IOS: Inter-Operator Scheduler for CNN Acceleration

by

Yaoyao Ding

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science

Department of Electrical & Computer Engineering
University of Toronto

IOS: Inter-Operator Scheduler for CNN Acceleration

Yaoyao Ding
Master of Applied Science
Department of Electrical & Computer Engineering
University of Toronto
2022

Abstract

To accelerate CNN inference, existing deep learning frameworks focus on optimizing *intra-operator* parallelization. However, a single operator can no longer fully utilize the available parallelism given the rapid advances in high-performance hardware, resulting in a large gap between the peak performance and the real performance. This performance gap is more severe under smaller batch sizes. In this work, we extensively study the parallelism *between* operators and propose Inter-Operator Scheduler (IOS) to automatically schedule multiple operators' parallel execution through a novel dynamic programming algorithm. IOS consistently outperforms state-of-the-art libraries (e.g., TensorRT) by 1.1 to 1.5× on modern CNN benchmarks.

Acknowledgements

I want to thank Xiaodan (Serina) Tan for constructive discussion about NVIDIA GPU-related problems. I also want to thank Professor Song Han and Zhihao Jia for their guidance on this project. Finally, I want to thank my supervisor, Professor Gennady Pekhimenko, for not only the guidance of this project, but also staying with me as I take the pleasant journey as a master student.

Contents

Chapter 1

Introduction

Convolutional neural networks (CNNs) have achieved state-of-the-art performance across many tasks, including computer vision [26, 17], machine translation [46, 12], and game playing [33, 43]. The success comes at the cost of growing computational requirements. The high demand for computation makes efficient inference more critical in real deployment [16, 6, 22].

A common practice to improve inference efficiency is parallelization. Deep learning frameworks such as Tensorflow [1] and Pytorch [40] exploit *intra-operator parallelism*, which parallelizes arithmetic operations within a *single* CNN operator (e.g., convolution). However, due to the rapid

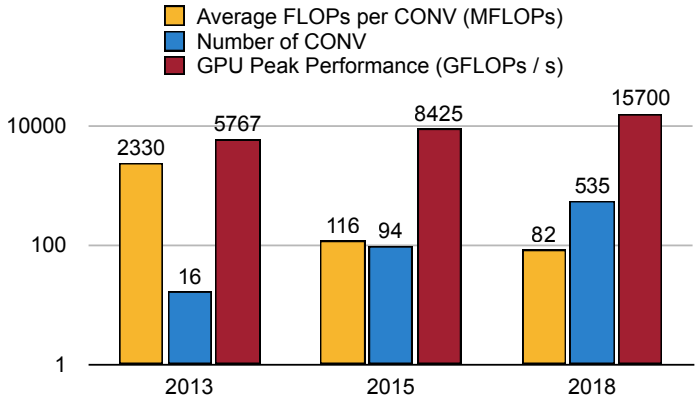


Figure 1.1: The trends of average computation per convolution, number of convolutions in a CNN and hardware peak performance. Device peek performance increases while average computation per convolution decreases, leading to a larger utilization gap. VGGNet and GTX 980Ti, Inception V3, and GTX 1080, NASNet and Tesla V100 are chosen as representatives for 2013, 2015, and 2018 respectively. All FLOPs are measured for single precision.

advances in high-performance hardware, intra-operator parallelism is no longer sufficient to obtain efficient resource utilization. As shown in Figure 1.1, the peak FP32 performance of a GPU has increased from 5.8 TFLOPs/s in 2013 to 15.7 TFLOPs/s in 2018 (shown in red). NVIDIA Tesla A100 even reaches a peak FP32 performance of 19.5 TFLOPs/s.

Meanwhile, there is a recent trend in CNN design to replace a single branch of convolutions with multiple branches of convolutions, which is advantageous due to increased model capacity under a fixed computation budget [48, 57, 52]. As a result, the number of convolutions grows while

the computation FLOPs in each convolution becomes smaller. For example, the average floating-point operations (FLOPs) per convolution has decreased from 2330 MFLOPs/kernel in VGG to 82 MFLOPs/kernel in NASNet. This exacerbates the device's under-utilization problem.

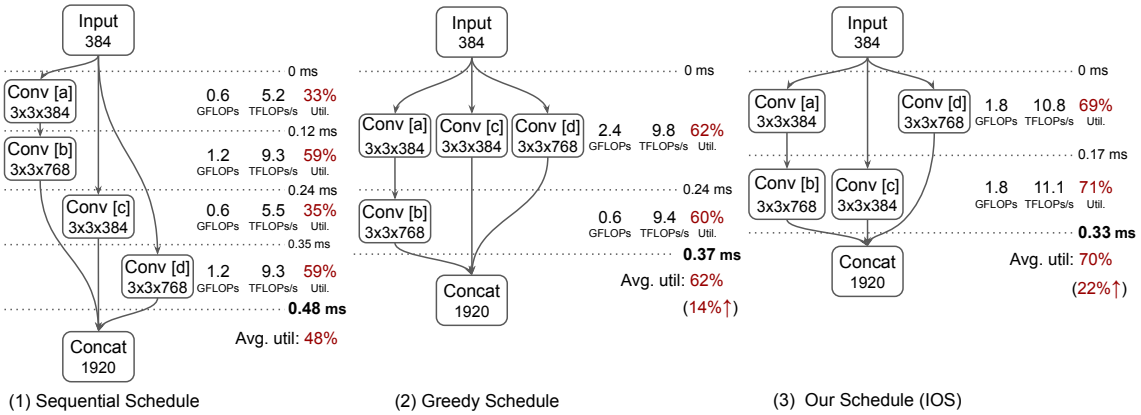


Figure 1.2: Different execution schedules for a computation graph on NVIDIA Tesla V100 GPU. Operators scheduled to run in parallel are placed at the same level between two dotted lines called a *stage*. Computation (GFLOPs), performance (TFLOPs/s), and hardware utilization (%) for each stage are profiled on the right. Both sequential and greedy schedules result in low resource utilization (48%-62%) and high latency (0.37-0.48ms). Our schedule yields higher utilization (70%) and lower latency (0.33ms).

To address this problem, recent work explores *inter-operator parallelism* by executing multiple CNN operators in parallel guided by different heuristics [50, 21, 30]. For example, MetaFlow [21] fuses multiple operators matching a specific pattern into a larger operator to increase operator granularity. Tang et al. [50] proposes a *greedy* strategy that directly executes all available CNN operators on CPU to maximize resource utilization. These approaches apply different heuristics to optimize local parallelization across a few CNN operators; however, such techniques do not lead to a *globally optimal* schedule for the entire CNN architecture. For example, given an input CNN (Figure 1.2 (1)), a greedy schedule (Figure 1.2 (2)) would perform convolutions [a], [c], and [d] in parallel, and run convolution [b] in a subsequent stage upon the completion of the previous stage.

This greedy schedule is sub-optimal for two reasons. First, a greedy schedule eagerly puts more operators in the early stages (as soon as they are available for execution) and fewer operators in subsequent stages, resulting in low utilization in later stages. Second, executing too many operators on the device concurrently may lead to resource contention problem that hurts the performance. For example, as shown in Figure 1.2, the greedy schedule (2) suffers from resource contention problem in the first stage and low-utilization problem in the second stage, comparing to our proposed schedule (3).

Obtaining an optimized schedule to parallelize a CNN model is a challenging task. On the one hand, the number of schedules grows exponentially with the number of operators, making it infeasible to evaluate all possible schedules exhaustively. For example, a network with 33 operators can have 9.2×10^{22} number of feasible schedules. On the other hand, an optimal schedule also depends on hardware specifications and inference settings (e.g., batch size). A high-end GPU (e.g., Tesla V100) can efficiently execute a schedule with many operators in parallel, while a low-end GPU (e.g., Tesla K80) might suffer from resource contention using the same schedule. A large batch size

naturally offers more intra-operator parallelism, while a small batch size has a stronger need for inter-operator parallelization. Therefore, given a diverse set of CNN architectures, hardware, and inference settings, it is hard to devise an efficient schedule manually for all scenarios.

To address this challenge, we propose IOS, an inter-operator scheduler that accelerates CNN inference by combining intra- and inter-operator parallelism. We observe that different schedules share common sub-schedules; thus, IOS adopts a dynamic programming technique to explore the schedule space and finds a highly optimized schedule under low search cost. We evaluate IOS on modern CNN models, including Inception-V3 [48], RandWire [52], NasNet-A [57], and SqueezeNet [19]. IOS consistently outperforms the sequential schedule and greedy schedule. IOS achieves 1.1 to 1.5× inference speedup compared to existing deep learning libraries (e.g., TensorRT). Furthermore, IOS demonstrates the necessity of *customizing the scheduling policy* for different hardware and inference configurations. IOS can achieve up to 1.15× inference speedup by customizing the scheduling recipe compared to itself with no customization.

Our contributions are summarized as follows:

- We point out a major bottleneck for efficient CNN inference: existing intra-operator parallelism cannot saturate modern hardware's high parallelism, especially for recent multi-branch CNN models. Inter-operator parallelism is crucial.
- We propose a novel dynamic programming algorithm to find a highly optimized schedule for inter-operator parallelization. This technique is platform-agnostic and can serve as a general technique for popular frameworks such as TensorFlow [31] and TVM [6].
- We apply IOS to various hardware and inference settings and show that the different configurations require different schedules. We can automatically customize the scheduling policy for different hardware and inference configurations. The specialized schedules consistently outperform existing deep learning libraries with 1.1 to 1.5× measured speedup in inference.

Chapter 2

Background

2.1 GPU Basics

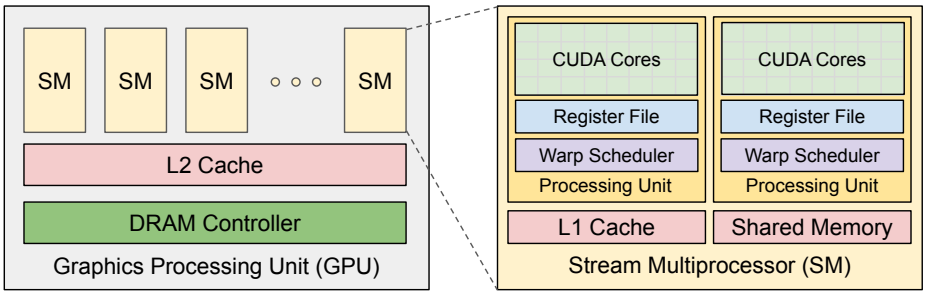


Figure 2.1: An overall architecture of modern GPUs. Each GPU contains tens of stream multiprocessors (SMs). All SMs share the L2 cache. Each SM contains several (e.g., 2 or 4) processing units. All these processing units share the L1 cache and shared memory. Each processing unit contains a warp scheduler, register file, and cores for arithmetic computation.

The Graphics Processing Unit (GPU) provides higher instruction and memory throughput than CPU, and is more suitable for workloads that can be decomposed into multiple independent parallel sub-workloads.

To define the workload, we write a piece of code called *kernel* that will be executed N times by N different threads, where N is specified when we launch the kernel. Once a kernel is launched, we have a *grid*[1] of threads to be executed on the GPU. The threads in a grid are partitioned into *thread blocks*. The threads in a thread block can communicate through block-scoped *shared memory*, and are able to synchronize. The threads in a thread block are further partitioned into *warps* of threads. For NVIDIA GPU, a warp usually contains 32 threads. Threads in a warp always execute a common instruction.

Figure 2.1 shows an overall architecture of modern GPU. A GPU contains tens of *stream multiprocessors* (SMs). When we launch a kernel, the GPU distributes the thread blocks of the derived grid onto the stream multiprocessors. Thread block will be executed on a multiprocessor and each multiprocessor may contain multiple thread blocks. When thread blocks terminate on a multipro-

[1]We adopt the terminology from NVIDIA GPUs[36], but the concepts are applicable to other GPUs (e.g., AMD GPU).

cessor, the multiprocessor will execute remaining thread blocks from the grid. Each multiprocessor contains a small number of *warp schedulers*. Each warp scheduler keep a pool of *active warps* (i.e., warps in execution status). At each cycle, each warp scheduler issues instructions from an *eligible warp* in its active warps that is eligible to execute it current instruction. Each thread in a grid uses a fixed number of registers and the number is determined during compilation. Each thread block in a grid uses a fixed size of shared memory that is determined during compilation or kernel launching. The number of registers of each thread and the size of shared memory of each thread block limit the maximum number of resident thread blocks on each stream multiprocessor.

Compared to CPU, GPU performs better on tasks that can be decomposed into massive independent subtasks while CPU performs better on task with strong serial dependency. There are a lot of transistors in CPU are not used for the actual computation units (e.g., ALU), but for the components reducing single thread execution interrupting, (e.g., large cache, speculative execution, and branch prediction). However, in GPU, transistors are mostly used for allowing more concurrent computation (e.g., large number of ALUs and large register file). In CPU, each thread has its own execution status (i.e., context that contains the values in each registers). When a processor swaps to another thread, it is necessary to save the current thread's context, which incurring a series of high cost operations. However, instead of doing the expensive context saving and restoring on CPU, when GPU wants to switch to other threads, GPU can directly issue the instructions in other threads, because GPU is designed to keep the execution contexts of all active threads on the chip. That's why GPU's register file is so large. With this design, GPU is able to do very lightful context switch when one thread (more precisely, warp) is stalled by some reason (e.g., memory access, hardware pipeline dependency). As long as there are eligible warps every cycle, each warp scheduler is able to issue active warp's instruction and hide the latency of other stalled active warps. Thus, it is important to keep a large number of active warps on GPU.

2.2 Convolution Neural Network

We usually represent a network by a data flow graph. In the graph, each node is an operator (also called layer). Each edge represents a tensor that is the output of the source operator and an input of destination operator. The input data flows following the data flow graph, is precessed by each operator it passed by, finally and reaches the end of data flow graph.

The input of the convolution neural network (CNN) is usually a tensor with shape (B, C, H, W), where B is the batch size, C is the number of channels, H and W are the height and width of the input image. We usually use a large batch size for training (e.g., $B = 32$) to have a better device utilization; on the other hand, we usually use single batch (e.g., $B = 1$) for inference to reduce end to end latency. The input of a CNN usually has 3 channels, representing the intensity of reg, green and blue colors, respectively. The H and W depends on the the input image itself. For ImageNet[11], we usually normalize the images to size 224×224.

2.2.1 Network Layers

A convolution neural network (CNN) usually consists of network layers such as convolutions, pooling, normalizations, and activations.

Convolution Layer

The convolution layer is the main layer in a convolution neural network. Its input is a tensor with shape $(N, C_{in}, H_{in}, W_{in})$. Each convolution has a weight tensor with shape $(C_{out}, C_{in}, K_H, K_W)$. Its output is also a tensor with shape $(N, C_{out}, H_{out}, W_{out})$. Each convolution also has some parameters. Besides number of output channels C_{out}, number of input channels C_{in}, kernel size K_H, K_W, each convolution layer also has the stride parameters (S_H, S_W) and padding parameters (P_H, P_W). The output image-dimension size can be derived from the input size and the parameters

$$H_{out} = \lfloor \frac{H_{in} + 2P_H - K_H}{S_H} + 1 \rfloor \qquad W_{out} = \lfloor \frac{W_{in} + 2P_W - K_W}{S_W} + 1 \rfloor$$

Each element of the output tensor is defined by formula 2.2.1:

$$Y[n, c_{out}] = \sum_{c_{in}=0}^{C_{in}} X[n, c_{in}] \star Weight[c_{out}, c_{in}]$$

where X is the input tensor, Y is the output tensor, $Weight$ is the weight tensor, $\star$ is a 2-D correlation operator.

For example, when kernel size and stride are 1×1 and padding size is 0, we have formula 2.2.1

$$Y[n, c_{out}, h, w] = \sum_{i=0}^{K_H-1} \sum_{j=0}^{K_W-1} \sum_{c_{in}=0}^{C_{in}} X[n, c_{in}, h+i, w+i] * Weight[c_{out}, c_{in}, i, j]$$

The convolution layer also has other dimensions(e.g., 3-D or 1-D), but the commonly used one for image tasks is the 2-D convolution defined above.

There are some variants of convolution are proposed to reduce the computation required by above regular convolution. To deploy CNNs to mobile device, MobileNet[41] is proposed to use separable convolutions. A separable convolution contains two adjacent convolutions. The first one is a convolution that with the same number of input channels and output channels. And it only conducts the channel-wise 2d correlation operation. Such a convolution is called depth-with convolution. The second convolution is a regular convolution with kernel size 1×1.

Pooling Layer

Pooling layer is used to aggregate the information and down sampling in the spatial dimension (H and W). Two kinds of pooling are commonly used, maximum pooling and average pooling. There are also kernel size and padding size parameters for pooling layer. When padding is zero, we can define each output element as in formula 2.2.1.

$$Y[n, c_{out}, h, w] = \frac{1}{K_H \times K_W} \sum_{i=0}^{K_H-1} \sum_{j=0}^{K_W-1} X[n, c_{in}, h * S_H + i, w * S_W + i]$$

This formula defines the average pooling, and the maximum pooling is similar but replacing the averaging of the subregion into its maximum value.

Normalization Layer

Normalization layer is used to update the value distribution. It is an effective way to make the training more stable and achieve better performance. The most commonly used normalization in CNN models is batch normalization[20].

The batch normalization will compute the average and standard deviation of all elements varying in batch dimension and spatial dimensions. Given the input tensor with shape (N, C, H, W), batch normalization first compute the average and standard deviation of each slice of (N, H, W). Then, for each slice of (N, H, W), it performs a element-wise linear computation as follows

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} \times \alpha + \beta$$

where α and β are optional affine parameters. During training, we usually calculate the running mean and running standard deviation. During inference, we fixed the mean and standard deviation, which makes the batch normalization a normal element-wise linear layer. A common optimization for inference is to fuse the batch normalization into the adjacent convolution layer.

Activation Layer

Activation layer is a element-wise non-linear function on the input tensor. The commonly used activation function in CNNs is ReLU [3]. The ReLU activation function is defined as follows

$$y = \text{ReLU}(x) = \max(x, 0).$$

A common used design pattern of CNNs is Conv-BatchNorm-ReLU.

Concatenation Layer

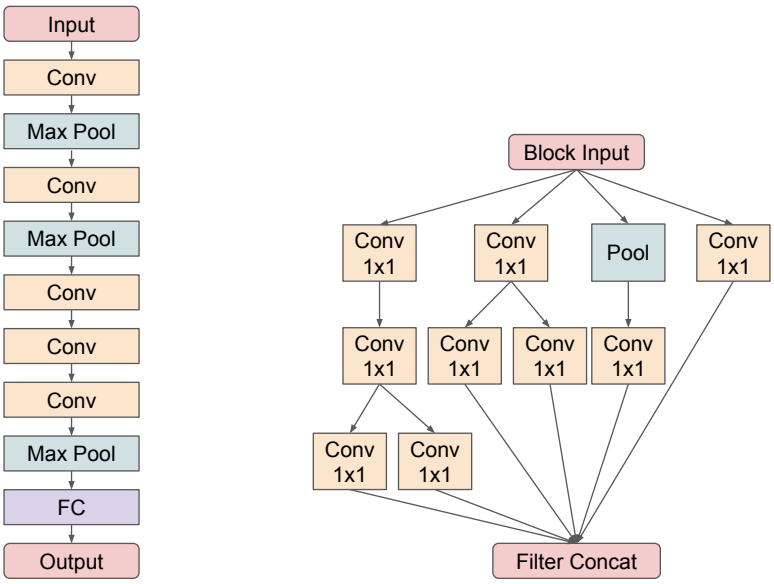
In deep learning, we may concatenate two tensors along one dimension to get a new tensor, which helps us to extract information from both tensors in following layers. In CNNs, we usually concatenate the tensors along the channel dimension.

2.2.2 Dataflow Graph Structure

We can classify the convolution neural networks into two categories in terms of whether it has the parallelism between operators (i.e., inter-operator parallelism). One category includes the CNNs with sequential structure (e.g., AlexNet[26]), and the other category includes CNNs with multi-branch structure (e.g., Inception V3[48]). Figure 2.2 demonstrates the two kinds of convolution neural networks.

The skip connection[17] is proposed to alleviate the gradient vanishing problem[39], which allows us to design much deeper network. With skip connection, the common practice to design the convolution neural networks becomes to design the network blocks first, then stacking the blocks.

Besides the hand-crafted convolution neural networks, neural architecture search (NAS)[56, 29, 4] is proposed to search the architecture (i.e. network structure) of CNNs. One important class of CNNs found by NAS contains multiple branches[49, 52].



(a) AlexNet (b) An Inception V3 Block

Figure 2.2: Two Categories of Convolution Neural Networks. The networks in the first category contains a sequence of layers with strictly sequential dependency, such as AlexNet in (a). The ones in the second category contains layers with branched-structure, such as Inception V3 in (b). To simplify the figure, we omitted the normalization layer and activation layer after each convolution layer.

For both categories of CNNs, existing frameworks execute each CNN layer by layer on GPU. This make sense when the parallelism with each operator (e.g., convolution) is able to saturate the whole GPU. However, this assumption no longer holds when the network designers tend to use smaller convolutions while the underlying GPU is getting more powerful, as shown in Figure 1.1. This trend leads to serious low-utilization of the GPU.

2.3 Low Utilization Problem

There are several reasons of low device-utilization when we execute a neural network on GPU, as shown in Figure 2.3, and the low-utilization can be alleviated via running multiple kernels on the GPU simultaneously.

Low Occupancy

Each stream multiprocessor has a limited number of warp schedulers (e.g., 2 for Fermi[37] architecture and 4 for Kepler[38] architecture). Each warp scheduler maintain a pool of limited active warps. Thus, there is a limit on active warps on a stream multiprocessor. We usually call the ratio of the actual number and maximum number of the active warps *occupancy* of the kernel. GPU heavily depends on the inter-thread parallelization and fast thread context switch to hide the long latency of instruction execution. Thus, a large number of active warps (i.e., high occupancy) are important to achieve high throughput. However, this is not always true. When we design the kernel for computation workload that will access the inputs multiple times (e.g., matrix multiplication

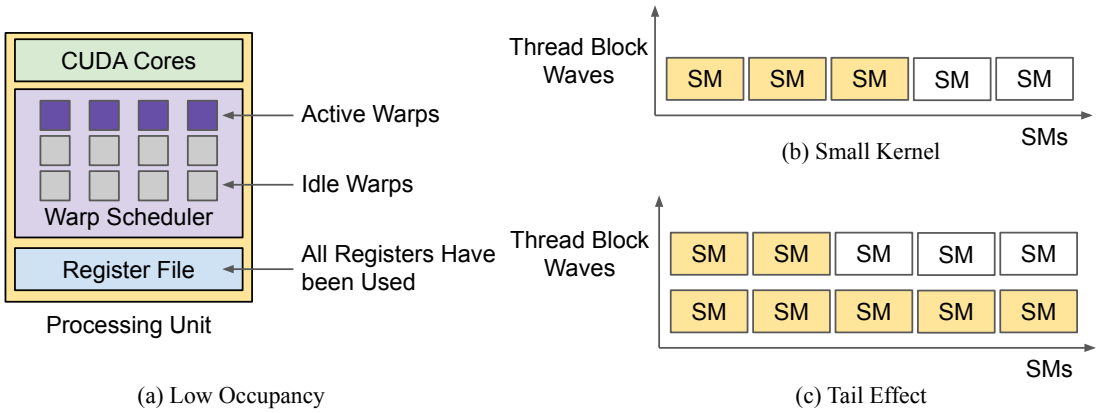


Figure 2.3: Three reasons of GPU low utilization.

and convolution), it is important to reduce the global memory access by caching the loaded data in shared memory and registers and reuse them. The usage of large number registers and shared memory limits the number of resident thread blocks and thus limits the occupancy. Some kernels like element-wise operation (e.g., element-wise addition) can be parallelized with this kernel to utilize the idle warp schedulers. Taking a real convolution kernel from cuDNN[8] as an example, it has 256 threads in a thread block and each thread has 126 32-bit registers. Thus, this thread block needs $256 \times 126 \times 4$ bytes $= 126$ KiB registers. In most modern NVIDIA GPUs, the register file for a multiprocessor has only 256 KiB, which means we can only have 2 resident thread blocks on a multiprocessor. The theoretical maximum number of active warps is $256/32 * 2 = 16$. If we run the kernel on a NVIDIA GPU with compute capacity 8.6 (e.g., RTX 3070 Laptop) that supports concurrent execution of 48 warps, the theoretical occupancy is only 33.3%.

Kernel is Small

Another reason of low utilization is that the kernel for network inference can be very small. Network inference usually takes small batch size (e.g., batch size 1), which makes the kernel has limited number of thread blocks. For example, if we use the kernel in above example to execute a convolution with batch size 1, input channels 375, output channels 750, kernel size 3×3, strides 1×1 and input image size 15×15, we will get only 48 thread blocks. If we run on a GPU in above example that has 40 SMs, only about half of the SMs will be used and the remaining half would be in idle status. This under-utilization would get more severe when the convolution gets smaller.

Tail Effect

Existing reports [32] have found tail effect when we run a kernel. A grid is finished only when the last thread block reach to the end of its execution. At the very late stage of execution, a lot of SMs are in idle status. This effect is not a big problem if we have a lot of waves of thread blocks to be executed because the tail effect in the last wave would be amortized by the large number of waves. However, the tail effect would be severe when the number of thread block waves that would be distributed to the GPU is small.

With inter-operator parallelization, above problems will be largely alleviated. If we launch a

kernel with low theoretical occupancy due to large register usage with another kernel with very low level of register usage (e.g., element-wise addition), the overall occupancy can be improved. We can launch multiple small convolution kernels to have enough thread blocks filled into the stream multiprocessors, which alleviates the under-utilization problem caused by small kernels. When we launch multiple kernels concurrently, the tail effect for each kernel would also be alleviated greatly because when a kernel is about to finish, we can use the thread blocks from other kernels to saturate the idle multiprocessors and the tail effect will only appear at the end of all executed kernels.

However, it is non-trivial to get a high performance inter-operator parallelization schedule. Figure 1.2 shows that the greedy schedule that launches all available kernels each time is sub-optimal because it does not consider the neural network holistically. In the next chapter, we will formulate the inter-operator scheduling problem, and introduce our proposed IOS (Inter-Operator Scheduler) to address this problem.

2.4 Classical Task Scheduling

In this section, we give the definition of classical task scheduling and existing methods to address this problem. Finally, we explain why existing works do not work well on the inter-operator scheduling problem on GPU.

2.4.1 Task Graph and Schedule

Task scheduling has existed for decades. In classical task scheduling[44] problem, the input is a directed-acyclic graph called task graph.

Definition 1 (Task Graph)**.** A task graph is a directed acyclic graph (DAG) $G = (V, E, w, c)$. The target workload consists of tasks in V. Each directed edge $e = (u, v)$ represents a dependency between task u and task v (i.e., task v depends on u). Each node u in V has a computation cost $w(u)$, representing the computation time used to completes task u. Each edge $e = (u, v)$ also has a communication cost $c(e)$, representing the time used to transfer the output of u to v when needed.

Given the task graph and number of processors, the output of a scheduling algorithm is a *schedule* defined below.

Definition 2 (Schedule)**.** Given a task graph $G = (V, E, w, c)$ and the number of processors n, a schedule is a function pair $(t_s, proc)$, where

- $t_s : V \to Q_0^+$ is the start time function. $t_s(u)$ is a non-negative number representing the start time of task u.
- $proc : V \to P$ is the processor allocation function. Task u will be executed by processor $proc(u)$.

There are two components in a schedule: the start time function and the processor assignment function. The start time function is task's temporal assignment and the processor assignment is a spatial assignment. A valid schedule must make sure

- No two tasks are running at the same processor at the same time.

- If there is a path from task u to task v in the task graph, task v can only start after completion of task u (i.e., $t_s(u) + w(u) <= t_s(v)$).

It has been proved that classical task scheduling problem is NP-hard [44]. The existing state-of-the-art heuristic algorithms [51] for classical task scheduling are list scheduling, clustering and their variants.

2.4.2 List Scheduling

The idea of list scheduling[2, 9, 18, 23, 24, 45, 53] is assign the processor of each task by a given order. There are two steps in the algorithm.

1. Sort the nodes in task graph according to a priority scheme and precedence constraint. The precedence constraint indicates that the order is a topological order of the task graph. The priority scheme allows us to assign the important tasks to processor first. There are different ways to choose the priority scheme.

2. Enumerate the nodes following the order found in previous step, for each node u

   (a) Choose a processor p to run task u,

   (b) Assign the start time of u on processor p, and make sure there is no overlap with existing tasks on processor p.

The schedule satisfies the precedence constraint because it schedule tasks in a topological order of task graph. It assigns the start time by avoiding overlap with existing tasks. Thus the schedule generated by list scheduling is valid. The question is how to choose priority scheme and how to choose the processor for each u.

One commonly used priority scheme is to put high priority on the tasks on critical path. When list scheduling chooses the processor p for task u, a commonly used heuristic method is to choose a processor to make the start time of u as early as possible.

2.4.3 Clustering

The idea of list scheduling is to assign each task according to a list of given tasks. The idea of clustering [42, 10, 28, 15, 25, 42], however, is to first determine which tasks should be assigned to the same processor. The clustering heuristic usually follows the following three steps

1. Find a partition of the tasks in task graph. Each component in the partition is called a cluster.

2. Because the number of clusters may be more than the number of processors, we need assign each cluster to a processor.

3. Assign the start time of each task to make it satisfy the non-overlap constraint and precedence constraint.

The motivation [42] of clustering is that, given two tasks should be run on the same processor when there is no constraint on the number of processors, they should also be executed on the same processor in any real system that has limited number of processors.

2.4.4 Challenges of Existing Methods

One key difference between the inter-operator scheduling problem on GPU and classical task scheduling problem is that the parallel execution of multiple tasks on GPU will interfere with each other. The modern GPUs have multiple processors (e.g., stream multiprocessor on NVIDIA GPU) and a task (e.g., a GPU kernel) is executed on all processors. It is different from the multi-core CPU and multi-nodes network where each task is executed on a single processor. Besides this, we can only get the parallel execution time for kernel A and B on GPU by actual measurement, and it is hard to infer the execution time of parallel execution from the separate execution time of A and B. Thus, these algorithms can not be directly used in the task scheduling problem on GPU.

Chapter 3

Methods

This chapter introduces our Inter-Operator Scheduler (IOS) in four parts. Section 3.1 formulates the problem IOS wants to solve. Section 3.2 elaborates the IOS design in details. Section 3.3 gives the time complexity of IOS and its proof. Section 3.4 introduces the pruning optimizations to reduce the search time of IOS.

3.1 Problem Formulation

This section defines the *schedule* in IOS and formulates the problem.

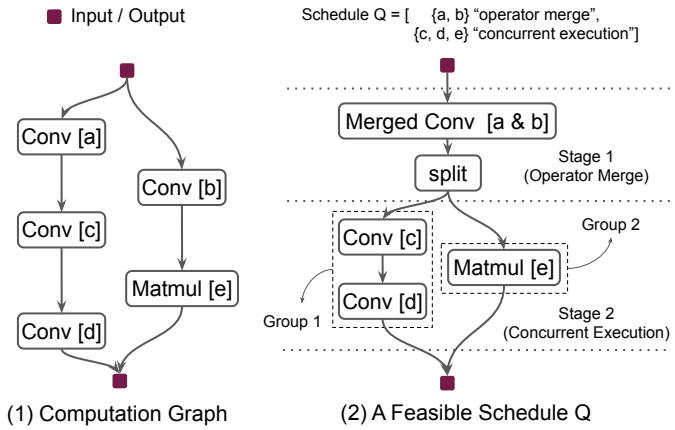


Figure 3.1: For a given *computation graph* (left), a possible *schedule* is shown to the right. There are five operators in the graph: convolutions a-d and matrix multiplication e. The schedule partitions operators into 2 *stages*. The first stage merges convolution a and b into a larger convolution; this parallelization strategy is named *operator merge*. The second stage partitions operator c, d and e into two *groups*, {c, d} and {e}. The operators in the same group are executed sequentially while different groups in the same stage are executed concurrently. This parallelization strategy is named *concurrent execution*. Stages are executed one-by-one.

Computation Graph. A CNN is defined by a computation graph $G = (V, E)$, where V is the set of operators, and E is the edge set representing dependencies. A computation graph is a directed acyclic graph (DAG). Each operator in the graph represents an operator such as convolution and

matrix multiplication. Each edge (u, v) is a tensor that is an output of operator u, and an input of operator v. Figure 3.1 (1) shows the computation graph of a simple CNN.

Stage. To take advantage of inter-operator parallelism in a CNN architecture, its computation graph is partitioned into multiple stages. Stages are executed sequentially and the operators in the same stage are executed according to a certain parallelization strategy. Figure 3.1 (2) shows a possible schedule that partitions the input graph into two stages, where the first stage contains operator a and b, and the second stage contains operator c, d, and e. The parallelization strategy is discussed below.

Parallelization Strategy. Each stage adopts one of the following two parallelization strategies: *operator merge* and *concurrent execution*. ISO considers both of them and automatically picks the more efficient one for each stage. The choice depends on operator types, input tensor shapes, and the hardware device to perform CNN computations.

To be eligible for *operator merge*, the operators' type must be the same while the hyperparameters can be different. For example, two convolutions with the same stride but different kernel sizes can be merged. The smaller kernel will be padded with zeros to fit the large kernel, so we can stack their kernels together. In Figure 3.1 (1), if `Conv[a]` has 128 3x3 kernels while `Conv[b]` has 256 3x3 kernels, we can stack their kernels together and replace `Conv[a]` and `[b]` by a `Merged Conv[a&b]` with 384 3x3 kernels. Besides increasing parallelism, it also reduces the memory accesses to the input tensor from twice to only once. A split operator is required to partition the merged convolution's output to recover the original outputs of `Conv[a]` and `Conv[b]`.

Under *concurrent execution*, the operators in the stage are partitioned into disjoint *groups*. More specifically, if two operators are connected by an edge, they are partitioned into the same group. Different groups within the same stage are executed concurrently, while the operators within the same group are executed sequentially. IOS considers simultaneous executions of operators with *different* types. In the second stage of Figure 3.1 (2), the three operators are partitioned into two groups. The first group contains operators `Conv[c]` and `Conv[d]` while the second group contains operator `Matmul[e]`. The two groups are executed concurrently while `Conv[c]` and `Conv[d]` are executed sequentially in their group.

Schedule. We define a *schedule* Q of a computation graph G as

$$Q = \{(S_1, T_1), (S_2, T_2), \ldots, (S_k, T_k)\},$$

where S_i is the set of operators in the ith stage and T_i is the corresponding parallelization strategy, either "concurrent execution" or "operator merge". For example, the schedule for Figure 3.1 (2) is:

$$Q = \{(\{a, b\}, \text{operator merge}), (\{c, d, e\}, \text{concurrent execution})\}.$$

The schedule Q executes the network from the first stage (S_1, T_1) to the last stage (S_k, T_k) sequentially. S_i may contain only one operator if it is the best choice (e.g., a very large operator that saturates the entire GPU).

Problem Formulation. Let c be a cost function defined on a computation graph G and schedule Q. We aim to find a schedule Q^* to minimize the cost function for a given computation graph G, i.e., $Q^* = \text{argmin}_Q c(G, Q)$. In this work, the cost function $c(G, Q)$ is defined as the latency of running G following schedule Q.

3.2 Inter-Operator Scheduler (IOS)

3.2.1 Dynamic Programming

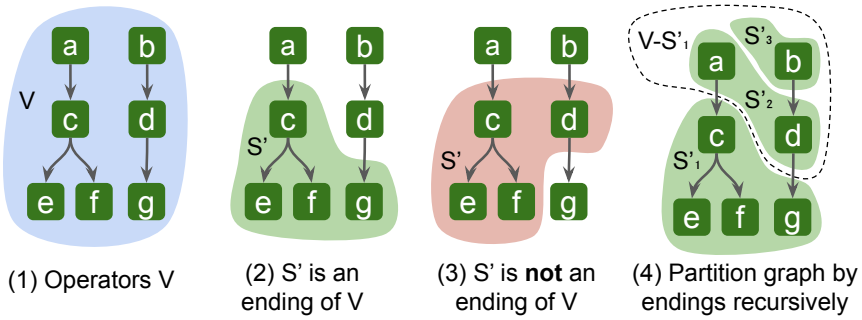


Figure 3.2: The illustration of *ending*. (1) shows all the operators V. S' in (2) is an ending of V. However, S' in (3) is not an ending of V because there is an edge from d to g (from S' to $V - S'$). We can partition a graph by selecting an ending for remaining operators recursively, as shown in (4), where S'_1 is an ending of V while S'_2 is an ending of $V - S'_1$.

To find an optimized schedule for a CNN architecture, we first partition its computation graph $G = (V, E)$ into $V - S'$ and S', where all edges between $V - S'$ and S' start from $V - S'$ and end in S'. Such S' is called an *ending* of V, as illustrated in Figure 3.2. There can be many endings of V. The last stage's operators in V's optimal schedule must be an ending of V. We can enumerate the ending S' of V and convert the original problem to a sub-problem that finds the optimal schedule for $V - S'$. The whole graph can be scheduled by applying the partition recursively.

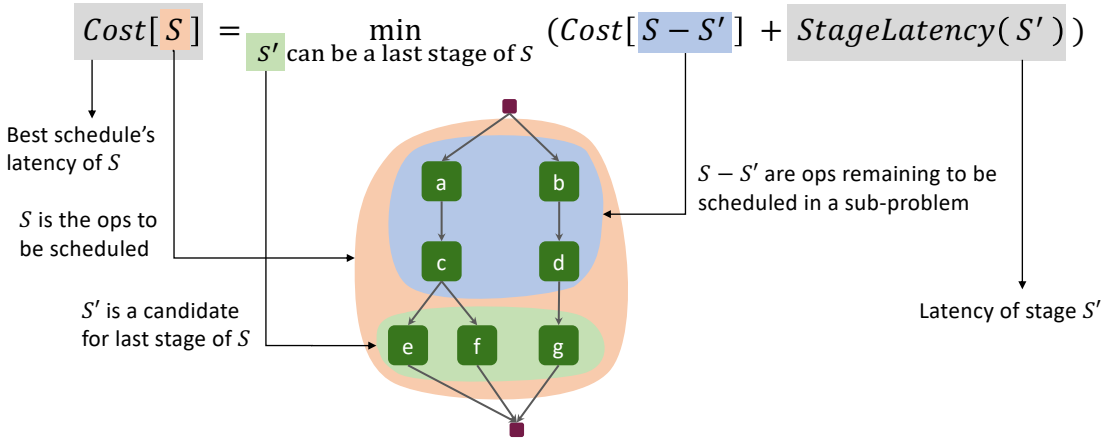


Figure 3.3: Dynamic programming algorithm used to find the optimal schedule given the cost model.

Let cost$[S]$ be the latency of an optimal schedule for S. Let StageLatency$[S']$ be the latency of stage (S', T) where T is the better parallelization strategy for S' among the two possible ones. We formalize this idea as the formula in Figure 3.3. In the formula, S' is an ending of S, and cost$[\varnothing] = 0$. Finally, cost$[V]$ is the latency of an optimal schedule for the entire computation graph G. To construct the optimal schedule we found, we record the corresponding S' that minimizes the latency for each S (i.e., cost$[S]$) in choice$[S]$.

Algorithm 1 Inter-Operator Scheduler (IOS)

Input: a computation graph $G = (V, E)$,
and a schedule pruning strategy P
Output: a schedule found by IOS

```
1: Let cost[S] = ∞ for all S ⊆ V but cost[∅] = 0
2: Let choice[S] = ∅ for all S ⊆ V
3: function InterOpeatorScheduler(G)
4:     V = all operators in computation graph G
5:     Scheduler(V)
6:     Q = empty list
7:     S = V
8:     while S ≠ ∅ do
9:         S', T = choice[S]
10:        Insert stage (S', T) before the head of Q
11:        S = S − S'
12:    return the schedule Q
13: function Scheduler(S)
14:    if cost[S] ≠ ∞ then
15:        return cost[S]
16:    for all ending S' of S satisfying pruning strategy P do
17:        L_S', T_S' = GenerateStage(S')
18:        L_S = Scheduler(S − S') + L_S'
19:        if L_S < cost[S] then
20:            cost[S] = L_S
21:            choice[S] = (S', T_S')
22:    return cost[S]
23: function GenerateStage(S')
24:    Partition S' into disjoint groups: S'_1, S'_2, ..., S'_k.
25:    L_concurrent = latency of parallel execution of {S'_i}
26:    if operators in S' can be merged then
27:        L_merge = latency of merged operator
28:    else
29:        L_merge = ∞
30:    if L_concurrent < L_merge then
31:        return L_concurrent, "concurrent execution"
32:    else
33:        return L_merge, "operator merge"
```

3.2.2 Pseudo Code Implementation

With this general idea, we implement IOS in three functions InterOperatorScheduler (L3-12), Scheduler (L13-22) and GenerateStage (L23-33), as shown in Algorithm 1. InterOperatorScheduler takes a computation graph as an input and returns the optimal schedule found by IOS. Scheduler is a recursive function implementing the dynamic programming algorithm to find the optimal schedule for a subset of operators in G. GenerateStage chooses a better parallelization strategy for given operators S'.

InterOperatorScheduler (L3-12) is the entry function. It takes a computation graph G as an input and returns an optimized schedule Q. This function calls Scheduler with operators V as an argument (L5). After calling Scheduler, the global variable cost[S] stores the latency of an optimal schedule for S, while choice[S] stores the last stage in the corresponding optimal schedule. Once choice[$\cdot$] is obtained, we can construct the schedule found by IOS (L6-11). We start with an empty list as the initial state of our schedule (L6) and let S be all the operators in G. We inquire about

the last stage (S', T) of S by choice[S] and put it at the head of the current schedule Q. We repeat this process by letting $S = S - S'$ to get the remaining operators' schedule in all previous stages (L8-11). $S = \varnothing$ indicates that we have discovered an optimized schedule Q for G.

Scheduler (L13-22) is the core part of our algorithm. It implements the dynamic programming algorithm recursivly, taking a subset of V as the state. It takes a set of operators S as an input and returns the minimal latency for S among all schedules. Because Scheduler may be called multiple times with the same argument S, for repeated calls, we cache the previous results cost[S] to avoid redundant computations (L14-15). To find an optimal schedule for S, we enumerate its last stage operators S' and reduce the problem into a sub-problem for $S - S'$ (L16-21). We use GenerateStage to choose a better parallelization strategy $T_{S'}$ for S' and get the latency $L_{S'}$ (L17). L_S is the minimal latency for S when taking S' as the last stage's operators (L18). We enumerate all possible endings of S and record the minimal latency L_S and the corresponding last stage $(S', T_{S'})$ in cost[S] and choice[S], respectively (L19-21).

GenerateStage (L23-33) chooses a better parallelization strategy from "concurrent execution" and "operator merge" for a given stage S'. It returns the parallelization strategy and the corresponding latency. It directly measures the latencies of both parallelization strategies on the hardware. The "concurrent execution" strategy partitions S' into multiple disjoint operator groups: $S'_1, S'_2, ..., S'_k$. Operators in different groups are executed concurrently while operators in the same group are executed sequentially. For the "operator merge" strategy, if all the operators in S' can be merged into a single operator (L26), we merge them and measure the latency of the merged operator (L27). Otherwise, we set L_{merge} to infinity to force ourselves to choose the "concurrent execution" strategy.

3.2.3 An Example

Figure 3.4 demonstrates how IOS discovers an optimized strategy for an input graph with three operators a, b, and c. Figure 3.4 (2) shows the dynamic programming process, the Scheduler in Algorithm 1. For simplicity, we only consider the concurrent execution parallelization strategy. There are six *states* (the operators to be scheduled, S) in the process. We start with all the operators in the computation graph as state $S = \{a, b, c\}$ (L5). For each state S, Scheduler enumerates the ending S' of S. The latency of S contains two parts: latency of S' as a stage and the latency of $S - S'$. While the result of S' is measured on the device directly ($L_{S'}$), the optimal latency of $S - S'$ is obtained via solving the sub-problem recursively. ❶ to ⓬ shows the computation path. Note that IOS memorizes the results for each calculated state to avoid redundant computations. Thus, step ❼ visits state $S = \{a\}$, and IOS gets its latency directly (L15) because it has been previously visited by step ❷. Scheduler stores the latency (cost[$\cdot$]) and last stage (choice[$\cdot$]) in its optimal schedule. We can construct the best schedule for the whole computation graph using choice[$\cdot$], as shown in Figure 3.4 (3). An optimal schedule found by IOS is shown in (4). Both stages take "concurrent execution" as the parallelization strategy.

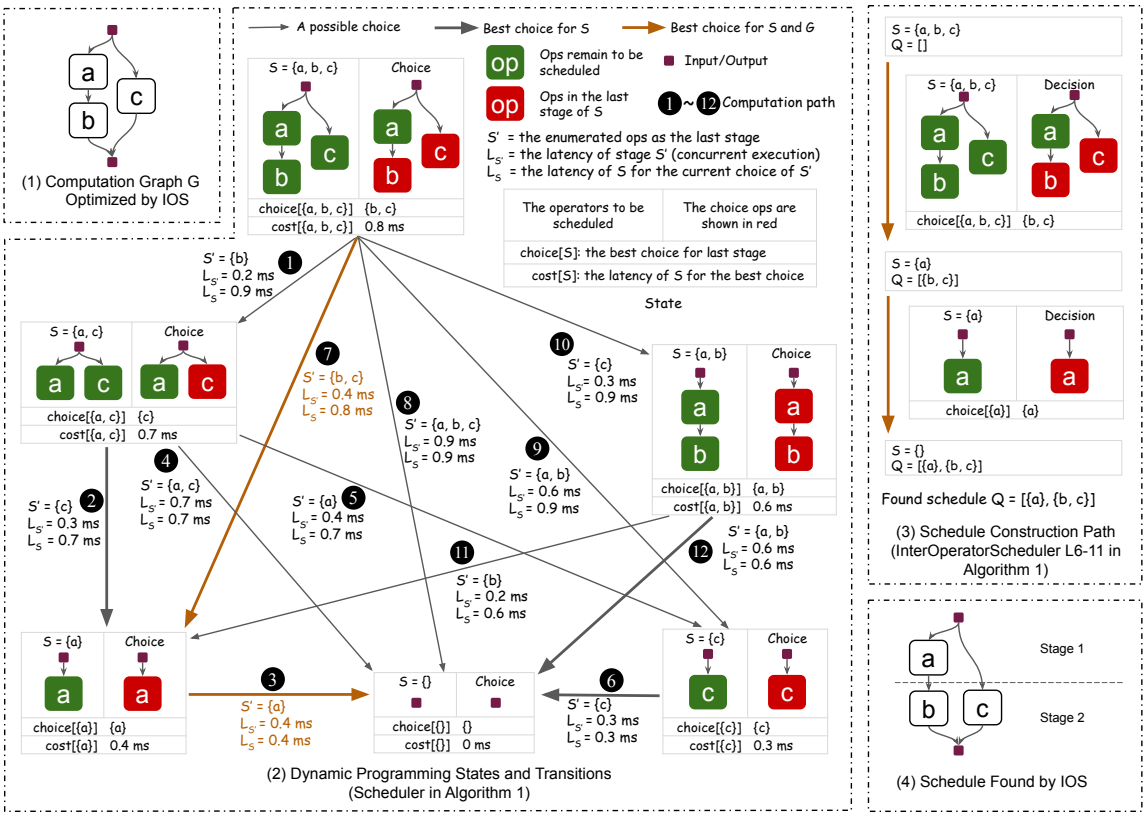


Figure 3.4: An example to illustrate how IOS finds the schedule. The computation graph to be optimized is shown in (1). It has three operators, a, b, and c, where a is followed by b, and c is independent with a and b. The states and transitions between these states are presented in (2). Here *state* means the operators to be scheduled, and *transition* means the dependency between states (edges in (2)). Any path from state $S = \{a, b, c\}$ to $S = \{\}$ is corresponded with a schedule. Upon finishing the dynamic programming process (Scheduler), the best schedule for the computation graph can be constructed according to choice[·], as shown in (3). The schedule found by IOS is shown in (4). For simplicity, in this example, we only consider the concurrent execution parallelization strategy.

3.3 Time Complexity of IOS

3.3.1 Exponential to Graph Width instead of Size

In this subsection, we give the time complexity of IOS. We take set operations (L18, L24) and latency measurement operations (L25, L27) as atom operations to make the analysis clear. To analyze the time complexity of IOS, we count the number of executions of L17-21, since they dominate the whole algorithm's execution. This number equals the number of edges (i.e., transitions) in Figure 3.4 (2). Furthermore, it is equivalent to count the number of pairs (S, S'), where S is a state and S' is an ending of S. Here we define the width of a directed acyclic graph and provide the time complexity of Algorithm 1.

Definition 3 (Width d of a DAG)**.** We call d the *width* of a directed acyclic graph G if we can find at most d operators in G such that there is no path connecting any two of them.

Theorem (Time Complexity of IOS)**.** The time complexity of Inter-Operator Scheduler (IOS) is $\mathcal{O}(\binom{n/d+2}{2}^d)$, which can be relaxed to $\mathcal{O}((\frac{n}{d}+1)^{2d})$, where n is the number of operators in the

computation graph and d is its width.

In fact, there are computation graphs that can reach this bound, so we can not improve it without other restrictions on the schedule space. Proof can be found in the following subsections.

Model	n	d	$\binom{n/d+2}{2}^d$	$\#(S, S')$	#Schedules
Inception V3	11	6	2.6×10^4	4.9×10^3	3.8×10^6
Randwire	33	8	3.7×10^9	1.2×10^6	9.2×10^{22}
NasNet	18	8	5.2×10^6	3.1×10^5	7.2×10^{12}
SqueezeNet	6	3	2.2×10^2	51	1.3×10^2

Table 3.1: For the largest block of each benchmarked network, we list the number of operators n, the width d, the upper bound of transitions $\binom{n/d+2}{2}^d$, the real number of transitions $\#(S, S')$, and number of schedules.

Modern convolution neural networks usually construct the network by stacking multiple blocks, making it possible to optimize each block separately. In this case, n and d refers to the number of operators within a block and the block width, rather than the full network. We list the information of the largest block for each network benchmark in Table 3.1.

The total number of feasible schedules is exponential to the number of operators (e.g., up to 9.2×10^{22} for Randwire [52]). Such a huge number makes it prohibitive to manually design or enumerate the schedules. However, by reusing the results of common sub-schedules in the schedule finding process, IOS finds the optimal schedule within 4 hours for each network with no pruning strategy used. The time complexity of IOS is only exponential to the width of the computation graph, which is usually very small and acceptable (e.g., ≤ 8 in all benchmarked networks).

3.3.2 Preliminary Definitions and Theorems

In this subsection, we give the definition of chain and anti-chain, Dilworth's theorem [13], and a corollary, which is used in our proof later.

Definition 4 (Chain and antichain)**.** A *chain* is a subset of a partially ordered set such that any two distinct elements in the subset are comparable. An *antichain* is a subset such that any two distinct elements in the subset are incomparable.

Definition 5 (Chain decomposition of partial order set)**.** A *chain decomposition* of a partial order set is a partition of the elements of the ordered set into disjoint chains.

Theorem (Dilworth's Theorem)**.** In any finite partially ordered set, the largest antichain has the same size as the smallest chain decomposition.

We apply the Dilworth's theorem to a directed acyclic graph and can get the following corollary.

Corollary 1. Let $G = (V, E)$ be a directed acyclic graph and d be the width of G. We can decompose V into d sets such that any two vertices in the same set can be connected by a path in G.

Proof. Let $P = (V, E')$ be the partial order derived from G by transitive closure. Then that two elements u, v in V are comparable in P is equivalent to that there is a path between them in G.

Thus, the width d of G equals the size of largest antichain of P. We apply the Dilworth's Theorem to P and can get a decomposition of V into d chains in P: $S_1, S_2, \ldots, S_d$. Because S_i is a chain in P, any two elements in S_i are comparable, which means there is a path bridge them in G. □

3.3.3 Time Complexity Proof

In this subsection, we will prove the time complexity of IOS. Then we will show that the upper bound can be reached by some computation graph.

Lemma 1. If S'_1 ends S and S'_2 ends $S - S'_1$, then $S'_1 \cup S'_2$ also ends S (S' ends S means that S' is an ending of S).

Proof. We prove it by contradiction. If $S'_1 \cup S'_2$ does not end S, there must exist $(u, v) \in E$ such that $u \in S'_1 \cup S'_2$ and $v \in S - S'_1 \cup S'_2$. Then we have $u \in S'_1$ or $u \in S'_2$. If $u \in S'_1$, we can get the contradiction that S'_1 is not an ending of S because $v \in S - S'_1 \cup S'_2 \subseteq S - S'_1$. If $u \in S'_2$, we can also get the contradiction that S'_2 is not an ending of $S - S'_1$ because $v \in S - S'_1 \cup S'_2 = (S - S'_1) - S'_2$. □

Lemma 2. Let S be a possible argument of SCHEDULER, we have $V - S$ ends V.

Proof. We can rewrite S as $S = V - \bigcup_{i=1}^{m} S'_i$, where $m \geq 0$ and S'_k ends $V - \bigcup_{i=1}^{k-1} S'_i$ according to L17 in Algorithm 1. By repeating apply Lemma 1, we can get that $\bigcup_{i=1}^{m} S'_i$ ends V, which means $V - S$ ends V. □

Lemma 3. Let V' be a subset of V and any two operators in V' are bridged by a path. Let c be the size of V'. Then

$$|\{(S \cap V', S' \cap V') \mid S' \text{ ends } S,\ V - S \text{ ends } V\ \}| = \binom{c+2}{2}$$

Proof. Because any two operators in V' is bridged by a path in G, operators in V' are ordered sequentially. Because $V - S$ ends V, there are only $c+1$ possible sets of $S \cap V'$ because S must be a prefix in the sequential ordered operators, including empty set. $S' \cap V'$ is a suffix of $S \cap V'$, including empty set. Then there are $\sum_{i=0}^{c} \sum_{j=0}^{i} 1 = \frac{(c+2)(c+1)}{2} = \binom{c+2}{2}$ possible pairs of $(S \cap V', S' \cap V')$. □

Theorem. The time complexity of inter-operator scheduler is $\mathcal{O}(\binom{n/d+2}{2}^d)$, which can be relaxed to $\mathcal{O}((\frac{n}{d}+1)^{2d})$, where n is the number of operators in the computation graph and d is its width.

Proof. We only need to count the number of pairs of (S, S') that can reach L17 of Algorithm 1 because L17-21 dominates the execution time of the scheduler, where S is a subset of V that is taken as the argument of SCHEDULER and S' is an ending of S. By Lemma 2, $V - S$ ends V. By Corollary 1, we can decompose V into d disjoint partitions $V_1, V_2, \ldots, V_d$ and any two operators u, v in the same partition can be bridged by a path in G. We can build a one-to-one mapping that maps pair (S, S') to $2d$-dimension tuple $(S \cap V_1, S' \cap V_1, \ldots, S \cap V_d, S' \cap V_d)$ based on the partition. Then we only need to count the number of valid tuples to get the number of valid pairs. By Lemma 3, the possible number of pairs $(S \cap V_i, S' \cap V_i)$ is $\binom{c_i+2}{2}$. Then an upper bound of the tuples is $\prod_{i=1}^{d} \binom{c_i+2}{2}$. It is an upper bound but not the exact number because currently we only consider the dependency inside each partition V_i and ignored the dependency between different partitions. So the upper bound of the number of pairs of (S, S') is $\prod_{i=1}^{d} \binom{c_i+2}{2}$. It can be relaxed to $\binom{n/d+2}{2}^d$

because $\sum_i^d c_i = n$ and it is maximized when c_i are equal. For simplicity, it can be further relaxed to $(\frac{n}{d}+1)^{2d}$. □

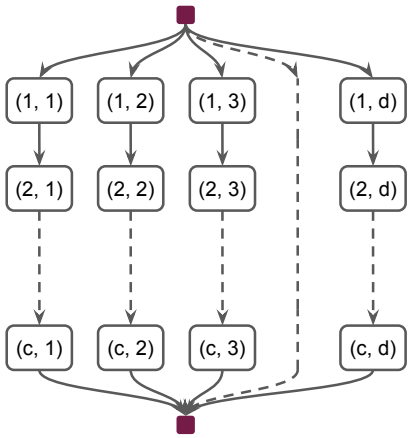


Figure 3.5: The example to make the time complexity $\mathcal{O}(\binom{n/d+2}{2}^d)$ tight. The time complexity for this graph is $\mathcal{O}(\binom{c+2}{2}^d)$

The computation graph shown in Figure 3.5 is an example to demonstrate that the time complexity of $\mathcal{O}(\binom{n/d+2}{2}^d)$ can be reached.

In this example, there are d independent paths and each path has c operators. Because the paths are independent with each other and there is no edge between two different paths, we can get the upper bound $\mathcal{O}(\binom{c+2}{2}^d)$ by the analysis in above time complexity proof.

3.4 Reduce the Search Time by Schedule Pruning

It is difficult for a dynamic programming algorithm to stop early, because it gets the best result at the very end. To reduce the search time, IOS introduces *schedule pruning* to reduce the exploration space by restricting the max number of groups and the max number of operators within a group. We define the pruning strategy P as a boolean function of S and S'. We only enumerate the ending S' of S that satisfies the pruning strategy P, that is, $P(S, S') = \text{True}$ (L16 of Algorithm 1). The pruning strategy consists of two parameters r and s: $P(S, S') = \text{True}$ if and only if ending S' has at most s groups and each group has at most r operators.

After applying the pruning strategy P, the time complexity is reduced from $\mathcal{O}((\frac{n}{d}+1)^{2d})$ to $\mathcal{O}((\frac{n}{d}+1)^d(r+1)^s)$. Of course, there is a trade-off between the search cost and the quality of the discovered schedule. We evaluate this trade-off in Section 4.6.1.

Chapter 4

Experiments

4.1 Implementation Setup

IOS is a framework-agnostic algorithm and can be implemented in popular frameworks. We implement the dynamic programming scheduling algorithm in Python and the execution engine in C++. The latency of a stage is directly measured in the execution engine to guide the scheduling. The execution engine is based on vendor-provided library cuDNN [8] and supports operators' parallel execution. To concurrently execute multiple groups of operators, IOS puts different groups into different CUDA streams. Kernels in different CUDA streams will be executed in parallel if there are enough computation resources. Throughout the experiments, we use cuDNN 7.6.5, cuda 10.2, NVIDIA driver 450.51.05, and adopt TensorRT 7.0.0.11 and TVM 0.7 as baseline libraries.

Networks	#Blocks	#Operators	Operator Type
Inception V3	11	119	Conv-Relu
Randwire	3	120	Relu-SepConv
NasNet	13	374	Relu-SepConv
SqueezeNet	10	50	Conv-Relu

Table 4.1: The CNN benchmarks. Number of blocks, number of operators and the main operator type for each network are listed in the table. Here "Conv-Relu" means a convolution followed by a ReLU activation and "Relu-SepConv" means ReLU activation followed by seperatble convolution.

We benchmark four modern CNNs in the experiment: Inception V3 [48], RandWire [52], NasNet-A [57] and SqueezeNet [19]. Table 4.1 shows the number of blocks, the number of operators, and the main operator type for each network. IOS supports the user-defined schedule unit. In this experiment, we take the operator type shown in the table, besides other operators such as Concat, as the basic schedule unit. Some models (e.g., ResNet [17]) might have limited inter-operator parallelization opportunities. For example, for ResNet-50 and ResNet-34, we can only achieve 2% to 5% speedup by paralleling the downsample convolutions. We do not consider it as our benchmarked model in the rest of the evaluation.

We conduct each experiment 5 times and report the average performance. We adopt the schedule pruning strategy with $r = 3$ and $s = 8$ and conduct each experiment on NVIDIA Tesla V100 unless otherwise stated. The IOS optimization cost for Inception V3 and SqueezeNet is less than 1 minute and the IOS optimization cost for Randwire and NasNet is within 90 minutes.

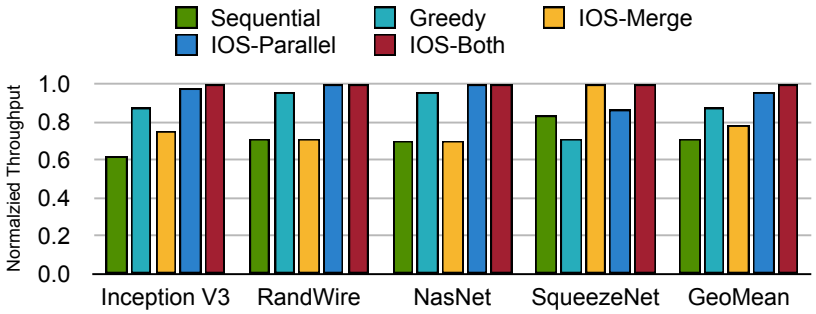


Figure 4.1: End-to-end performance comparison of different schedules across different CNNs on batch size one. The throughput is normalized to the best one for each model.

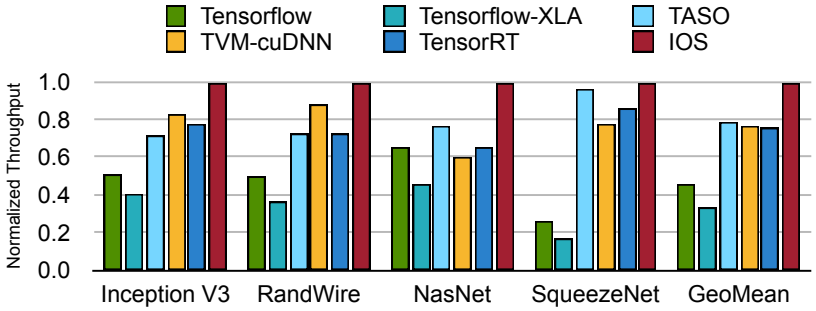


Figure 4.2: End-to-end performance comparison of different frameworks across different CNNs on batch size one. The throughput is normalized to the best one for each model.

4.2 Comparison of Different Schedules

We first compare the inference performance among different schedules with batch size one. We compare five schedules: sequential schedule, greedy schedule, IOS-Merge schedule, IOS-Parallel schedule, and IOS-Both schedule. The sequential schedule executes the operator one-by-one according to certain topological ordering. The greedy schedule puts all the operators that can be executed currently in one stage, and repeats this process until all operators have been scheduled. IOS-Merge, IOS-Parallel, and IOS-Both schedules use the proposed approach to find the schedule but take different parallelization strategies. IOS-Merge only takes the "operator merge" strategy. IOS-Parallel only takes the "concurrent execution" strategy. IOS-Both considers both parallelization strategies. All schedules are executed on IOS execute engine for a fair comparison.

Figure 4.1 shows that IOS-Both outperforms all the other four schedules. The greedy schedule gets good results on RandWire and NasNet. However, it degrades the performance of SqueezeNet because of the overhead of synchronization. Because we can not merge "Relu-SepConv" operators in RandWire and NasNet, IOS-Merge gets the same schedule as Sequential, and IOS-Both gets the same schedule as IOS-Parallel. IOS-Both considers two parallelization strategies and outperforms all the other four schedules. In later experiments, "IOS" refers to "IOS-Both" by default.

4.3 Comparison of cuDNN-based Frameworks

For popular frameworks, there are two ways to exploit the intra-operator parallelism. Frameworks such as Tensorflow [31], TASO [22], and TensorRT [35] use the vendor-provided library cuDNN.

Frameworks such as TVM [6] and Ansor [55] search the tensor program schedule for each kernel. TVM also supports to call external libraries such as cuDNN to implement some kernels (e.g., convolution). In this subsection, we compare the performance of cuDNN-based frameworks with batch size one. Larger batch size is studied in the ablation study section.

There are five baselines: Tensorflow, Tensorflow-XLA, TASO, TVM-cuDNN, and TensorRT. Tensorflow-XLA is the tensorflow framework with XLA optimization turning on. TVM-cuDNN is the TVM framework that compiles a convolution neural network with cuDNN library, which would use the convolution kernel provided by cuDNN to execute convolutions. All other operators such as addition and concatenation would use their own kernels. For fair comparison, we only compare cuDNN-based libraries here. The comparison between TVM-AutoTune and IOS can be found in the ablation study section. Figure 4.2 shows that IOS consistently outperforms all five baseline frameworks on four benchmark CNNs. IOS can achieve 1.1 to 1.5× speedup comparing to the state of the art library TASO, TVM-cuDNN, and TensorRT.

4.4 More Active Warps Improve Utilization

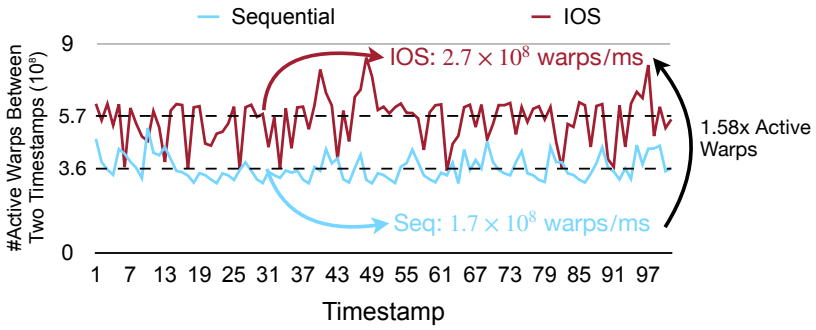


Figure 4.3: Active Warps for sequential schedule and IOS schedule. We use the model in Figure 1.2 in this experiment.

As introduced in Section 2.1, model operators are mapped to GPU kernels to execute. A kernel invokes a collection of threads that are grouped into multiple thread blocks. Thread blocks are distributed to stream multiprocessors (SMs). Each thread block on a SM is further partitioned into multiple warps. A warp, as a basic execution unit, contains a fixed number of threads (e.g., 32 for NVIDIA GPU) to execute in a Single Instruction Multiple Thread (SIMT) fashion.

A warp is considered active from the time it is scheduled on an SM until it completes the last instruction. SM can hide the warps stall caused by memory accesses through fast context switching: at every cycle, each warp scheduler will pick an eligible warp and issue instructions. If no eligible warp is available for a warp scheduler, the no instruction will be issued from this warp scheduler and a cycle is wasted. Increasing the number of active warps is an effective approach to increase the likelihood of having eligible warps to execute at each cycle. Thus, it is *crucial* to increase the number of active warps. Figure 4.3 shows the number of active warps on the whole GPU throughout the repeated execution of both the IOS and the Sequential schedule, sampled using NVIDIA's CUPTI profiling toolset every 2.1 ms. IOS schedule achieves 58% more active warps on average compared to the Sequential schedule. This explains the reason for IOS overall performance speedup.

4.5 Consistent Speedup on Other Devices

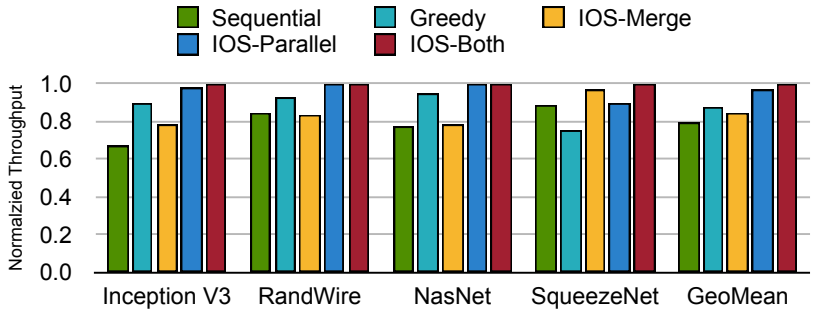


Figure 4.4: End-to-end performance comparison of different schedules across different CNNs on batch size one. The throughput is normalized to the best one for each model. This experiment is conducted on NVIDIA RTX 2080Ti.

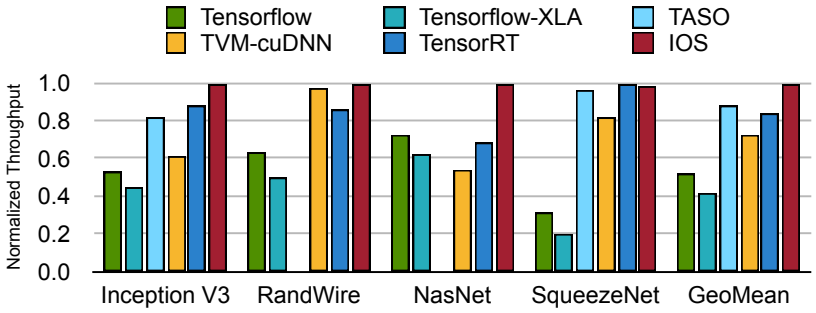


Figure 4.5: End-to-end performance comparison of different frameworks across different CNNs on batch size one. The throughput is normalized to the best one for each model. This experiment is conducted on NVIDIA RTX 2080Ti.

In addition to results on NVIDIA Tesla V100 (Volta architecture), we also conduct experiments on NVIDIA RTX 2080Ti (Turing architecture) to show that our optimization is generally effective across different GPU architectures. We use the same models and baselines for comparisons as in Section 4.2 and Section 4.3.

Figure 4.4 shows that IOS with two parallelization strategies (i.e., IOS-Both) outperforms all other schedules. In particular, IOS-Both achieves 1.1× to 1.5× speedup comparing to the sequential schedule. Figure 4.5 shows that IOS outperforms all other cuDNN-based frameworks[1] on Inception V3, RandWire, and NasNet. IOS achieves comparable performance with TASO and TensorRT on SquuezeNet. These results align with the results on V100.

4.6 Ablation Study

4.6.1 Schedule Pruning Reduces Search Time

To explore the trade-off between optimized latency and optimization cost (i.e. search time), we experiment Inception V3 and NasNet with pruning strategy parameters $r = \{1, 2, 3\}$ and $s = \{3, 8\}$.

[1]TASO runs out of GPU memory on NVIDIA 2080Ti for RandWire and NasNet.

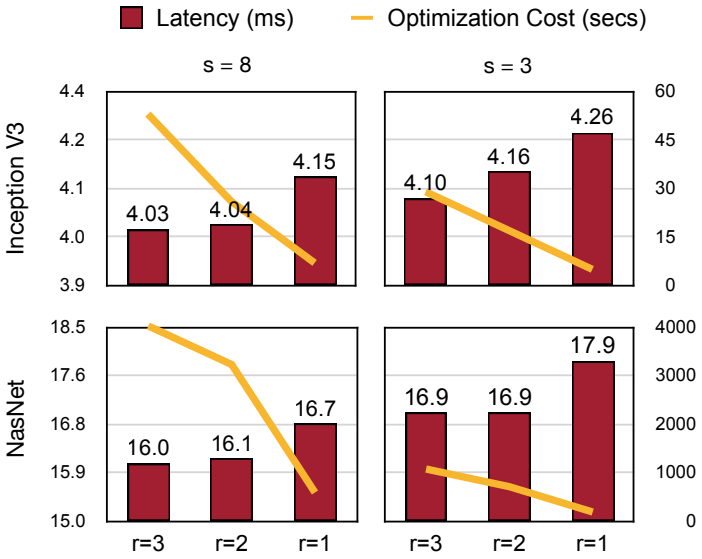


Figure 4.6: Trade-off between the optimized latency and the optimization cost for Inception V3 and NasNet. Two pruning strategy parameters r and s are used to prune the schedule space. r limits the maximum number of operators in each group while s limits the maximum number of groups in a stage. The left axis shows the optimized latency, and the right axis shows the optimization cost.

As shown in Figure 4.6, when s and r get smaller, the optimization cost decreases at the cost of larger network latency. This is because smaller s and r restrict the schedules that IOS explores, thus reduce the optimization cost and increase schedule latency. By setting $r = 1$ and $s = 8$, IOS still achieves 1.59× and 1.37× speedup for Inception V3 and NasNet, comparing to sequential schedule. Meanwhile, the optimization cost for each network is within 30 seconds and 18 minutes, respectively.

4.6.2 Specialized Scheduling is Beneficial

Specialization for Different Batch Sizes		Optimized for		
		1	32	128
Execute on	1	**4.03**	4.50	4.63
	32	29.21	**27.44**	27.93
	128	105.98	103.74	**103.29**

(1) Specialization for Batch Sizes

Specialization for Different Devices		Optimized for	
		K80	V100
Execute on	K80	**13.87**	14.65
	V100	4.49	**4.03**

(2) Specialization for Devices

Table 4.2: Latency (ms) of specialized schedules for batch size 1, 32 and 128, and specialized schedules for NVIDIA Tesla K80 and V100. The best performance is achieved when the schedule is specialized for each batch size and device. Each row is the batch size or device that the model is executed on. Each column is the batch size or device that IOS optimized for. InceptionV3 is used as a benchmark.

Different workloads (e.g. network with different batch sizes) have different computation features; thus it is necessary to specialize the schedule for different workloads. We optimize Inception V3 with batch size 1, 32 and 128. Then we execute the network with these schedules on batch size 1, 32 and 128 separately. In Table 4.2 (1), the numbers in a row represents the latency executed with the same batch size but using schedules optimized for different batch sizes. The specialized schedule for

each batch size achieved the best result. To explore the specialization for devices, we also optimize the network on both NVIDIA Tesla K80 and V100 with batch size one. Table 4.2 (2) shows that the specialized schedule for each device also achieved better results.

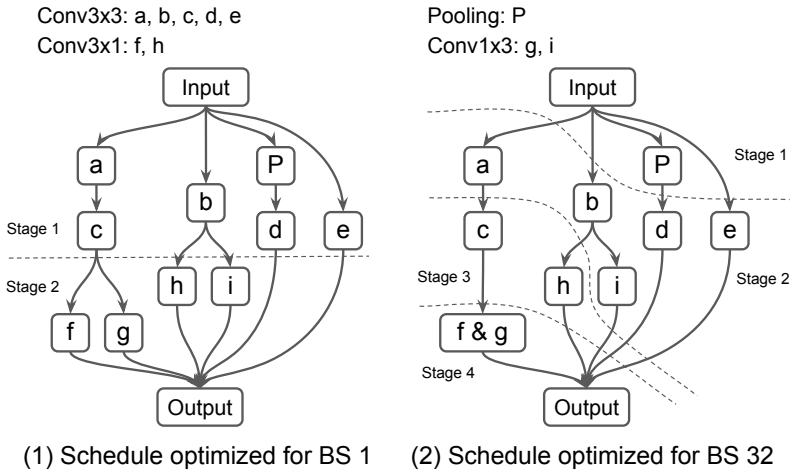


Figure 4.7: The schedule found by IOS for the last block of Inception V3. Operator a-e are convolution operator while operator P is the pooling operator. Schedule (1) and (2) are optimized for batch size 1 and 32, respectively. There are two stages in schedule (1) while there are 4 stages in schedule (2). Schedule (1) is 28% faster than schedule (2) on batch size 1. Schedule (2) is 8% faster than schedule (1) on batch size 32.

IOS discovers different schedules for different batch sizes. For example, Figure 4.7 shows the schedule of the last block of Inception V3 optimized for batch size 1 and 32, respectively. There are two stages in the schedule (1), which is optimized for batch size 1 while there are four stages in the schedule (2), which is optimized for batch size 32. The schedule (1) is 28% faster than the schedule (2) on batch size 1, while the schedule (2) is 8% faster than (1) on batch size 32. There are two differences between them. The first one is that convolution f and g in the schedule (2) are merged into a single convolution. This is because activation (the output tensor of an operator) is the memory bottleneck at large batch size. It is more crucial to reduce memory access, even at the cost of larger computation cost. Merging can reduce the memory access, because the merged kernel only access the output of convolution c once, instead of twice in the schedule (1). However, because the kernel size of f and g are 3x1 and 1x3, respectively, their kernel size would be expanded to 3x3 by padding zeros, which increases the amount of computation. Another difference between the schedule (1) and (2) is that the schedule (2) has more stages than the schedule (1). We found a similar phenomenon for large batch sizes because of resource contention. When multiple operators are executed on the device, there is a conflict over access to the shared resources such as the last-level cache, making the concurrent execution degrades the performance. This gets more severe for larger batch sizes because the demand for shared resources gets larger.

4.6.3 Blockwise Speedup in Inception V3

To explore the speedup for different blocks, we compare the performance of each block of Inception-V3 [48] between sequential and IOS schedule (Figure 4.8). IOS consistently runs faster than the sequential schedule. The speedup for the individual block is up to 2.3×, and the end-to-end speedup

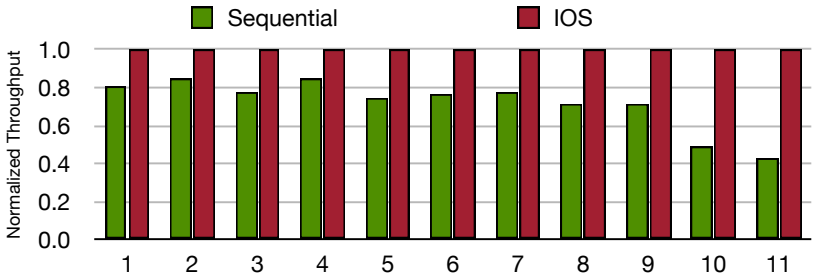


Figure 4.8: IOS consistently outperforms sequential executions on each block of Inception-v3.

is 1.6×. More speedup is achieved for back blocks because the width gets larger and more inter-parallelism is possible.

4.6.4 Consistent Improvement for Different Batch Sizes

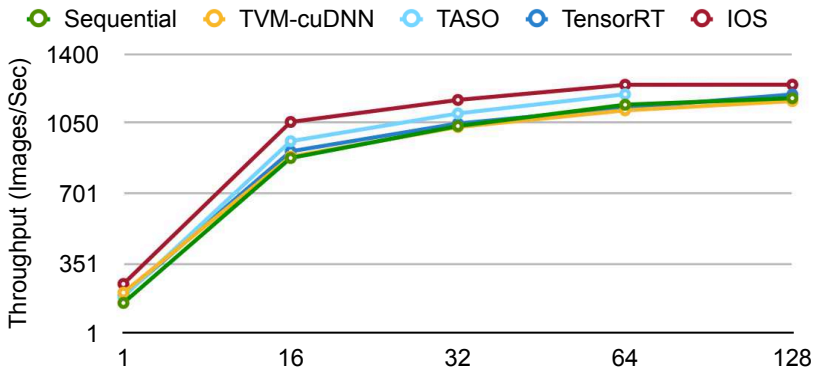


Figure 4.9: The throughput comparison of Sequential schedule, TVM-cuDNN, TASO, TensorRT and IOS on batch size 1 to 128 for Inception V3. TASO runs out of memory with batch size 128.

In real-world applications, we need to handle different batch sizes for inference. For example, for real-time applications on edge devices, we usually use a batch size of one to reduce latency. In contrast, in cloud settings, the larger batch size is preferred to increase throughput. Changing the workload requires different inter-operator parallelization schedules. We optimize Inception V3 with the batch sizes of 1, 16, 32, 64, 128, and compare the throughput. Figure 4.9 shows that the throughput increases with the batch size. When the batch size is larger than 128, the performance saturates, and the throughput does not increase significantly anymore. The throughput of IOS outperforms all the baselines consistently on all batch sizes. Even though a larger batch size provides more data parallelism, we can still utilize inter-operator parallelism to further improve the throughput.

4.6.5 Intra- and Inter-Operator Parallelism

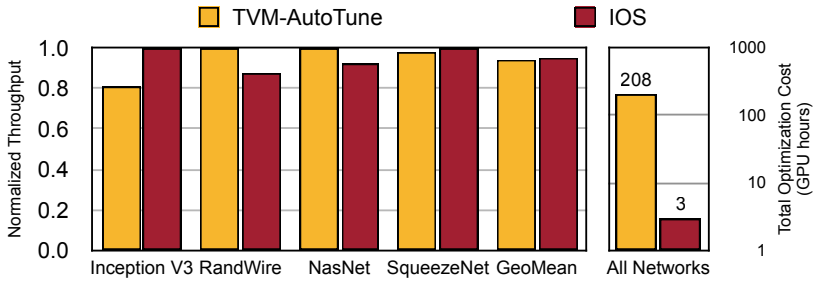


Figure 4.10: End-to-end performance comparison between TVM-AutoTune and IOS. TVM-AutoTune and IOS are *orthogonal* because TVM focuses on the intra-operator parallelism while IOS focuses on inter-operator parallelism. They can be combined to boost the inference performance further. The optimization cost of IOS is two orders of magnitude less than TVM.

TVM exploits the intra-operator parallelism by searching the schedule for each kernel on a specific device. IOS focuses on inter-operator parallelism and leaves the exploitation of intra-operator parallelism to cuDNN library. Although intra- and inter-operator parallelism is *orthogonal* and can be combined, we compare TVM and IOS here to give some insight into each parallelism's benefit. As shown in Figure 4.10, TVM takes 208 GPU hours while IOS only takes 3 GPU hours to optimize the four networks. IOS outperforms TVM on Inception V3 and SqueezeNet. This is because only utilizing intra-parallelism can not provide enough parallelism for the powerful computing device. Meanwhile, TVM outperforms IOS on Randwire and NasNet, because TVM finds more efficient kernels for separable convolutions, which occupy the majority of operators in Randwire and NasNet. We believe the combination of TVM and IOS would boost the performance further and leave this for future work.

Chapter 5

Related Work

Graph transformation. MetaFlow [21] performs functional-preserving graph transformations to optimize DNN architectures. Merging operators with the same input enables more parallelism (a larger operator compared to two small sequential operators) and reduces accesses to GPU memories. TASO [22] further introduces an automated generation of substitution rules and it explores more mathematically equivalent DNN architectures of the input one comparing to MetaFlow. MetaFlow and TASO consider the whole computation graph and search for highly optimized substitution strategies. However, the inter-oprator parallelism utilized by MetaFlow and TASO is still limited as only the same type of operators can be merged.

To address the large schedule space problem, IOS utilizes dynamic programming to take advantage of the common sub-schedules among different schedules. Also, IOS supports concurrent execution of different types of operators, addressing the limitation of MetaFlow and TASO.

CNN Design. Several lightweight design primitives have been recently introduced to improve the efficiency of CNNs. Examples include SequeezeNet [19], MobileNet [41] and ShuffletNet [54]. However, such design patterns cannot fully utilize the hardware. Hardware under-utilization becomes more severe as accelerators are getting more powerful (shown in Figure 1.1). On the other hand, multi-branch CNNs become a trend in model architecture design, including both manually designed networks [47, 19, 48] and the networks discovered by neural architecture search [5, 57]. With a fixed computation budget, multi-branch CNNs use more small convolution primitives, which further amplifies the resource under-utilization problem on modern hardware.

Intra-operator Parallelism. Current deep learning frameworks (e.g., TensorFlow and PyTorch) generally focus on intra-operator parallelism, which executes arithmetic operations within a *single* operator in parallel (e.g., tiled matrix multiplication). Tensorflow and PyTorch are built upon vendor-provided libraries (e.g., cuDNN), a set of DNN compute primitives heavily optimized by vendor engineers to achieve near-peak machine performance. However, these DNN operators are executed sequentially on a hardware device. The degree of parallelism within an operator is limited; thus, intra-operator parallelism cannot provide sufficient parallelizable computation to feed powerful hardware devices. As a result, the hardware is often under-utilized using these frameworks.

Different from manual performance tuning, Auto-Halide [34], TVM [6] and Ansor [55] exploit intra-parallelism through automatically *learning* efficient schedule for individual DNN kernels. This automation saves a large amount of engineering effort and can generate more efficient DNN kernels

than the manually designed counterparts. However, still, all these libraries only focus on intra-operator parallelism but do not exploit inter-operator parallelism.

Inter-Operator Scheduling. Recent work has explored inter-operator scheduling. Tang et al. [50] proposes a greedy heuristic approach, Graphi, that executes all available CNN operators whenever possible to saturate CPU's computation capability. The greedy strategy does not *holistically* optimize the computation graph's performance, hence yields unbalanced and sub-optimal schedules. Rammer[30] optimizes the execution of DNN workloads by holistically exploiting parallelism through inter- and intra- operator co-scheduling, enabling a richer scheduling space for executing a DNN model. IOS focuses on the inter-operator scheduling and leaves the intra-operator scheduling to the hardware. Nimble[27] is a DNN engine that supports parallel execution of DNN operators on GPU and minimizes the scheduling overhead using ahead-of-time (AOT) scheduling. The scheduling algorithm used in Nimble does not consider the latency of each operator, while IOS is a profile-based scheduler.

Chapter 6

Future Work

In this work, we proposed IOS to explore the scheduling of inter-operator parallelization of a multi-branch model. There are still some directions we can explore based on IOS:

1. Parallelizing multiple models on the same device. Currently, we only studied the parallelization of a single model on a single device. However, there are still cases that we need to run multiple different models on the same device. How to coordinates the parallelization of operators from different models remain an open question.

2. Optimizing the model with a single branch of operators. If the given model has no inter-operator parallelization opportunity, there is nothing IOS can do to improve the performance. How to increase the device-utilization of single-branched small network is still a very challenging problem.

3. Intra- and inter-operator co-scheduling. Existing kernels from vender libraries [8, 35] and tensor program compilation [6, 7, 55] assume that each kernel occupies the whole underlying device (e.g., GPU). However, when we do inter-operator parallelization, this assumption is no longer valid. Thus, how to search the intra- and inter-operator schedules is still a challenging problem.

4. Expanding IOS on other workloads. In this work, we focus on optimizing CNN models. However, IOS is a very general algorithm and is also applicable to other models with inter-operator parallelization opportunity.

5. Explore schedule algorithm on other schedule space. The IOS schedule space contains schedules with multiple dependency-preserving stages. This is not the only schedule space we can consider. Another schedule space that partition the computation graph into different operator streams with dependency on this streams is also deserve consideration. However, we can not directly use IOS to search schedule in this schedule space because the optimal substructure property no longer holds for schedules in this schedule space.

We leave the exploration in the future work.

Chapter 7

Conclusion

In this work[1], we observe that the sequential execution of CNNs no longer provides sufficient parallelization opportunities to fully utilize all the computation resources, especially for the single-batch inference on a power GPU. In machine learning community, the number of float operators (FLOPs) instead of the latency or throughput on actual device is often used as the metric of execution performance because the later metrics might vary from different devices. Thus, we can easily find a network [49, 29] that contains a lot of small operators because multi-branched structure with more small operators can have a better accuracy performance. But the same FLOPs might not indicate the same latency or throughput because whether we can map the workloads to underlying hardware is also a key factor to influence the execution efficiency. A network with multi-branched small operators usually under-utilizes the hardware when we execute each operator one by one.

To address the under-utilization problem, we propose IOS that combines intra- and inter-operator parallelism. Inter-operator parallelism provides more parallelization opportunity than only utilizing intra-operator parallelism. We adopt dynamic programming to find an efficient schedule that better utilizes the hardware. We have proved that the time complexity of our algorithm is only exponential in the width of computation graph (i.e., the maximum number of parallelizable operators), which is usually small for CNNs. A pruning technique is also introduced to reduce the search time.

We did extensive experiments to show the efficacy of IOS. We compared IOS with existing state-of-the-art libraries (e.g., TensorRT) and showed that IOS achieves 1.1 to 1.5× speedup on modern CNNs. We also compared different schedules and profiled the active warps of sequential schedule and IOS schedule to show that IOS finds highly optimized schedule.

[1] This thesis is a based on our published conference paper [14] at MLSys 2021.

Bibliography

[1] Martín Abadi et al. "Tensorflow: A system for large-scale machine learning". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 265–283.

[2] Thomas L Adam, K. Mani Chandy, and JR Dickson. "A comparison of list schedules for parallel processing systems". In: *Communications of the ACM* 17.12 (1974), pp. 685–690.

[3] Abien Fred Agarap. "Deep learning using rectified linear units (relu)". In: *arXiv preprint arXiv:1803.08375* (2018).

[4] Han Cai, Ligeng Zhu, and Song Han. "ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware". In: *International Conference on Learning Representations*. 2019. URL: https://arxiv.org/pdf/1812.00332.pdf.

[5] Han Cai et al. "Path-Level Network Transformation for Efficient Architecture Search". In: *ICML*. 2018.

[6] 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 ({OSDI} 18)*. 2018, pp. 578–594.

[7] Tianqi Chen et al. "Learning to optimize tensor programs". In: *Advances in Neural Information Processing Systems*. 2018, pp. 3389–3400.

[8] Sharan Chetlur et al. "cudnn: Efficient primitives for deep learning". In: *arXiv preprint arXiv: 1410.0759* (2014).

[9] Edward G Coffman and Ronald L Graham. "Optimal scheduling for two-processor systems". In: *Acta informatica* 1.3 (1972), pp. 200–213.

[10] Alain Darte, Yves Robert, and Frederic Vivien. *Scheduling and automatic parallelization*. Springer Science & Business Media, 2000.

[11] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[12] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[13] R. P. Dilworth. "A Decomposition Theorem for Partially Ordered Sets". In: *Annals of Mathematics* 51.1 (1950), pp. 161–166. ISSN: 0003486X. URL: http://www.jstor.org/stable/1969503.

[14] Yaoyao Ding et al. “IOS: Inter-Operator Scheduler for CNN Acceleration”. In: *Proceedings of Machine Learning and Systems*. Vol. 3. 2021, pp. 167–180.

[15] Apostolos Gerasoulis and Tao Yang. *A comparison of clustering heuristics for scheduling DAGs on multiprocessors*. Rutgers University, Department of Computer Science, Laboratory for Computer, 1991.

[16] Song Han, Huizi Mao, and William J Dally. “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding”. In: *arXiv preprint arXiv:1510.00149* (2015).

[17] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[18] Te C Hu. “Parallel sequencing and assembly line problems”. In: *Operations research* 9.6 (1961), pp. 841–848.

[19] Forrest N Iandola et al. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and 0.5 MB model size”. In: *arXiv preprint arXiv:1602.07360* (2016).

[20] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[21] Zhihao Jia et al. “Optimizing DNN Computation with Relaxed Graph Substitutions”. In: *Proceedings of Machine Learning and Systems*. Ed. by A. Talwalkar, V. Smith, and M. Zaharia. Vol. 1. 2019, pp. 27–39.

[22] 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*. SOSP ’19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 47–62. ISBN: 9781450368735. DOI: 10.1145/3341301.3359630. URL: https://doi.org/10.1145/3341301.3359630.

[23] Hironori Kasahara and Seinosuke Narita. “Practical multiprocessor scheduling algorithms for efficient parallel processing”. In: *IEEE Transactions on computers* 33.11 (1984), pp. 1023–1029.

[24] AA Khan, Carolyn L McCreary, and Mary S Jones. “A comparison of multiprocessor scheduling heuristics”. In: *1994 Internatonal Conference on Parallel Processing Vol. 2*. Vol. 2. IEEE. 1994, pp. 243–250.

[25] SJ Kim. “A general approach to mapping of parallel computations upon multiprocessor architectures”. In: *Proc. International Conference on Parallel Processing*. Vol. 3. IEEE Computer Society. 1988.

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

[27] Woosuk Kwon et al. “Nimble: Lightweight and Parallel GPU Task Scheduling for Deep Learning”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 8343–8354. URL: https://proceedings.neurips.cc/paper/2020/file/5f0ad4db43d8723d18169b2e4817a160-Paper.pdf.

[28] Kuo-Bin Li. “ClustalW-MPI: ClustalW analysis using distributed and parallel computing”. In: *Bioinformatics* 19.12 (2003), pp. 1585–1586.

[29] Hanxiao Liu, Karen Simonyan, and Yiming Yang. “DARTS: Differentiable Architecture Search”. In: *arXiv preprint arXiv:1806.09055* (2018).

[30] Lingxiao Ma et al. “Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 881–897. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/ma.

[31] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[32] Paulius Micikevicius. “GPU performance analysis and optimization”. In: *GPU technology conference*. Vol. 3. 2012.

[33] Volodymyr Mnih et al. “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602* (2013).

[34] Ravi Teja Mullapudi et al. “Automatically scheduling halide image processing pipelines”. In: *ACM Transactions on Graphics (TOG)* 35.4 (2016), pp. 1–11.

[35] NVIDIA. “NVIDIA TensorRT: Programmable Inference Accelerator”. In: (). URL: https://developer.nvidia.com/tensorrt.

[36] NVIDIA Corporation. *NVIDIA CUDA C++ Programming Guide*. Version 11.6. 2022.

[37] NVIDIA Corporation. *NVIDIA’s Next Generation CUDA Compute Architecture: Fermi*. Version 1.1. 2009.

[38] NVIDIA Corporation. *NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110/210*. Version 1.1. 2014.

[39] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. “On the difficulty of training recurrent neural networks”. In: *International conference on machine learning*. PMLR. 2013, pp. 1310–1318.

[40] Adam Paszke et al. “Automatic differentiation in PyTorch”. In: *NIPS-W*. 2017.

[41] Mark Sandler et al. “Mobilenetv2: Inverted residuals and linear bottlenecks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 4510–4520.

[42] Vivek Sarkar. “Partitioning and scheduling parallel programs for execution on multiprocessors”. PhD thesis. Stanford University, 1987.

[43] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *nature* 529.7587 (2016), p. 484.

[44] Oliver Sinnen. *Task scheduling for parallel systems*. Vol. 60. John Wiley & Sons, 2007.

[45] Oliver Sinnen and Leonel Sousa. “List scheduling: extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures”. In: *Parallel Computing* 30.1 (2004), pp. 81–101.

[46] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to sequence learning with neural networks”. In: *Advances in neural information processing systems*. 2014, pp. 3104–3112.

[47] Christian Szegedy et al. “Going deeper with convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[48] Christian Szegedy et al. “Rethinking the inception architecture for computer vision”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2818–2826.

[49] Mingxing Tan et al. “Mnasnet: Platform-aware neural architecture search for mobile”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 2820–2828.

[50] Linpeng Tang et al. “Scheduling computation graphs of deep learning models on manycore CPUs”. In: *arXiv preprint arXiv:1807.09667* (2018).

[51] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. “Performance-effective and low-complexity task scheduling for heterogeneous computing”. In: *IEEE transactions on parallel and distributed systems* 13.3 (2002), pp. 260–274.

[52] Saining Xie et al. “Exploring randomly wired neural networks for image recognition”. In: *arXiv preprint arXiv:1904.01569* (2019).

[53] Tao Yang and Apostolos Gerasoulis. “List scheduling with and without communication delays”. In: *Parallel Computing* 19.12 (1993), pp. 1321–1344.

[54] Xiangyu Zhang et al. “Shufflenet: An extremely efficient convolutional neural network for mobile devices”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 6848–6856.

[55] Lianmin Zheng et al. “Ansor: Generating High-Performance Tensor Programs for Deep Learning”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. Banff, Alberta: USENIX Association, Nov. 2020. URL: `https://www.usenix.org/conference/osdi20/presentation/zheng`.

[56] Barret Zoph and Quoc V Le. “Neural architecture search with reinforcement learning”. In: *arXiv preprint arXiv:1611.01578* (2016).

[57] Barret Zoph et al. “Learning transferable architectures for scalable image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 8697–8710.