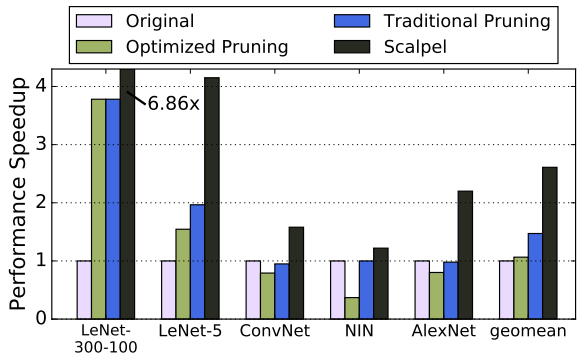

Figure 18: Relative performance speedups of the original models, traditional pruning, optimized pruning and Scalpel on Intel Core i7-6700 CPU.

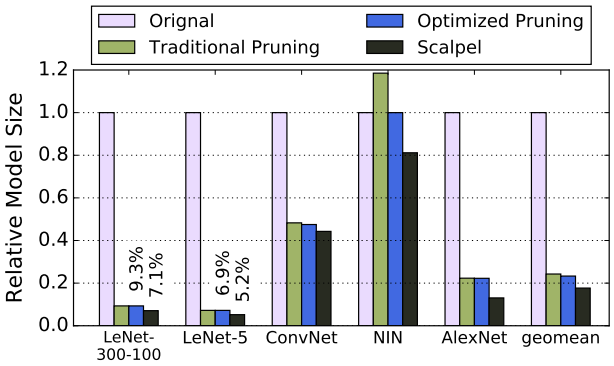

Figure 19: Relative model sizes of the original models, traditional pruning, optimized pruning and Scalpel for Intel Core i7-6700 CPU.

5.2 CPU - Moderate Parallelism

For Intel Core i7-6700 CPU, SIMD-aware weight pruning and node pruning are combined and applied to LeNet-300-100, LeNet-5, ConvNet, NIN and AlexNet. The processor has an 8-way SIMD unit and, therefore, the size of weight groups in SIMD-aware weight pruning is set to 8.

Figure 18 and Figure 19 are the relative performance speedups and relative model sizes of the original models, traditional pruning and Scalpel. For all tested networks, Scalpel achieves better performance and lower model size than traditional pruning and optimized pruning. The performance speedup can be up to 6.86x, and the relative size is reduced by up to 94.8%.

Notice that NIN pruned with optimized pruning holds the same performance and the same model size with the original model. This is because none of the layers in NIN can have more than 50% of weights removed through traditional pruning. Converting to sparse is expected to yield performance loss. Therefore, all the layers are not pruned for NIN with the optimized pruning. For the same reason, traditional pruning hurts the computation performance and increases the model size of NIN.

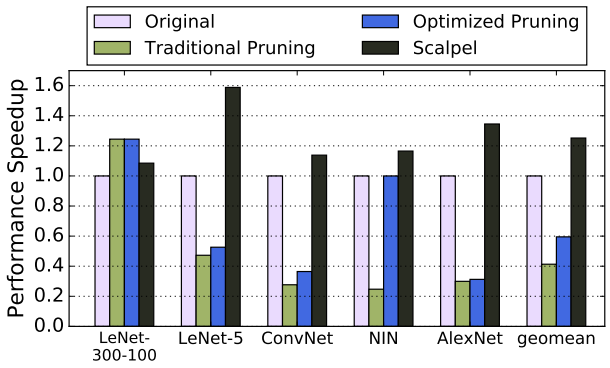

Figure 20: Relative performance speedups of the original models, traditional pruning, optimized pruning and Scalpel on NVIDIA GTX Titan X GPU.

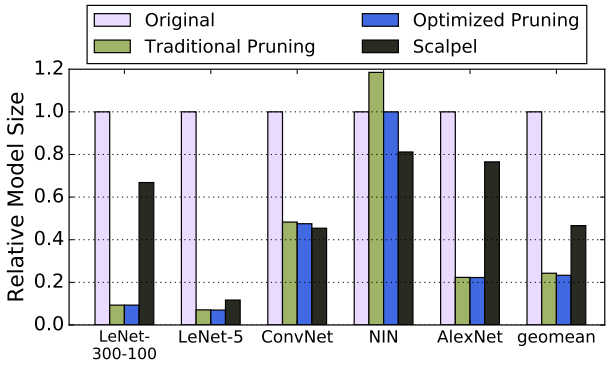

Figure 21: Relative model sizes of the original models, traditional pruning, optimized pruning and Scalpel for NVIDIA GTX Titan X GPU.

5.3 GPU - High Parallelism

For NVIDIA GTX Titan X GPU which is considered as high-parallelism hardware, node pruning is applied to keep the regular structure of DNNs. Figure 20 shows the relative performance speedups of the original models, traditional pruning, optimized pruning and Scalpel. Batch size is set to 50. For LeNet-5, ConvNet, NIN and AlexNet, Scalpel has much higher speedups compared to traditional pruning and optimized pruning. However, for LeNet-300-100, the speedup from Scalpel is lower. The reason is that LeNet-300-100 has very tiny size and it is difficult to map the computation efficiently onto the GTX Titan X GPU. Another possible reason is the matrix tiling strategy in cuBLAS. A small change in the matrix size may lead to a large difference in the detailed matrix tiling strategy. As a result, although we can remove 31% and 32% of the nodes from the first and the second layers in LeNet-300-100, respectively, the computation performance is improved by only 8.50%.

The corresponding relative model sizes are shown in Figure 21. Models generated by Scalpel, except ConvNet and NIN, have higher model sizes than those generated by traditional and optimized pruning. It is because node pruning is applied now and we need to remove DNN redundancy at a granularity of nodes to keep the regular structure. But traditional and optimized pruning can prune DNN redundancy at a finer granularity of weights. Therefore, node pruning cannot reduce the model size as much as traditional pruning and optimized pruning do. However, the high-parallelism hardware is designed for high-throughput, and the DNN model size is not critical. Scalpel is more beneficial than traditional and optimized pruning on high-parallelism hardware since it can dramatically increase the computation throughput.

Table 4: Percentage of nodes removed by node pruning in each layer. Output layers are not included.

DNNs	Percentage of Nodes Removed in Each Layer
LeNet-300-100	31% (fc1)- 32% (fc2)
LeNet-5	50% (conv1)- 68% (conv2)- 65% (fc3)
ConvNet	28% (conv1)- 25% (conv2)- 49% (conv3)
NIN	28% (conv1)- 20% (cccp1)- 5% (cccp2)- 2% (conv2)- 14% (cccp3)- 8% (cccp4)- 22% (conv3)- 48% (cccp5)
AlexNet	3% (conv1)- 20% (conv2)- 24% (conv3)- 18%(conv4)-0%(conv5)-17%(fc6)-23%(fc7)

Table 4 gives the percentage of nodes we can remove from each layer in DNNs. As an example, NIN consists of 9 convolutional layers, and each feature map is considered as one node in the convolutional layers. Layers close to the input or output have more redundancy and tend to be less important than the layers in the middle of NIN. 28% and 48% of the nodes can be removed from the first layer (conv1) and the eighth layer (cccp5), respectively.

6 RELATED WORK

Network compression. Deep Neural Network models have significant redundancy [13] which leads to redundant computation and storage. Various techniques have been proposed to remove the redundancy inside the network models.

Weight pruning has been used to remove redundant connections inside a network. LeCun et al. [31] calculates the saliencies for each weight based on the Hessian matrix of the loss function and remove low-saliency weights. Hassibi et al. [21] employ Lagrange Multiplier to find the weight with low saliencies. However, the calculation of the Hessian matrix of the loss function needs unacceptable computation. Han et al. [20] directly remove low-value weights and retrain the network to keep the original network performance. Guo et al. [16] uses mask matrices to incorporate connection splicing into the entire pruning process, which can help avoid incorrect pruning.

For node pruning, He et al. [23] introduce a DNN reshape method which measures the importance of each node with a certain importance function. It works for fully-connected networks without convolutional layers. Miconi et al. [35] propose a method to make network structure differentiable. It imposes a penalty on the outgoing weights from each neuron to determine the importance of each neuron. However, it only works for simple recurrent networks.

Another typical method is to reduce the numerical precision. Gupta et al. [17] demonstrates that DNN can be trained with the 16-bit fixed-point representation using stochastic rounding. Vanhoucke et al. [42] use 8-bit quantization for activations and weights to accelerate computation. Binary Neural Networks [12] introduces the

method to train neural networks with both binary weights and binary activations constrained to +1 or -1.

Multiple papers propose to train a small network (student model) to mimic the function of a large network. Ba et al. [3] train shallow networks on regressing logits which are logarithms of predicted probabilities produced by the large network. Hinton et al. [24] introduces the temperature of the softmax loss function to produce a softer probability distribution over classes for training student models.

DNN acceleration. Many hardware-based and software-based approaches have been proposed to accelerate DNN computation. For hardware-based methods, different ASIC designs [2, 5, 7, 8, 10, 14, 18, 25, 27, 33, 36, 37] and tools for accelerating DNN on existing platform have been proposed. Minerva [36] reduces the overall power consumption by aggressive data types optimization, selective operations pruning, and SRAM voltages reduction. Han et al. [18] propose EIE, a dedicated hardware accelerator which utilizes the sparsity in compressed DNNs. Caffe [26], MXNet [6] and cuDNN [9] are designed to accelerate DNNs on existing computation platform especially GPUs.

In addition, software-based methods accelerate DNN computation by introducing efficient computation strategies and algorithms. Batched lazy computation [42] is introduced to utilize the temporal locality and exploit the trade-off between execution latency and computation efficiency. Frame skipping [41] computes the DNN once and use the same output for consecutive multiple inputs. Vasilache et al. [43] accelerate DNN by faster convolutional computation into FFT computation, which suffers from extra memory cost. Lavin et al. [30] introduce a fast algorithm for only 3x3 filters using Winograd's minimal filtering algorithms.

7 CONCLUSION

In this paper, we propose *Scalpel* to customize DNN pruning for different hardware platforms based on their parallelism. It includes two techniques: *SIMD-aware weight pruning* and *node pruning*. For low-parallelism hardware, SIMD-aware weight pruning is applied to keep remaining weights in aligned groups to fully utilize the SIMD units. All groups have the same size equal to the SIMD width. For high-parallelism hardware, node pruning removes redundant nodes. It avoids the sparsity in weights matrices caused by traditional pruning techniques. SIMD-aware weight pruning and node pruning can be combined and applied to DNN models for moderate-parallelism hardware. On the microcontroller, CPU and GPU, Scalpel achieves mean speedups of 3.54x, 2.61x, and 1.25x while reducing the model sizes by 88%, 82%, and 53%.

8 ACKNOWLEDGEMENTS

This work is supported in part by ARM Ltd and by the National Science Foundation under grant XPS-1438996. The authors would like to thank fellow members of the CCCP research group, and the anonymous reviewers for their time, suggestions, and valuable feedback.

REFERENCES

[1] 2016. NIVDIA DIGITS DevBox. (2016). https://developer.nvidia.com/devbox.
[2] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: ineffectual-neuron-free deep neural network computing. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 1–13.
[3] Jimmy Ba and Rich Caruana. 2014. Do deep nets really need to be deep?. In *Advances in neural information processing systems*. 2654–2662.
[4] Guoguo Chen, Carolina Parada, and Georg Heigold. 2014. Small-footprint keyword spotting using deep neural networks. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 4087–4091.
[5] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, Vol. 49. ACM, 269–284.
[6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).
[7] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and others. 2014. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 609–622.
[8] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. (2016).
[9] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).
[10] Ping Chi, Shuangchen Li, Z Qi, P Gu, C Xu, T Zhang, J Zhao, Y Liu, Y Wang, and Y Xie. 2016. PRIME: A Novel Processing-In-Memory Architecture for Neural Network Computation in ReRAM-based Main Memory. In *Proceedings of ISCA*, Vol. 43.
[11] Ronan Collobert and Jason Weston. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*. ACM, 160–167.
[12] Matthieu Courbariaux and Yoshua Bengio. 2016. Binarynet: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).
[13] Misha Denil, Babak Shakibi, Laurent Dinh, Nando de Freitas, and others. 2013. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*. 2148–2156.
[14] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 92–104.
[15] Ross Girshick. 2015. Fast R-CNN. In *International Conference on Computer Vision (ICCV)*.
[16] Yiwen Guo, Anbang Yao, and Yurong Chen. 2016. Dynamic Network Surgery for Efficient DNNs. *arXiv preprint arXiv:1608.04493* (2016).
[17] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. (2015).
[18] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. *arXiv preprint arXiv:1602.01528* (2016).
[19] Song Han, Huizi Mao, and William J Dally. 2015. A deep neural network compression pipeline: Pruning, quantization, huffman encoding. *arXiv preprint arXiv:1510.00149* (2015).
[20] Song Han, Jeff Pool, John Tran, and William J Dally. 2015. Learning both Weights and Connections for Efficient Neural Networks. *arXiv preprint arXiv:1506.02626* (2015).
[21] Babak Hassibi, David G Stork, and Gregory J Wolff. 1993. Optimal brain surgeon and general network pruning. In *Neural Networks, 1993., IEEE International Conference on*. IEEE, 293–299.
[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385* (2015).
[23] Tianxing He, Yuchen Fan, Yanmin Qian, Tian Tan, and Kai Yu. 2014. Reshaping deep neural network for fast decoding by node-pruning. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 245–249.
[24] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).
[25] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation. (2016).
[26] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.
[27] Patrick Judd, Jorge Albericio, and Andreas Moshovos. 2016. Stripes: Bit-serial deep neural network computing. *IEEE Computer Architecture Letters* (2016).

[28] Alex Krizhevsky. 2012. cuda-convnet. (2012). https://code.google.com/p/cuda-convnet/.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[30] Andrew Lavin. 2015. Fast algorithms for convolutional neural networks. *arXiv preprint arXiv:1509.09308* (2015).

[31] Yann Le Cun, John S Denker, and Sara A Solla. 1989. Optimal brain damage.. In *NIPS*, Vol. 89.

[32] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[33] Robert LiKamWa, Yunhui Hou, Julian Gao, Mia Polansky, and Lin Zhong. 2016. RedEye: analog ConvNet image sensor architecture for continuous mobile vision. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 255–266.

[34] Min Lin, Qiang Chen, and Shuicheng Yan. 2013. Network in network. *arXiv preprint arXiv:1312.4400* (2013).

[35] Thomas Miconi. 2016. Neural networks with differentiable structure. *arXiv preprint arXiv:1606.06216* (2016).

[36] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 267–278.

[37] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *Proc. ISCA*.

[38] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[39] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.

[40] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1–9.

[41] Vincent Vanhoucke, Matthieu Devin, and Georg Heigold. 2013. Multiframe deep neural networks for acoustic modeling. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 7582–7585.

[42] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. 2011. Improving the speed of neural networks on CPUs. (2011).

[43] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2014. Fast convolutional nets with fbfft: A GPU performance evaluation. *arXiv preprint arXiv:1412.7580* (2014).