# A Comprehensive Analysis of Low-Impact Computations in Deep Learning Workloads

Hengyi Li
Zhichen Wang
Xuebin Yue
Dept. of Electronic and Computer Engineering, Ritsumeikan University
Kusatsu, Shiga, Japan

Wenwen Wang
Dept. of Computer Science
University of Georgia
Athens, GA, USA

Tomiyama Hiroyuki
Lin Meng
Dept. of Electronic and Computer Engineering, Ritsumeikan University
Kusatsu, Shiga, Japan

## ABSTRACT

Deep Neural Networks (DNNs) have achieved great successes in various machine learning tasks involving a wide range of domains. Though there are multiple hardware platforms available, such as GPUs, CPUs, FPGAs, and etc, CPUs are still preferred choices for machine learning applications, especially in low-power and resource-constrained computation environments such as embedded systems. However, the power and performance efficiency become critical issues in such computation environments when applying DNN techniques. An attractive optimization to DNNs is to remove redundant computations to enhance the execution efficiency. To this end, this paper conducts extensive experiments and analyses on popular state-of-the-art deep learning models. The experimental results include the numbers of instructions, branches, branch prediction misses, cache misses, and etc, during the execution of the models. Besides, we also investigate the performance and sparsity of each layer in the models. Based on the analysis results, this paper also proposes an instruction-level optimization, which achieves the performance improvement ranging from 10.26% to 28.0% for certain convolution layers.

## CCS CONCEPTS

• **Computer systems organization** → *Parallel architectures*; *Very long instruction word*.

## KEYWORDS

Deep Learning Neural Network, SIMD-CPU-Architecture, Comprehensive analysis, Instruction-level, Model optimization

**ACM Reference Format:**
Hengyi Li, Zhichen Wang, Xuebin Yue, Wenwen Wang, Tomiyama Hiroyuki, and Lin Meng. 2021. A Comprehensive Analysis of Low-Impact Computations in Deep Learning Workloads. In *GLSVLSI '21: ACM Great Lakes Symposium on VLSI, June 22–25, 2021, Virtual Event*. USA ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3453688.3461747

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*GLSVLSI '21, June 22–25, 2021, Virtual Event, USA*
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8393-6/21/06…$15.00
https://doi.org/10.1145/3453688.3461747

## 1 INTRODUCTION

Recently, Deep Neural Networks (DNNs) have achieved great successes with the availability of massive data and the computational support of high-performance computer hardware [1, 2]. Meanwhile, multiple state-of-the-art DNNs have been adopted in various domains, including computer vision, self-driving vehicles, cultural heritage preservation [3], environmental protection [4], and etc.

The computation of a DNN can be split into two main phases: the training phase and the inference phase. The training phase is the process of creating the neural network model by learning from the data, which configures the parameters of the network, such as weights of convolution layers, and then fine tuning the model by paring and optimizing the model. The inference phase is the deployment of the trained neural network models. GPUs have been proved to be great and powerful devices to efficiently run the computation of deep learning workloads. They play an indispensable role in improving the performance of DNN tasks because of their excellent features of parallel processing, which is exactly deep learning models need. On the other side, CPU is still an attractive platform for deep learning tasks, especially for the inference stage, due to the broad availability.

A previous research work has pointed out that a significant fraction of the future computation demands will come from the workloads corresponding to deep learning inference [5]. The demands of deep Learning inference are increased sharply as the performance of the models is improved significantly and their applications are popularized greatly. Meanwhile a large number of inference tasks are taking place on CPUs. Furthermore, CPUs are also continuing to evolve around semiconductor design, semiconductor manufacturing technology, and CNNs software frameworks.

The modern CPUs have developed several techniques to improve the performance and speed up the processing of data, e.g., thread-level parallelism and single-instruction-multiple-data (SIMD). One API Deep Neural Network Library (oneDNN), which is a high-performance library of basic building blocks for deep learning applications, has also been distributed to help developers improve the productivity and enhance the performance of deep learning applications on CPU platforms. Further, on x86-64 CPUs, oneDNN automatically detects the instruction set architecture (ISA) at runtime and uses just-in-time (JIT) code generation techniques to generate optimized code to exploit the latest features of the ISA. With the help of oneDNN, the CPU performance can be greatly enhanced. For embedded systems, such as mobile devices and robotics, the hardware resources are quite constrained while the

application efficiency is highly demanded. Neural networks that require high computational and memory resources are hard to be deployed in such environments. Example networks include VGGNet [6], ResNet [7], DensNet [8], etc. Thus, optimizing DNNs has been a major research topic. Although the operations of DNNs have been highly optimized with the support from various techniques like oneDNN, there is still room for further improving the performance. For example, a previous research work [9] reveals that many computations in DNNs are actually low impact, and the activation sparsity in DNNs have not been utilized sufficiently. In particular, CPU-based DNN applications have not been fully researched and CPU features have not been fully exploited for DNN workloads. In this paper, we make a systematic study on the inference process of DNNs using a set of representative models with the emphasis on the embedded CPU architecture. Our study includes:

- The number of instructions, branches, branch miss predictions, cache misses and so on for the inference process on CPUs with the SIMD feature.
- The layer-wise performance and sparsity analysis on CPU platforms.

The study discovers several interesting findings on the performance of the inference process of DNNs and provides insightful information about how to identify the performance bottleneck. According to the results of the study, we further propose a method for the code generation process to optimize the computations of DNNs. The technical contribution of the paper is providing a guideline for software-level optimizations of DNNs on embedded CPUs. Furthermore, it also presents some positive feedback on designing new hardware architectures using FPGA or heterogeneous computation units to accelerate the performance of DNNs.

The rest of the paper is organized as follows: Section 2 provides the background of the DNNs studied in the paper and introduces the experimental methodology. Section 3 analyzes the characteristics of CPUs with the SIMD feature. In Section 4 and Section 5, we analyze the layer-wise execution time and sparsity of DNNs in detail. Section 6 presents the experimental results of our proposed optimization, followed by the conclusion in Section 7.

## 2 PRELIMINARY

We analyze several state-of-the-art DNNs in this paper. They include VGGNet [6], InceptionNet [2, 10, 11], ResNet [7], DenseNet [8], ShuffleNet V2 [12, 13], MobileNet V2 [14], MnasNet [15].

VGGNet, as well as InceptionNet, demonstrates that the depth of CNNs with multiple stacked small size kernel filters is a vital factor for the performance of the network. In theory, it allows to learn more complex patterns by increasing the depth of the network with multiple non-linear layers. The Inception architecture has been proposed with multi-level feature extractors to optimize the computations when the network is deeper. The deep residual learning framework proposed in ResNet is mainly to address the degradation problem introduced by the deep network architecture. DenseNet strengthens the feature propagation mechanism by exploiting the potential of feature reuses and alleviates the problem of exploding gradients. The last three architectures are lightweight models that are designed for resource-constrained hardware platforms such as mobile devices. All of the networks are the state-of-the-art architectures and have been used widely in many applications.

There are several representative deep learning frameworks, such as TensorFLow, PyTorch, Keras, and etc. In our experiments, we use PyTorch (version 1.7.0a0) as the framework because it is one of the most popular frameworks. The experimental platform is Dell OptiPlex 700 equipped with an Intel i7-9700 CPU at 3.00GHz and 2×32GB DDR4 main memory. The operating system of our experimental platform is Linux 20.04.

We adopt the ImageNet ILSVRC 2012 data set for the experiments. The obtained performance results correspond the inference process for one input that are averaged from 1000 inputs. In section 6, the dataset CIFAR100 and Kuzushiji are added for a more thorough analysis. ImageNet and CIFAR100 are both public data sets, and Kuzushiji is a data set consisting of 64132 images that are classified into 1117 categories [16]. For these databases, the input size is unified to be RGB channels and cropped out the centered [224×224] from each image to keep the inputs consistent for all networks. And the inputs are pre-processed with the normalization of mean and standard deviation.

## 3 CHARACTERISTICS OF DNNS ON A CPU ARCHITECTURE WITH SIMD

In this section, we analyze the characteristics of the inference process of DNNs on a CPU architecture with SIMD. Table 1 shows the experimental results, which include Floating-Point Operations (FLOPs), the number of instructions (the num. of inst.), branches, mispredicted branches (branch misses), cache misses. Furthermore, the inference results indexes are also listed, such as the top-1 accuracy and inference time. Due to the fact each DNN architecture has multiple models with different layers and versions, the paper chooses one representative model of these DNN architectures: VGG19_BN, Inception V3, ResNet101, DensNet121, ShuffleNet V2_1.0, MobileNet V2, MnasNet1.0. The size of the networks and the number of the parameters are also listed in Table 1 because they are closely related to the performance of the models.

As DNNs differ greatly in exploiting inter-operation parallelism, we use one thread to run the inference process to achieve a fair comparison between different networks.

We list the findings discovered from Table 1 as follows:

**Model size and parameters:** The size and parameters of models are key factors of the models. This is because the models with smaller sizes and less parameters have higher computation and memory efficiency. Although VGG19_BN has much less layers, i.e., only 16 convolution layers, the model size is much larger than others, i.e., 548MB, which is about 18 times larger than DensNet121 that has 120 convolution layers. It can be seen that for some DNNs, like DensNet121, when the network architecture gets deeper with more convolution layers, the size and parameters become less. Since the three lightweight models are designed mainly for resource-constrained devices, all of them have the least model size, i.e., no more than 17MB, and parameters, i.e., no more than 5 millions, which are far less than other models.

**Inference time, FLOPs and num. of inst.:** The results of these three columns are closely correlated with each other. Ordinarily, the more instructions and FLOPs, the more inference time. For example,

**Table 1: The inference performance of different DNN models**

| DNNs | Inference time/ms | FLOPs num./G | Inst. num./mil. | Branches num./mil. | BPKI | Branch-misses num./mil. | Branch-misses Rto | MPKI | Cache-misses num./mil. | Top-1 accuracy | Model size/MB | Params /mil. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **VGG19_BN** | 398.47 | 19.68 | 4315.81 | 74.80 | 17.33 | 0.97 | 1.30% | 0.22 | 24.00 | 74.7% | 574 | 143.68 |
| **Inception V3** | 92.30 | 5.75 | 908.56 | 59.67 | 63.69 | 0.89 | 1.49% | 0.95 | 5.76 | 69.3% | 104 | 27.16 |
| **ResNet101** | 197.76 | 7.87 | 1816.44 | 35.51 | 19.94 | 0.57 | 1.62% | 0.32 | 14.29 | 75.5% | 170 | 44.55 |
| **DenseNet121** | 94.32 | 2.90 | 823.30 | 45.34 | 55.14 | 0.74 | 1.63% | 0.90 | 7.46 | 57.3% | 30.8 | 7.98 |
| **ShuffleNet V2_1.0** | 17.88 | 0.15 | 110.75 | 14.05 | 127.30 | 0.33 | 2.36% | 2.99 | 1.22 | 70.1% | 8.8 | 2.28 |
| **MobileNet V2** | 24.77 | 0.33 | 134.11 | 10.51 | 80.04 | 0.13 | 1.26% | 0.99 | 3.01 | 71.1% | 13.6 | 3.50 |
| **MnasNet10** | 20.50 | 0.34 | 122.61 | 5.84 | 47.87 | 0.12 | 2.05% | 2.22 | 2.41 | 73.2% | 16.9 | 4.38 |

VGG19_BN takes the longest inference time of 384.84 *ms* with the most number of instructions 4302.47 million, and 19.77 GFLOPs. Shufflenet v2_1.0 takes the shortest inference time of 9.57 *ms* with the least number of instructions of 55.34 million and 0.05 GFLOPs, and the size of Shufflenet v2_1.0 is also the smallest.

**Branch and branch miss:** For modern CPUs, the branch predictor is critical in performance and the number of branch misses is a vital performance factor for modern deeply pipelined microprocessor architectures [17]. The models with fewer branch instructions and branch misses may have a relatively better performance. In these models, VGG19_BN has the maximum number of branch misses. The lightweight architectures enjoy less branch instructions and branch misses. However, the branch miss rate of ShuffleNet v2_1.0 is several times higher than others: 1.04x - 2.19x. And MobileNet V2 has the lowest branch miss rate.

The branches per kilo instructions (BPKI) of lightweight neural networks are much higher than other models in general. ShuffleNets v2_1.0 with the least instructions has the most BPKI with the value of 127.30, which is about 7 times more than VGG19_BN.

The lightweight models have a larger number of the Missed Predictions Per Kilo Instructions (MPKI) with the value of no less than 2 except for MobileNet V2 with the value of 0.99.

**Cache miss:** CPU caches play critical roles in mitigating the performance gap between CPU and the main memory. A cache miss happens when the requested data is not found in the cache. Cache misses slow down the overall execution process. The result is closely related to the size of models, the number of parameters, and the amount of data processed. Therefore, the models with larger sizes and more parameters often have more cache misses. For example, VGG19_BN and ResNet101 have 22.00 million and 14.29 million cache misses, respectively, which are multiple times more than others. Generally, lightweight models with less model sizes and parameters perform well.

## 4 PERFORMANCE ANALYSIS OF LAYERS

A DNN typically consists of multiple layers that include an input layer, convolution layers, non-linear scalar operator layers, down-pooling layers, fully connected layers, and etc. This section conducts a layer-wise performance analysis on popular DNNs by profiling each CPU kernel implementation to gain insights about the performance of neural networks. Fig.1 shows the details of five DNN architectures.

The VGG architecture [6] analyzed in the paper is VGG19_BN, which has an optimization by adopting Batch-Normalization (BN) compared with VGG19. As Fig.1-VGG19_BN shows, the convolution operations can be divided into five stages from conv1_x to conv5_x according to the input and output. The time cost of each layer is consistent for the convolution layers with the same input shape and output shape, such as the last three convolution layers of the stage conv3_x and conv4_x, and the four convolutions layers of conv5_x. In summary, the convolution operations account for 80% of the execution time of the model.

Inception V3 [11] is selected to analyze the performance of Inception architectures. As Fig.1-Inception V3 shows, the layer-wise execution time shows a certain periodicity in the stacked "Inception module" architecture. Among the layers, the four convolution layers and the two max pool layers consume the most of the execution time, more than 2 *ms*.

Fig.1-ResNet121 represents the details of the ResNet architecture [7]. The convolution operations are divided into five stages based on the output size and channels. The only max pooling layer consumes the most time of nearly 5 *ms*, and the convolution computation takes up to nearly 90% of the execution time of the inference process. And the time consumption for ResNet architectures changes periodically, which aligns well with the residual block of $[1 \times 1]$ - $[3 \times 3]$ - $[1 \times 1]$ sequence convolution. Convolution layers with the kernel size of $[1 \times 1]$ takes about the half time of the convolution layer with the kernel size of $[3 \times 3]$ within the block. The time cost of other layers is subtle and can be negligible.

As Fig.1-DensNet121 shows, the convolution operations can be simply divided into four dense blocks (DB) [8]. In terms of convolution layers: the fourth bar in the histogram represents the time consumption of the max-pooling layer with a high value, which is similar to ResNet. The four groups of bars with linear increasing refer to the pointwise convolution layers for their input channels increased linearly. And the four groups of the bars with equal value reflect the convolution layers with $[3 \times 3]$ kernels. The regularity of the figure reflects the architecture to certain extent.

ShuffleNet [13] utilizes the pointwise group convolution and channel shuffle to greatly reduce the computation cost while maintaining the accuracy. The architecture uses pointwise group convolutions to reduce the computation complexity of the costly dense $[1 \times 1]$ convolutions, and utilizes the channel shuffle operation to overcome the side effects brought by group convolutions. The convolution operations can be divided into four stages as shown in Fig.1-ShuffleNet V2. The max pool layer of the fourth layer takes the most time of 2.2 *ms*. The first and the last convolution layer consume more time compared with other convolution layers. The performance of convolution layers within the three stages shown in the figure represents an obvious pattern corresponding to the ShuffleNet block.

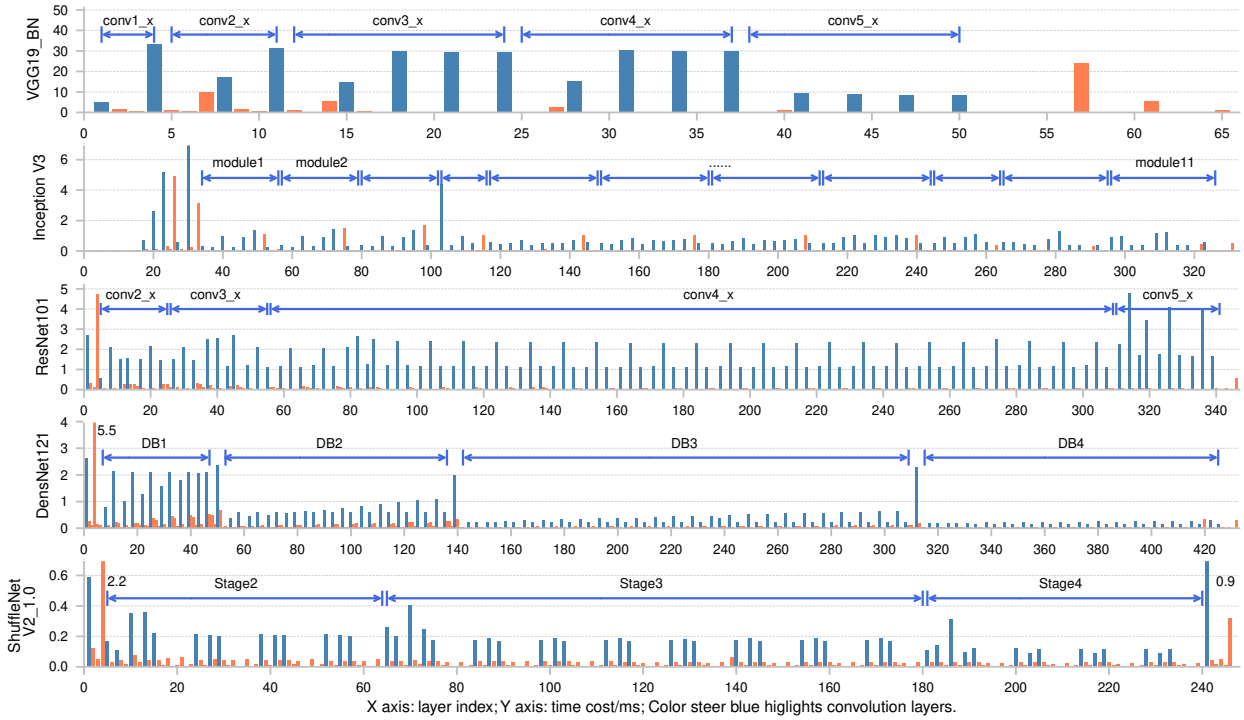


**Figure 1: Layer-wise performance analysis**

The data shows that although DNNs consist of multiple parts, it is generally the convolution operations that dominant the execution time of the inference process. The time cost of other layers, such as Max Pool, ReLU, Batch Normalization, etc, is negligible.

## 5 LAYER-WISE SPARSITY

The computation intensity is a heavy burden for hardware with constrained resources. Reducing the intensity of computation has been an important research topic for hardware platforms with constrained resources, e.g., embedded devices. The sparsity of the computation, which is defined as the fraction of zeros in the parameters and input activation matrices in this paper, has emerged as a quite effective solution.

The sparsity can be classified into two categories [18]: static sparsity and dynamic sparsity. The static sparsity means the sparsity in the parameters of the DNN models (i.e., weight sparsity), while the dynamic sparsity is introduced when the value of a neuron element turns into 0 and does not contribute anything to follow-up neurons, as shown in the following equation. In this paper, we focus on the analysis of the layer-wise dynamic sparsity of the inputs of the convolution layers.

$$DS(n) = \frac{\sum_{i=0}^{N}\sum_{j=0}^{H}\sum_{k=0}^{W} f(O(i,j,k) == 0)}{N \times H \times W} \quad (1)$$

where DS(n) denotes the dynamic sparsity of the input of the n-th convolution layer; N, H, and W denote the number of channels, the height, and the width of the feature maps, respectively; O(i,j,k) denotes the element of feature maps.

Fig.2 shows the layer-wise sparsity of the five DNN models with pre-trained parameters in PyTorch. There is no sparsity in the raw input of the models, so that the percentage of the first convolution layer for each model is zero.

Fig.2-VGG19_BN shows the layer-wise sparsity of VGG19_BN. The sparsity of some convolution layers, including the third, the fifth, the ninth and the thirteenth layers, is significantly reduced because of the max pool operation before the layers. The sparsity of other layers is as high as 80%.

For the "Inception Block" stacked architectures, as Fig.2-Inception V3 shows, the twelfth, nineteenth, twenty-sixth, fortieth, fiftieth, sixtieth, seventieth, eighty-sixth layers that have an obvious sparsity decrease correspond to the last layer of each inception blocks. The sparsity keeps rising as a whole with layer gets deeper and reach a very high percentage even up to 90% for some layers.

Fig.2-ResNet101 illustrates that the sparsity pattern of the ResNet architecture coincides with the residual block of $[1 \times 1]$ - $[3 \times 3]$ - $[1 \times 1]$ sequence convolution, just like the layer-time performance. Except for individual layers especially most of the $[3 \times 3]$ convolution layers, the sparsity is as high as 75%.

Fig.2-DensNet121 reflects the "bottleneck" architecture of the DensNet: the layer structure of BN-ReLU-Conv$[1 \times 1]$-BN-ReLU-Conv$[3 \times 3]$, and the dense block with stacked "bottleneck" as Fig.1-DensNet121 shows. The sparsity of the input feature maps corresponding to the $[1 \times 1]$ convolution layers have a lower value compared with the adjacent $[3 \times 3]$ convolution layers. However, the sparsity of the last several $[3 \times 3]$ convolution layers is still quite high, i.e., up to 95%.

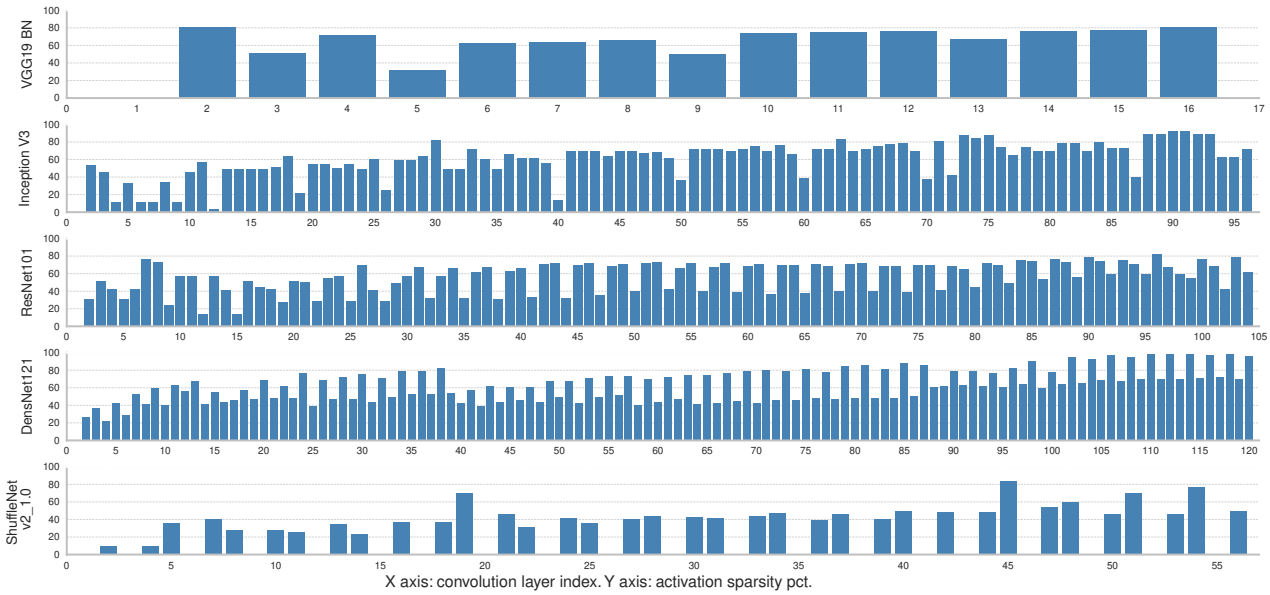


**Figure 2: The sparsity of the layer-wise feature maps**

As a lightweight architecture designed for resource-constrained platforms, the ShuffleNet architecture has been optimized in terms of over-parameterization. Fig.2-ShuffleNet V2_1.0 also shows that the architecture has much less sparsity compared with other models investigated in the paper. Since the architecture does not have a ReLU layer for each convolution layer, so the input sparsity of many convolution layers is zero.

Through the above analysis, we can conclude that the activation sparsity of DNNs is very high, which offers plenty of room for optimizations. And there have been several algorithms based on this [18]. However, DNNs running on CPUs with SIMD extensions have not been fully explored yet.

## 6 AN INSTRUCTION-LEVEL OPTIMIZATION

It can be concluded from the above sections that the convolution computation which dominates the DNN models has a large number of low-impact computations, due to the sparsity. CPUs can achieve the peak performance through SIMD extensions. With the support of SIMD instructions, a single instruction can perform the same operation simultaneously on multiple data. This is achieved by loading multiple values into SIMD registers (e.g. *ymm* registers) and processing them simultaneously. And the SIMD resources have been utilized by MKL-DNN to optimize the convolution operations. In this paper, we propose an approach at the SIMD instruction level to optimize the convolution computations.

The idea of the approach is to eliminate the convolution computation that has a very high sparsity, which may be up to 80%. The performance improvement of the approach is evaluated using the VGG_BN architectures and the computing supported by MKL-DNN is taken as the baseline. The basic computing unit of convolution operations in VGG_BN is shown below. The operation process can be simply divided into four stages. *S3* and *S4* are iterated four times:

*S1*: Using the "*vbroadcastss m32, %ymm1*" instruction to broadcast three feature-map elements (the three elements corresponding to the $[3 \times 3]$ kernel) in the memory address to the eight locations in the *ymm* register.

*S2*: Checking the values of the three elements using the "*vpor*" and "*vptest*" instructions. If the three elements are all zero, then skip to the end of the computing unit and go to the next iteration starting from *S1*.

*S3*: Using the "*vmovups %ymm2/m256, %ymm1*" to move 8 weights (the eight values corresponding to 8 filters) in the memory address to the eight locations in the *ymm* register.

*S4*: With the "*vfmadd231ps %ymm3/m256, %ymm2, %ymm1*" instruction to multiply the eight packed single-precision floating-point values in the first source operand and the second source operand, adding the intermediate result to the third source operand, performing the rounding operation, and storing the result values to the third source operand, i.e., the destination operand.

The difference between the optimization and the benchmark happens at the adding stage *S2*. We use the pre-trained parameters in PyTorch for ImageNet. For CIFAR100 and Kuzshiji we train the models again. The top-1 accuracy is listed in Table 2.

**Table 2: Models inference process performance**

| dataset | ImageNet | | | CIFAR100 | | | Kuzushiji | | |
|---|---|---|---|---|---|---|---|---|---|
| model | A | B | C | A | B | C | A | B | C |
| Top-1 accuracy | 71.5% | 73.6% | 74.7% | 61.0% | 62.7% | 61.4% | 85.1% | 86.4% | 87.4% |

A: VGG13_BN. B: VGG16_BN. C: VGG19_BN.

Fig.3 shows the result of *S2* applied to the second convolution layer of VGG13_BN, VGG16_BN, VGG19_BN with three datasets, using the DNNs supported by MKL-DNN. The conclusion is described as follows:

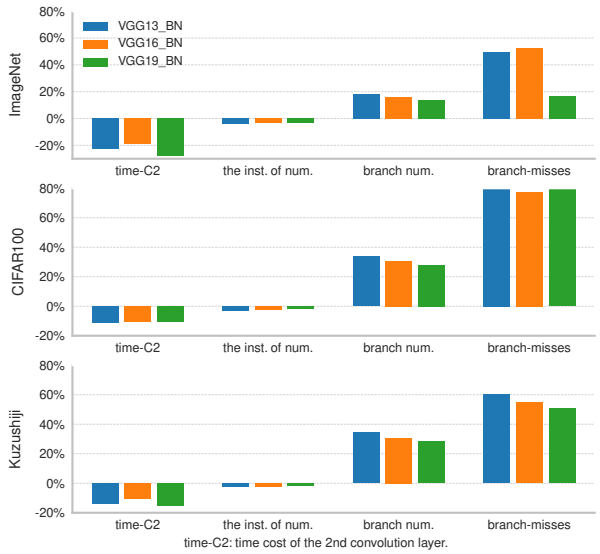


**Figure 3: Optimization on the second convolution layer**

- The time cost of the second convolution layer is decreased from 10.26% to 28.02% compared with the original benchmark. VGG19_BN on ImageNet has the highest decrease and on CIFAR100 has the lowest decrease.
- The overall number of instructions is decreased from 1.38% to 4.19%. VGG13_BN on ImageNet has the highest decrease and VGG19_BN on CIFAR100 has the lowest decrease.
- The total branches of the model is increased from 14.12% to 34.57%. VGG13_BN on Kuzushiji has the highest increase and VGG19_BN on ImageNet has the lowest increase.
- The miss rate of branches is increased from 16.76% to 82.96%. VGG13_BN on CIFAR100 has the highest increase and VGG19_BN on ImageNet has the lowest increase.

The proposed approach achieves performance improvement in the second convolution layer of VGG_BN. Although the inputs of other convolutions layers have high sparsity as well, the approach does not attain performance improvement.

Pipelining is a key technique for modern CPUs to improve their efficiency by executing multiple overlapped instructions. However, the branch predictor is on the critical path of pipelining and branch misses are a significant impediment for modern deeply-pipelined microprocessor architectures. The penalty for a branch miss can be much higher than the pipeline refill time [19]. Although *S2* eliminates low-impact computations and thus decreases the overall instructions, it also brings in much more branch instructions and mispredicted branches. As a result, the pipelining processes is broken by branch miss predictions, which leads to inefficient execution on CPUs.

We investigate the characteristics of the second and other convolution layers of the architecture. Despite the high sparsity percentage of the input feature maps they share, the input size of the second convolution layer is $[224 \times 224]$, which is the same as the first convolution layer. However, the size of input feature maps decreases as the layer gets deeper from the third convolution layer to the last one. The feature map size of the last convolution layer is only $[14 \times 14]$. And the sparsity of the larger sized input feature maps is more concentrated compared with the small sizes. Therefore, the branch misses of the second convolution layer, caused by *S2*, are much less than other convolution layers. As a result, the proposed approach works for the second convolution layer but not other layers in VGG_BN.

## 7 CONCLUSIONS

This paper conducts an analysis on the hardware characteristics and layer-wise performance of representative DNNs on CPUs with SIMD extensions. We find that the convolution operations, which dominate the inference process of DNNs, has a very high sparsity, which leads to lots of low impact computations. We also propose an approach at the instruction level to optimize the convolution operations. The proposal achieves the performance improvement from 10.26% to 28.0% for certain convolution layers in the inference process. Although it reduces the number of instructions, it also brings lots of branch instructions and mispredicted branches. This points out an interesting research direction for the future design of DNN accelerators. For example, we can design a dedicated branch predictor for DNNs. The paper provides a guideline for optimizing DNNs on CPUs with SIMD extensions, as well as the potential hardware solutions based on FPGAs and heterogeneous accelerators.

## REFERENCES


[1] Alex Krizhevsky et al. Imagenet classification with deep convolutional neural networks. volume 25. Curran Associates, Inc., 2012.
[2] Christian Szegedy et al. Going deeper with convolutions. *CVPR2015*, pages 1–9, 2015.
[3] Lin Meng et al. Oracle bone inscription detector based on ssd. *ICIAP2019*, pages 126–136, 2019.
[4] Lin Meng et al. Underwater-drone with panoramic camera for automatic fish recognition based on deep learning. *Access*, pages 17880–17886, 2018.
[5] Jongsoo Park et al. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv:1811.09886*, 2018.
[6] Karen Simonyan et al. Very deep convolutional networks for large-scale image recognition. *ICLR2015*, 2015.
[7] Kaiming He et al. Deep residual learning for image recognition. *CVPR2016*, pages 770–778, 2016.
[8] Gao Huang et al. Densely connected convolutional networks. *CVPR2017*, pages 2261–2269, 2017.
[9] Masahiko Atsumi et al. A comprehensive analysis of low-impact computations in deep learning applications. *FIT2020*, 2020.
[10] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *The 32nd Int.Conf.*, 2015.
[11] Christian Szegedy et al. Rethinking the inception architecture for computer vision. *CVPR2016*, pages 2818–2826, 2016.
[12] Xiangyu Zhang et al. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *CVPR2018*, pages 6848–6856, 2018.
[13] Ningning Ma et al. Shufflenet V2: practical guidelines for efficient CNN architecture design. In *ECCV2018*, volume 11218, pages 122–138, 2018.
[14] Andrew G. Howard et al. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, 2017.
[15] Mingxing Tan et al. Mnasnet: Platform-aware neural architecture search for mobile. In *CVPR2019*, pages 2820–2828, 2019.
[16] Center for Open Data in the Humanities:. Kuzushiji dataset (in japanese). http://codh.rois.ac.jp/char-shape/book/100249416/, 2019. (Apr. 22, 2021 accessed).
[17] Lin Meng et al. A novel branch predictor using local history for miss-prediction. *CDES2012*, pages 77–83, 2012.
[18] Xuda Zhou et al. Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In *Proc. 51st IEEE/ACM Int. Symp. Microarchitecture*, pages 15–28, 2018.
[19] Stijin Eyerman et al. Characterizing the branch misprediction penalty. In *ISPASS2006*, pages 48–58, 2006.