2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)

DLFusion: An Auto-Tuning Compiler for Layer Fusion on Deep Neural Network Accelerator

Zihan Liu, Jingwen Leng*, Quan Chen, Chao Li, Wenli Zheng, Li Li, Minyi Guo*
Emerging Parallel Computing Center, Department of Computer Science and Engineering
Shanghai Jiao Tong University
Shanghai, China
Email: {altair.liu, leng-jw}@sjtu.edu.cn, { chen-quan, lichao, zheng-wl, lilijp, guo-my } @cs.sjtu.edu.cn

***Abstract*—Many hardware vendors have introduced specialized deep neural networks (DNN) accelerators owing to their superior performance and efficiency. As such, how to generate and optimize the code for the hardware accelerator becomes an important yet less explored problem. In this paper, we perform the compiler-stage optimization study using a novel and representative Cambricon DNN accelerator and demonstrate that the code optimization knobs play an important role in unleashing the potential of hardware computational horsepower. However, even only two studied code optimization knobs, namely the number of cores and layer fusion scheme, present an enormous search space that prevents the naive brute-force search. This work introduces a joint, auto-tuning optimization framework to address this challenge. We first use a set of synthesized DNN layers to study the interplay between the hardware performance and layer characteristics. Based on the insights, we extract the operation count and feature map channel size as each layer's characteristics and derive a joint optimization strategy to decide the performance-optimal core number and fusion scheme. We evaluate the performance of the proposed approach using a set of representative DNN models and show that it achieves the minimal of 3.6x and the maximal of 7.9x performance speedup compared to no optimization baseline. We also show that the achieved speedup is close to the oracle case that is based on a reduced brute-force search but with much less search time.**

***Keywords*-Auto-Tuning;Layer Fusion;Hardware Accelerator;**

I. Introduction

Deep learning has achieved great success in the key application domains such as computer vision and natural-language processing. The derived deep neural network (DNN) models have significant requirement for computation and memory resources, which exceed the capability of the existing architectures. Owing to the repeated common computation pattern in different DNN models, such as 2D convolution and matrix multiplication, both the academia and industry begin to embrace the specialized hardware accelerators [14, 11, 18] for their high performance and energy efficiency. Compared to the general-purpose architecture like CPU/GPU, hardware accelerators are specialized for a specific task and have simplified control logic so that they can dedicate more resources for computation and memory structure [14]. Their design decision has also lead to their distinctive programming models from the CPU/GPU.

General-purpose architectures have well-defined ISAs so that the compiler can perform various performance optimizations. However, optimizing the code for the hardware accelerators is challenging because hardware vendors do not expose the accelerator's low-level ISA. Instead, hardware vendors provide a SDK with high-level API for application/model developers, but the SDK is highly abstracted and difficult to control the exact hardware behavior. As such, significant efforts are devoted in computational-graph level optimization such as operator fusion, operator concatenation [3], and operator substitutions [10]. Nonetheless, the graph-level optimization are independent of underlying hardware, cannot be used to tune the performance for a specific accelerator.

In this paper, we explore the code optimization for a novel accelerator Cambricon MLU100 [27]. MLU100 has a higher peak performance in FP16/INT8 than Tesla V100 [18], but also requires highly optimized code to fully unleash its computational horsepower. The accelerator SDK includes both high-level and low-level APIs. The high-level APIs are highly abstracted and have little optimization space. In contrast, the low-level API exposes two execution hyper-parameters, which are number of cores and layer fusion schemes. Our characterization results show that we need to carefully and wisely select those two parameters to achieve the optimal performance for a specific DNN model. However, even with only two parameters, the search space is too large for a brute-force search.

On the basis of the low-level SDK, we propose an compiler-stage auto-tuning optimizer, *DLFusion*, for MLU100 accelerator which performs joint optimization for the two exposed execution hyper-parameters. To the best of our knowledge, this work is the first to consider arbitrary auto-fusion patterns that are mathematically equivalent. In contrast, existing fusion optimizations such as TensorRT and XLA [17, 16] are rule-based and therefore can only support a limited set of pre-defined fusion patterns. The

* Jingwen Leng and Minyi Guo are corresponding authors of this paper

978-1-6654-1485-2/20/$31.00 ©2020 IEEE
DOI 10.1109/ISPA-BDCloud-SocialCom-SustainCom51426.2020.00041

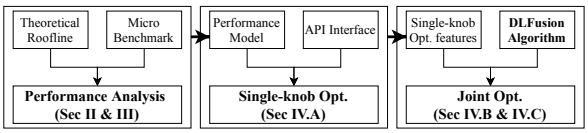

Figure 1. Overall flow of the optimizer design in our work.

basic architecture of the optimizer is shown in Figure 1, and this paper is organized as followed. In Section II, we first show the performance gap between theoretical model and actual execution using a series of micro-benchmark, then we analyze the optimization knobs and performance in detailed in Section III, by which we present our performance model for later optimization. Moreover, we find it difficult to achieve the optimal settings for different networks by btute methods, so an auto-tunning optimization is necessary. In Section IV.A, we first propose single knob optimization for Model Parallelism using the aforementioned performance model, then we integrate fusion scheme into the optimization procedure guided by the insight of Section IV.B, finally, we present our optimization algorithm and software implementation in Section IV.C.

II. Investigated Platform

In this paper, we use Cambricon MLU-100-C3 [27], a dedicated deep learning accelerator designed by Cambricon Technologies. The MLU-100 can deliver the computing performance with much better energy efficiency than general-purpose processors like CPU and GPU. In this section, we first give a brief introduction of the accelerator and then conduct an analysis to reveal the factors that can impact the accelerator's performance efficiency.

A. Experimental Setup

1) Hardware Setup: Table I lists the detailed specifications of the studied Cambricon MLU100 accelerator. The accelerator has 32 cores in total, where each core has the computation power 1 TFLOPS in FP32, 2 TFLOPS in FP16, and 4 TFLOPS in INT8. The accelerator can support common computation patterns in deep neural network models, including the convolution layer, ReLU layer, and batch normalization layer [27]. However, its core microarchitecture is not revealed, which we use a microbenchmark-based methodology to study. With those auto-generated microbenchmarks covering different computational intensity and operation count, we can quickly have a high-level understanding of the target hardware's computational characteristics. One of the salient features of our work is that, for other accelerators with different microarchitectures, this microbenchmark methodology can also be applied to reveal hardware characteristics for optimization.

2) Software Setup: The chip vendor provides an operator-level SDK and a high-level runtime library for programming the accelerator. The operator-level SDK, called CNML,

Table I
The MLU100 hardware specification.

Item	Descriptions
Core freq.	1GHz
Float perf. (FP16)	64 TFLOPS
Integer perf. (INT8)	128 TOPS
Memory size	8 GB
Memory bandwidth	102.4 GB/s
Memory bit width	256-bit
Host Interface	PCIe 3.0x16
TDP	110 W
ECC Enabled	Yes

```
/******************** Model Parallelism *********************/
cnmlBaseOp_t op;
// Configuring parameters, allocating memory, ...
cnmlCreateOperator(&op, ... /* operator specification */);
cnmlCompileOperator(&op, Model_Parallelism);
output = cnmlComputeOperatorForward(&op, input);
/********************** Layer Fusion ************************/
cnmlBaseOp_t op_1, op_2;
cnmlFusionOp_t fusion_op;
// Configuring parameters, allocating memory, ...
cnmlCreateOperator(&op_1, ... /* operator 1 specification */);
cnmlCreateOperator(&op_2, ... /* operator 2 specification */);
cnmlFuseOperator(&op_1, &fusion_op);
cnmlFuseOperator(&op_2, &fusion_op);
cnmlCompileFusionOperator(&fusion_op, Model_Parallelism);
output = cnmlComputeFusionOperatorForward(&fusion_op, input);
```

Figure 2. SDK code sample for setting MP and layer fusion.

supports common operators such as convolution, ReLU, and BatchNorm, which are used in computer vision and natural language processing models. The CNML supports two hyper-parameters for running those operators, as shown in Figure 2. The first hyper-parameters is *model parallelism* (*MP*), which specifies the number of cores (up to 32) used by the operator. The second hyper-parameter is *layer fusion*, which specifies the number of layers that are fused for concurrent execution and therefore increased parallelism. These two hyper-parameters can represent the execution of fusion operation on multi-core architecture based accelerators, and this optimization is orthogonal to other graph-level optimizations including Common Subexpression Elimination (CSE) [16], operator substitution [10], etc. In this work, we use this operator-level SDK and tune the hyper-parameter settings to optimize different DNN models.

B. Performance Analysis

We first construct a single-layer based microbenchmark to study the accelerator's performance efficiency for DNN model layers with different characteristics. We focus on the convolutional layer (`Conv`) and fully connected layer (`FC`) because they represent most of the computation in today's DNN models [29]. For each layer, we sweep its different parameters and compute their required operation count

through Equation 1 and 2 respectively. We first perform the experiment using the single core and then multiple cores to understand the impact of the core number on performance efficiency.

$$GOPS_{Conv} \leftarrow 2{\times}H_{Out}{*}W_{Out}{*}H_K{*}W_K{*}C_{In}{*}C_{Out} \quad (1)$$

$$GOPS_{FC} \leftarrow 2 \times M \times K \times N \quad (2)$$

1) Single-core Performance: We first use the simple roofline model [31] to model the performance of convolution and fully-connect layers under different parameters, and the operation intensity is calculated as equation 3.

$$Intensity \leftarrow \frac{GOPS}{\sum(sizeof(tensors))} \quad (3)$$

However, as shown in Figure 3, there's significant gap

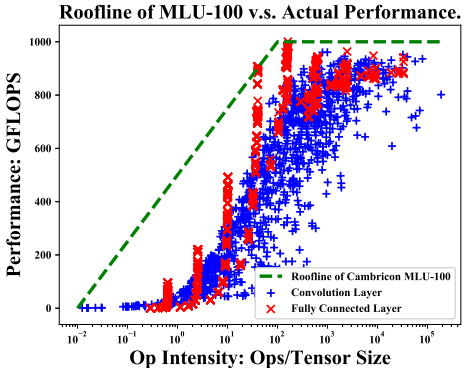

Figure 3. Roofline model of Cambricon MLU-100 and actual performance.

between the exact performance and theoritical performance on Cambricon MLU-100, moreover, operation intensity in roofline model can not distinct the performance of operations under the same intensity effectively leading to the difficulty in modeling the performance.

So, we applied `PCA` method to extract the parameters that are most likely to influence the performance (by which we can adjust the hardware resource assigned to the operation and thus improve the performance), and for Cambricon MLU-100, we found that operation count has the most significant influence on the performance, and channel the second.

Figure 4(a) shows the relationship between the layer's operation count and achieved performance efficiency measured by GFLOPS (Giga floating-point operations per second), and the operations with similar op count have similar performance (the error bar stands for the standard diviation in the figure is short). As Figure 4 shows, layers performance efficiency is largely determined by its operation count: the higher the operation count, the better performance efficiency on the accelerator, and once the operation count reaches a critical value, the performance will not increase. We called it $OpCount_{critical}$ and will be used in our optimization process.

On the other hand, layers with medium and high intensity exhibit a larger variability of performance efficiency for the layers with the same operation count, according to the PCA result, channel should be the main reason. To verify the assumption, we observe the performance of convolution layers with different channel/kernel size/output image size with other parameters fixed. As shown in Figure 4(b), We found that channel have non-negligible influence on layer's performance. Actually, the errorbar in Figure 4(a) is mainly introduced by changing channel. For kernel size and feature size, they contribute little to distinct the performance of different layers, which match the result of PCA well. As such, we also explore the setting of the parameters with first and second largest influence according to PCA result: channel of convolution.

2) Multi-core Performance: The above single-core performance experiment shows that the operation count of a layer impacts the accelerator performance efficiency. Based on this observation, we further study the impact of the number of cores with varying operation count. For this experiment, we start with a fixed convolutional layer from the VGG-19 model [26]: `Input/Output Channel=64`, `Output Size=`224×224, `Kernel Size=`3×3, for which we use the notation of $\{64, 64, 224 \times 224, 3 \times 3\}$ to represent its parameters in the following sections. We increase the operation count of the layer via expanding the `Channel` dimension by different factors. Figure 4(c) shows that the layers with large operation count prefer a large number of cores. Layers with small (moderate) operation count prefer a small (moderate) core numbers to achieve the best performance.

III. Performance Analysis of Optimization Knobs

After analyzing the performance features of the DNN accelerator, we focus on the problem of achieving the best performance for a give DNN model, i.e., lowest inference latency. This section motivates the need for an auto-tuning compiler by demonstrating that the hyper-parameter setting for achieving the best performance for a given DNN model is highly dependent on model characteristics.

In this work, we focus on the aforementioned two hyper-parameters, namely `model parallelism` and `layer fusion` for optimizing the performance of a DNN model on the MLU-100 accelerator. For the convenience of the experiment, we first study the two hyper-parameters separately.

A. Model Parallelism.

As we describe in Section II, `model parallelism` (MP) represents the number of cores for running the given DNN layer. We sweep this MP hyper-parameter for different DNN models and show the results in Figure 5(a).

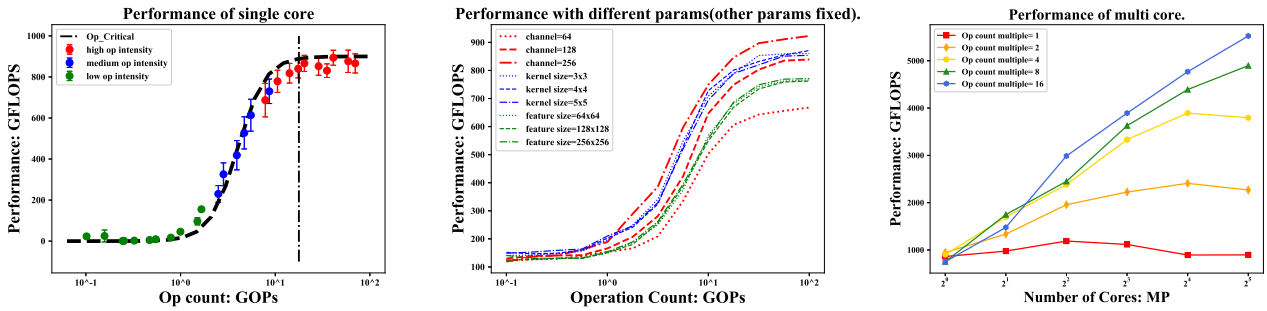

Figure 4. (a) Single core performance of ops. (b) Influence convolution parameter with other parameters fixed. (c) Multi core performance of ops.

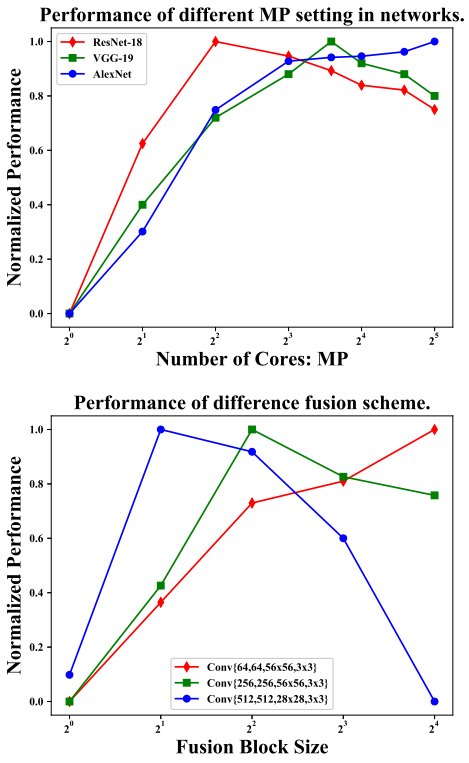

Figure 5. (a) Optimal MP setting of different networks with all layers sharing the same MP. (b) Optimal fusion block size of convolutions with different parameters.

The results show that **using the maximum number of cores does not necessarily lead to the best performance**. The optimal core number for ResNet-18 and VGG-19 is 4 and 16, respectively. The reason is that when the MP is too large, each core is dispatched with less number of operation count, leading to net performance degradation. As such, it is essential to find a method to set the optimal MP based on the characteristic of different DNN models.

B. Layer Fusion.

This layer fusion hyper-parameter lets users combine multiple layers into a single block, which has two benefits. It first increases the concurrent operation count than the layer-wise, no-fusion execution. Consider an example of fusing two layers: the computation of the second layer can start when the first layer's output is partially available. It also reduces the cost of off-chip memory round trip because the output of a layer can be generated on-chip and immediately reused.

Prior work like TVM [3] only considers the fusion of convolutional layers and other types of layers such as ReLu and batch normalization. However, the vendor-provided CNML programming frameworks supports the fusion of almost arbitrary types and numbers of layers. One of the major differences between our work and TVM is that we consider multiple convolution layers in a fusion block. As a result, the optimization space is much larger and therefore more challenging.

To study its impact on the performance, we construct three CNN models, each of which has 16 identical baseline `Conv` layers. The three baseline layers, which are selected from ResNet [8] and VGG [26], have parameters of $\{64, 64, 56 \times 56, 3 \times 3\}$, $\{256, 256, 56 \times 56, 3 \times 3\}$, and $\{512, 512, 28 \times 28, 3 \times 3\}$, respectively.

In this experiment, we sweep the fusion block size B_{size} for executing each CNN model, which leads to $16/B_{size}$ fused blocks. As Figure 5 shows, different models have different optimal fusion block sizes. The reason is that although layer fusion has two major benefits, it also involves redundant computation that needs a careful trade-off. If the fusion block size is too large, the redundant calculation will even degrade the overall performance. As such, we must also find a strategy to decide the optimal fusion scheme for different CNN models.

C. Infeasibility of Brute-Force Search.

For the optimal performance, we must jointly consider the fusion scheme and model parallelism, which leads to a huge space that makes the brute-force search infeasible.

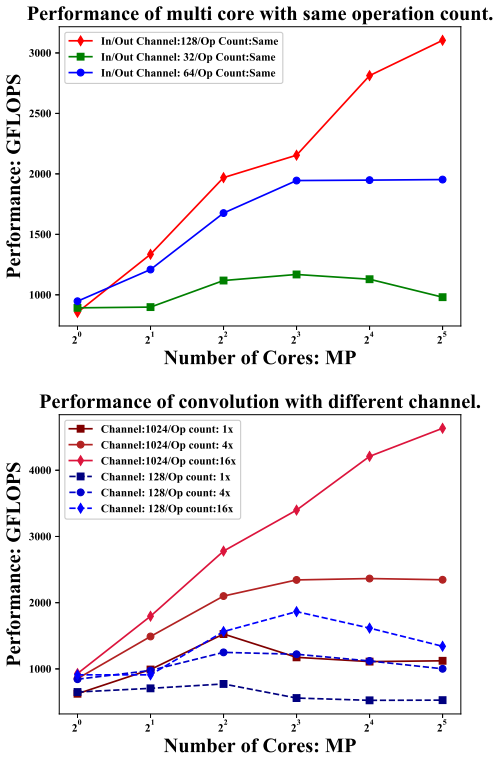

Figure 6. (a) Multi-core performance fixing operation count. (b) Multi-core performance fixing channel.

Assume a CNN model with n layers, Equation 4 gives the number of possible combinations for setting the two hyper-parameters. When n equals 50, there are 8.17×10^{75} possible combinations. Based on our experimental insights, we propose an intelligent auto-tuning optimizer that uses the inherent characteristics of CNN models to quickly identify their hyper-parameter setting.

$$Space(n) = \sum_{i=1}^{n-1}(32^{i+1} \times \frac{\prod_{x=1}^{i}(n-x)}{i!}) \quad (4)$$

IV. DLFusion Approach

In this section, we present the design and implementation of *DLFusion*, which jointly optimizes the two knobs, *model parallelism, MP* and *layer fusion*, to maximize the inference performance of DNN models on the target MLU-100 accelerator. We first explain how to select the optimal `MP` for a given DNN layer, and then explain how to determine the layer fusion scheme for multiple layers based on their characteristics. In the end, we formalize our optimization algorithm and present its implementation details.

A. Single Layer MP

In Section II, we show that the operation count of the `Conv` layer has a significant impact on its optimal `MP`. However, operation count along can not be used for deciding the layer's optimal `MP`. In this subsection, we propose a joint model that uses both the operation count and in/out channel size of convolutional layers to determine the optimal `MP` configuration for the `Conv` layer.

We conduct a more detailed `MP` impact analysis for three different layers ($\{32, 32, 224 \times 224, 3 \times 3\}, \{64, 64, 112 \times 112, 3 \times 3\}, \{128, 128, 56 \times 56, 3 \times 3\}$). In short, those three layers have the same operation count but different input/output channels. Figure 6(a) shows that those three layers have different optimal `MP` values. This is because the hardware partitions the tensor on channel dimension with a certain minimal partition size. In addition, we also find the operation count itself impacts the optimal `MP` values. As Figure 6(b) shows, `Conv` layers with the same input/output channel but different operation count have different optimal `MP` values. `Conv` layers with fewer channels with high operation count could prefer more cores than layers with more channels but less operation count. Given those findings, we use both a layer's channel size (C in the formula) and operation count to determine its optimal core number, as shown in Equation 5, where α, β are hardware-tuned scaling factors. We emperically decide the value of α and β for MLU100 is 0.316 and 0.659 respectively according to the weight result of PCA.

$$MP(C, OpCount) \propto \alpha \times \log_2(C) + \beta \times \log_2(OpCount) \quad (5)$$

B. Multiple Layers Fusion and MP

As presented in Section II, layer fusion can reduce the data movement and increase the operation count dispatched to cores to improve the performance. The fusion block composed of multiple layers can also leverage multiple cores to further reduce the latency. On the other hand, using more mores for the fusion block also leads to redundant computation owing to the halo effect of 2D-convolution illustrated in Figure 7(a). Moreover, the redundant computation increases when the number of layers in the fusion block grows, which means fusing more layers does not necessarily improve the performance. To efficiently enable fusion into our optimization procedure, we first present our insight using several identical layers to illustrate the factors that influence the performance of fusion blocks. Then, given the significant layer heterogeneities in actual neural networks, we present our DLFusion algorithm for joint optimization considering both MP and fusion schemes.

1) Identical Layers: We use two different `Conv` layers and compare their performance when fusing 4 and 16 layers. Figure 7(b) shows the performance comparison where `Conv1` is $\{512, 512, 28 \times 28, 3 \times 3\}$ and `Conv2` is $\{512, 512, 14 \times 14, 3 \times 3\}$, they have 1.72 GOPs and 0.43 GOPs respectively. While fusing more layers for `Conv2` leads to better performance, the situation is opposite for `Conv1`. The main reason is that when a large fusion

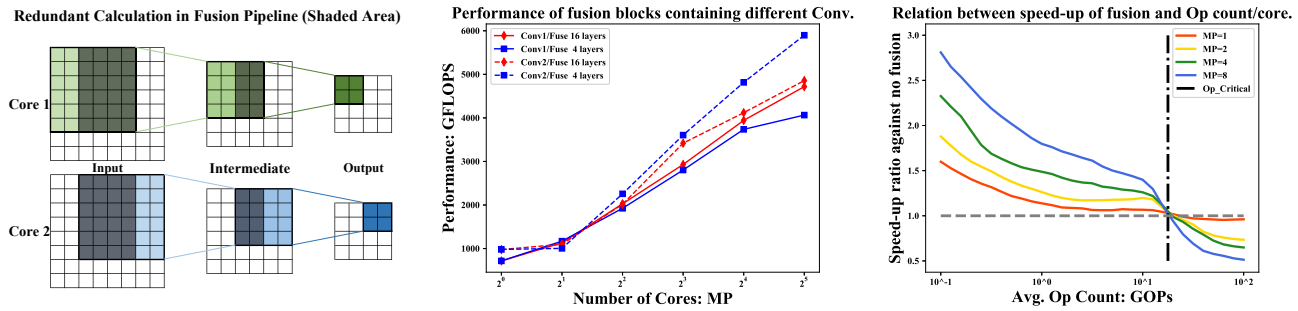

Figure 7. (a) Redundant computation in layer fusion [1]. (b) Fusing different CONV layers. (c) Relation between speed-up ratio and the cores used.

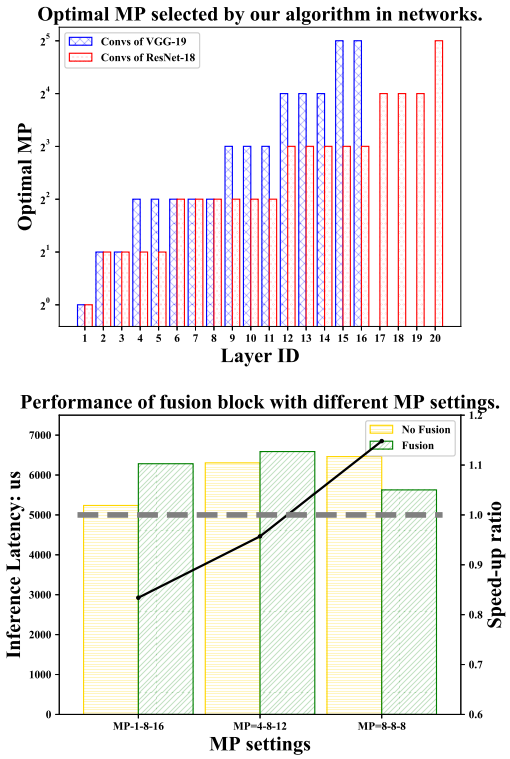

Figure 8. (a) Optimal MP selected by our method in ResNet-18 [8] and VGG-19 [26]. (b) Performance and the speed-up ratio of fusion block containing convolutions with different optimal MP. Generally, we should determine the optimal MP given the convolution layer parameters. However, in this experiment, since we want to observe the influence of different optimal MP of convolutions in a fusion block, we determine the MP first and than determine the convolution parameters according to selected MP.

block uses more cores, it leads to the more redundant computation. As shown in Figure 7(c), before reaching the critical operation count, using fusion can deliver better performance than not using fusion, since the single-core performance increase rapidly before the critical point according to Figure 3(b). Once exceeding the critical point, the performance drops significantly due to the redundant computation (and that's why the single-core performance is stable before and after the critical point because using a single core will not introduce redundant computation). It should be noted that when using more cores, the critical value is slightly smaller also because of the redundant computation account for more op count. Moreover, before reaching the critical value, using more cores leads to better performance since more parallelism can be leveraged. For single-core, the improvement gained by fusion is mainly from reduced memory round-trip. So, for layer fusion, we should limit the size of fusion block close to but below critical operation count of the cores to benefit the most from parallelism and avoid unacceptable redundant computation.

2) Non-identical Layers: In the actual DNN models, we find that the layers have different parameters that lead to different optimal MP as shown in Figure 8(a). When fusing layers with significant different optimal MPs into one block, severe underperformance is observed as Figure 8(b). The reason is that all layers in one fusion block will share the same MP, which is only suitable for a small part of layers in the block. Given this finding, we choose to determine optimal MP of every single layer first to avoid this underperformance for later joint optimization with fusion (since we can gather layers with the same or similar optimal MP together). Once the optimal MPs are determined, to strike a balance between the increased operation count and redundant computation in layer fusion, we use the following heuristics. When performing layer fusion, we gradually fuse layers until the operation count of fused layers is greater than a preset threshold, which provides enough parallelism for the hardware and bounds the amount of redundant computation.

C. Implementation

In this subsection, we first present the core of our work: the DLFusion optimization algorithm, and then we introduce how we implement and evaluate our algorithm on actual hardware.

1) DLFusion Algorithm: The DLFusion optimization algorithm aims to find a schedule with optimal MPs and fusion schemes for the hardware. We use the pseudo-code shown in Algorithm 1 for the joint optimization of fusion scheme and optimal MP. Our algorithm requires the input

Algorithm 1 Finding fusion scheme and hyper-parameter setting

Input: onnx_file, num_of_layers, $OpCount_{critical}$
Output: $fusion\_partition\_index[]$, $mp\_of\_fusionblock[]$

1: **function** JOINTOPTFUSIONANDMP(onnx_file, num_of_layer, $OpCount_{critical}$)
2: $layers\_spec \leftarrow [], sum\_Op \leftarrow 0$
3: $current\_mp \leftarrow 0, avg\_mp \leftarrow 0, block\_size \leftarrow 0$
4: **for** $i = 0 \rightarrow num\_of\_layer$ **do**
5: $layers\_spec[i] \leftarrow$ Specification of i^{th} layer interpreted by TVM.Relay
6: **if** $layers\_spec[i].type = Convolution/Fully - Connected$ **then**
7: $current\_mp \leftarrow$ selection based on channel(major) and Op count(minor)
8: $sum\_Op \leftarrow sum\_Op+$ operation count of i^{th} layer
9: $avg\_mp \leftarrow avg\_mp + current\_mp, block\_size \leftarrow block\_size + 1$
10: **end if**
11: $avg\_mp \leftarrow \frac{avg\_mp}{block\_size}$
12: **if** $\frac{sum\_Op}{avg\_mp} \geq OpCount_{critical}$ **then**
13: $fusion\_partition\_index.push(i)$
14: $mp\_of\_fusionblock.push(2^{\lfloor \log_2(avg\_mp) \rfloor})$
15: $sum\_Op \leftarrow 0, avg\_mp \leftarrow 0, block\_size \leftarrow 0$
16: **end if**
17: **end for**
18: **return** $fusion\_partition\_index, mp\_of\_fusionblock$
19: **end function**

of ONNX-based neural network description files, number of layers, and $OpCount_{critical}$, which is a tunable parameter that represents the operation count required by a single core to reach its peak performance. For MLU100, we choose this parameter as $10^{1.25}$GOPs as suggested in Figure 3(b) and Figure 7(c). The interpreter first reads the network parameters (Line 5). The algorithm then decides the optimal `MP` for each `CONV` layer based on its channel dimension and operation count (Line 8). The algorithm adds the current layer into the fusion block. It calculates the current total operation count and average `MP` for all the current fused layers (Line 8 to 11). If the operation count dispatched to each core exceeds the critical operation count $OpCount_{critical}$, we stop the fusion for the current block and start a new fusion block (Line 12 to 13). For the newly formed fusion block, we decide its `MP` as the closed to average `MP` and round it to 2^n (Line 14 to 15). This process repeats until all layers are processed.

2) DLFusion Compiler Tool Chain: To evaluate the aforementioned optimization algorithm, we design and implement a compiler tool-chain for Cambricon MLU-100. Figure 9 shows the details of our framework containing code generator and optimizer, which takes the input of ONNX format based neural network description file and generates the C++ code that leverages the MLU100's operator-level SDK CNML. The core in the framework is the optimizer, which is a specialized instance of Figure 1 and includes an optimization pass for tuning the execution parameters according to the DNN model characteristics.

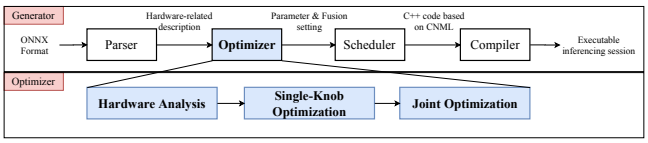

Figure 9. Overall architecture of our framework.

The code generator produces the C++ source code based on a template file to call the CNML library. The produced source code can be compiled to the executable inference session via the *g++* compiler. In our work, we choose the DNN format ONNX [19] because it is independent of specific deep learning frameworks. We use the TVM [3] as the parser to convert the ONNX-based network description format to the TVM's internal graph representation that the following scheduler and optimizer use.

V. EVALUATION

In this section, we evaluate DLFusion using a set of representative CNN models, including ResNet, VGG, AlexNet and mobileNet, as listed in Table II. We focus on the inference and use the frame per second (FPS) as the performance metric. To demonstrate the effectiveness of DLFusion, we compare the performance of different optimization strategies, including a reduced brute-force search strategy.

1) Evaluated Strategy: We evaluate different optimization strategies that are listed in Table III. Strategy 1 referring to no fusion and no model parallelism ($MP = 1$) is used as the baseline. Both strategy 2 and 3 perform no fusion while the

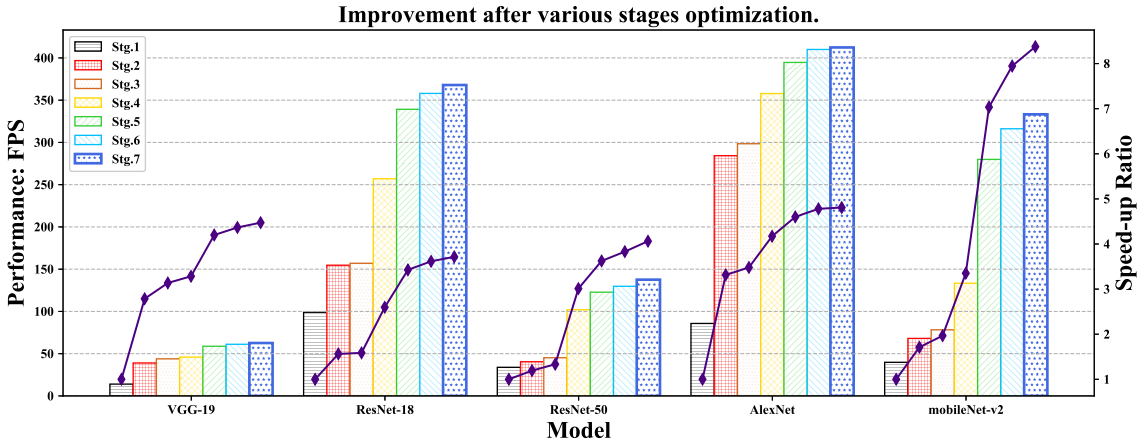

Figure 10. Performance comparison of different optimization strategies.

Table II
NETWORKS DESCRIPTION.

Network	Total Op	Avg. Op	No. of CONV
ResNet-18/50 [8]	3.38/7.61	0.169/0.144	20/53
VGG-19 [26]	36.34	2.27	16
AlexNet [12]	1.22	0.244	5
mobileNet [25]	10.33	0.199	52

Table III
DIFFERENT OPTIMIZATION STRATEGIES.

No.	Strategy Name	Description
1	Non-Optimization	No fusion with $MP = 1$
2	Fixed MP	No fusion, all the layers have the same MP.
3	Dynamic MP	No fusion, each layer its own MP.
4	All Fusion & Max. MP	All layers fused into one block, MP is set to be maximal
5	Fusion & Fixed MP	All layer fused to multiple blocks (Alg. 1) with the same MP.
6	**DLFusion** (Fusion & Dynamic MP)	Layer fused by Alg. 1 set MP for each fused block.
7	Brute-force Search	Optimal performance.

former uses the same MP for all layers, and the latter uses the layer-specific MP. Strategy 4 simply fuses all layers and uses the maximal MP for the fused block. Strategy 5 uses Alg. 1 to fuse layers to multiple blocks and use a single fixed MP for all blocks. In contrast, Strategy 6 uses Alg. 1 to fuse layers and set a block-specific MP value for each block. Strategy 7 represents the optimal performance through a brute-force search that we detail later.

2) Performance Comparison: Figure 10 shows the performance comparison of different optimization strategies. Except for the oracle case (the last bar), DLFusion has the best performance, which achieves a speedup of 3.6 - 7.9× against the baseline. Our algorithm leads to significant performance improvement for the two following reasons. First, each fusion block has a proper operation count that gains plenty of parallelism while with acceptable redundant computation. Second, the number of cores used in each fusion block is also close to their optimal number of cores that balances computation and memory access. The studied CNN models have different performance trend over the different optimization strategies, for which we make the following observations.

- CNN models with low operation count per layer (e.g., ResNet and mobileNet) are not sensitive to MP optimization because using more cores leads to less utilization of each core. In contrast, the model with high operation count per layers (e.g., VGG-19) benefits more from MP optimization.
- CNN models with low operation count per layer (e.g., ResNet and mobileNet) benefits significantly from the layer fusion optimization because layer fusion produces a block with more operations, which results in performance improvement. In contrast, the model with high operation count per layers (e.g., VGG-19) benefits less from layer fusion.
- With the increasing of the number of layers (from 4 CONV layers in AlexNet to over 50 CONV layers in mobileNet and ResNet-50), the model gets more sensitive to the fusion strategy because of larger fusion scheme space.

3) Oracle Case: To evaluate the effectiveness of the DLFusion approach, we design a feasible brute-force search as the oracle case. As we explain in Section III, the hyper-parameter space is too large for the brute-force search. We reduce the search space based on the performance characterization analysis on the existing CNN models. First, we limit the choice of MP from $1, 2, 3...32$ to $1, 2, 4, 8, 12, 16, 24, 32$. Second, we limit the size of a fusion block to the multiple of four. These two rules lead to acceptable search time. The last column of each CNN model in Figure 10 represents

the oracle case achieved by our reduced brute-force search. The performance between the DLFusion and the oracle case is less than 10%. Meanwhile, with the increased number of layers, this performance gap gets smaller. In general, DLFusion achieves the performance that is close to the oracle case, with much reduced search time ($O(n)$, where n is the number of layers).

VI. Related Work

To address the difficulty in compiler tool-chain design and optimization, researchers have proposed Domain Specific Language (DSL) to schedule the hardware more efficiently. Halide is a popular DSL for image processing pipelines [24], and Taichi is proposed for CG processing [9]. DSLs are concise by omitting control logic in the regular programming language, which makes it more convenient for optimization. Given the similarity between image processing pipeline and DNN models, Halide-based compilation frameworks such as TensorComprehension [29] and TVM [3] have been proposed. Those frameworks target a general optimization at the computation graph level. In contrast, DLFusion is a hardware-specific optimization framework that can be integrated as a backend for the graph-level frameworks.

Previous researchers have studied various general optimizations, such as loop fusion [22] and kernel fusion [30, 21]. In the TensorComprehension [29] framework, fusion is performed with the use of the polyhedral model [2]. TASO [10] explores the graph substitution optimization using a cost-based backtracking search. Grappler [28] of TensorFlow conducts a series of rule-based arithmetic transformation, operation fusion. On the other hand, instead of optimize the execution on single hardware, Google REGAL [20] focus on the problem of scheduling the execution on multiple hardware. Those optimizations work at the graph-level and can be used for any hardware back-end.

Other possible optimization options for compilers include batching [4], model sparsity [32, 6], and data movement reduction between specialized hardware accelerators and general purpose processors [7]. Besides performance optimization, increasing efforts have been put on the robustness of the DNN systems including Ptolemy, an architecture that detect adversarial samples at inference time [5] based on critical path method [23]. Other researchers have also explored traditional reliability on heterogeneous systems [13]. To generalize the compiling stage optimization to these architectures is also an urgent need. DLFusion targets the layer fusion optimization that is specific to DNN models on a specific hardware accelerator, researchers have also explored the programming framework of this accelerator that support the optimization options with less code effort [15]. Prior work explores the layer fusion as an architecture optimization [1] while we use it for compiler optimization.

VII. Conclusion

In this work, we propose an end-to-end code generator with optimizer for the DNN accelerator Cambricon-MLU100, which is capable of generating optimized C++ code for a DNN model with ONNX format. We propose an auto-tuning algorithm to jointly optimize the two execution hyper-parameter (i.e., number of cores and layer fusion scheme) to maximize the accelerator performance for a given DNN model. The algorithm uses the operation count and channel size as the features to decide the optimal core count for each layer. It then gradually fuses layers into a block that has just enough computation to fully utilize the hardware and avoids excessive redundant computation. Evaluation shows that our approach achieves almost the same performance of the reduced brute-force search base oracle case, but with a much less search time. To the best of our knowledge, our work represents the first auto-tuning algorithm for a DNN accelerator can we hope it can foster more research efforts in this direction.

Acknowledgement

We thank the anonymous reviewers for their constructive feedback for improving the work. This work was supported by Major Scientific Research Project of Zhejiang Lab (No.2019DB0ZX01) and the National Natural Science Foundation of China (NSFC) grant (61702328, 61832006, and 61972247). Any opinions, findings, and conclusion in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] Manoj Alwani et al. “Fused-layer CNN accelerators”. In: *49th Annual IEEE/ACM International Symposium on Microarchitecture*. 2016.

[2] Uday Bondhugula et al. “A practical automatic polyhedral parallelizer and locality optimizer”. In: *Proceedings of the Conference on Programming Language Design and Implementation*. 2008.

[3] Tianqi Chen et al. “TVM: An Automated End-to-End Optimizing Compiler for Deep Learning”. In: *13th USENIX Symposium on Operating Systems Design and Implementation*. 2018.

[4] W. Cui et al. “Ebird: Elastic Batch for Improving Responsiveness and Throughput of Deep Learning Services”. In: *2019 IEEE 37th International Conference on Computer Design (ICCD)*. 2019, pp. 497–505. DOI: 10.1109/ICCD46524.2019.00075.

[5] Yiming Gan et al. “Ptolemy: Architecture Support for Robust Deep Learning”. In: *CoRR* abs/2008.09954 (2020).

[6] Cong Guo et al. “Accelerating Sparse DNN Models without Hardware-Support via Tile-Wise Sparsity”. In: *CoRR* abs/2008.13006 (2020).

[7] Cong Guo et al. “Balancing Efficiency and Flexibility for DNN Acceleration via Temporal GPU-Systolic Array Integration”. In: *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*. IEEE, 2020, pp. 1–6.

[8] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition*. 2016.

[9] Yuanming Hu et al. “Taichi: a language for high-performance computation on spatially sparse data structures”. In: *ACM Trans. Graph.* 38.6 (2019), 201:1–201:16.

[10] Zhihao Jia et al. “TASO: optimizing deep learning computation with automatic generation of graph substitutions”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019.

[11] Norman P. Jouppi et al. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017.

[12] A. Krizhevsky et al. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Adv. in Neural Info. Proc. Systems*. 2012.

[13] Jingwen Leng et al. “Asymmetric Resilience: Exploiting Task-Level Idempotency for Transient Error Recovery in Accelerator-Based Systems”. In: *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*. IEEE, 2020, pp. 44–57.

[14] Dao-Fu Liu et al. “PuDianNao: A Polyvalent Machine Learning Accelerator”. In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 2015.

[15] Zihan Liu et al. “Survey and design of paleozoic: a high-performance compiler tool chain for deep learning inference accelerator”. In: *CCF Trans. of High Performance Computing* (2020).

[16] Ruben Mayer, Christian Mayer, and Larissa Laich. “The TensorFlow Partitioning and Scheduling Problem: It’s the Critical Path!” In: abs/1711.01912 (2017).

[17] NVIDIA. *NVIDIA TensorRT: Programmable Inference Accelerator*. 2020.

[18] NVIDIA. *Tesla V100 Performance Guide*. 2018.

[19] ONNX. *Open Neural Network Exchange. The open standard for machine learning interoperability*. http://onnx.ai. Accessed Jun. 29, 2020.

[20] Aditya Paliwal et al. “Reinforced Genetic Algorithm Learning for Optimizing Computation Graphs”. In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. 2020.

[21] Bo Qiao et al. “Automatic Kernel Fusion for Image Processing DSLs”. In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*. 2018.

[22] Bo Qiao et al. “From Loop Fusion to Kernel Fusion: A Domain-Specific Approach to Locality Optimization”. In: *IEEE/ACM International Symposium on Code Generation and Optimization*. 2019.

[23] Yuxian Qiu et al. “Adversarial Defense Through Network Profiling Based Path Extraction”. In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 2019, pp. 4777–4786.

[24] J. Ragan-Kelley et al. “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines”. In: *Conference on Programming Language Design and Implementation*. 2013.

[25] M. Sandler et al. “MobileNetV2: Inverted Residuals and Linear Bottlenecks”. In: *Conference on Computer Vision and Pattern Recognition*. 2018.

[26] Karen Simonyan and Andrew Zisserman. “VVGGery Deep Convolutional Networks for Large-Scale Image Recognition”. In: *3rd International Conference on Learning Representations*. 2015.

[27] Cambricon Technologies. *Cambricon MLU100 Datasheet*. Aug. 2019.

[28] TensorFlow. *TensorFlow graph optimization with Grappler*. https : / / www . tensorflow . org / guide / graph_optimization. Access Jun. 29, 2020.

[29] N. Vasilache et al. “Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions”. In: *CoRR* 1802.04730 (2018).

[30] Guibin Wang, Yisong Lin, and Wei Yi. “Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU”. In: *2010 IEEE/ACM Int’l Conference on Green Computing and Communications*. 2010.

[31] Samuel Williams, Andrew Waterman, and David A. Patterson. “Roofline: an insightful visual performance model for multicore architectures”. In: *Commun. ACM* 52.4 (2009), pp. 65–76. DOI: 10.1145/1498765.1498785.

[32] Xuda Zhou et al. “Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach”. In: *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. IEEE Computer Society, 2018, pp. 15–28.