Ansor: Generating High-Performance Tensor Programs for Deep Learning

Lianmin Zheng [1], Chengfan Jia [2], Minmin Sun [2], Zhao Wu [2], Cody Hao Yu [3] ,
Ameer Haj-Ali [1], Yida Wang [3], Jun Yang [2], Danyang Zhuo [1,4] ,
Koushik Sen [1], Joseph E. Gonzalez [1], Ion Stoica [1]

[1] *UC Berkeley*, [2]*Alibaba*, [3]*Amazon Web Services*, [4] *Duke University*

arXiv:2006.06762v2 [cs.LG] 15 Jun 2020

Abstract

High-performance tensor programs are crucial to guarantee efficient execution of deep learning models. However, obtaining performant tensor programs for different operators on various hardware platforms is notoriously difficult. Currently, deep learning systems rely on vendor-provided kernel libraries or various search strategies to get performant tensor programs. These approaches either require significant engineering efforts in developing platform-specific optimization code or fall short in finding high-performance programs due to restricted search space and ineffective exploration strategy.

We present Ansor, a tensor program generation framework for deep learning applications. Compared with existing search strategies, Ansor explores much more optimization combinations by sampling programs from a hierarchical representation of the search space. Ansor then fine-tunes the sampled programs with evolutionary search and a learned cost model to identify the best programs. Ansor can find high-performance programs that are outside the search space of existing state-of-the-art approaches. Besides, Ansor utilizes a scheduler to simultaneously optimize multiple subgraphs in a set of deep neural networks. Our evaluation shows that Ansor improves the execution performance of deep neural networks on the Intel CPU, ARM CPU, and NVIDIA GPU by up to 3.8×, 2.6×, and 1.7×, respectively.

1 Introduction

Low latency execution of deep neural networks (DNN) plays a critical role in autonomous driving [13], augmented reality [3], language translation [14], and other applications of AI. DNNs can be expressed as a directed acyclic computational graph (DAG), in which nodes represent the operators (*e.g.*, convolution and matrix multiplication) and directed edges represent the dependencies between operators. Existing deep learning frameworks (*e.g.*, Tensorflow [1], PyTorch [36], MXNet [9]) map the operators in DNNs to vendor-provided kernel libraries (*e.g.*, CuDNN [12], MKL-DNN [24]) to achieve high performance. However, these kernel libraries require significant engineering efforts in manual tuning towards each hardware platform and operator. The significant manual efforts required to produce efficient operator implementations for each target accelerator limit the development and innovation of new operators [6] and specialized accelerators [32].

Given the importance of DNNs' performance, researchers and industry practitioners have turned to search-based compilation [2, 10, 29, 45, 53] for automated generation of *tensor programs*, *i.e.* low-level implementations of tensor operators. For an operator or a (sub-)graph of multiple operators, users define the computation in a high-level declarative language (§2), and the compiler then searches for programs tailored towards different hardware platforms.

To find performant tensor programs, it is necessary for a search-based approach to explore a large enough search space to cover all the useful tensor program optimizations. However, existing approaches fail to capture many effective optimization combinations, because they rely on either predefined manually-written templates (*e.g.*, TVM [11], FlexTensor [53]) or aggressive pruning by inaccurately evaluating incomplete programs (*e.g.* Halide auto-scheduler [2]), which prevents them from covering a large enough search space (§2). The rules they use to construct the search space are also limited.

In this paper, we want to explore a novel search strategy for generating high-performance tensor programs. It can automatically generates a large search space with comprehensive coverage of optimizations and gives every tensor program in the space a chance to be chosen. It thus enables us to find high-performance programs that existing approaches miss.

Realizing this goal faces multiple challenges. First, it requires us to automatically construct a large search space to cover as many tensor programs as possible for a given computation definition. Second, we need to search efficiently without comparing incomplete programs in the large search space that can be orders of magnitude larger than what existing templates can cover. Finally, when optimizing an entire DNN with many subgraphs, we should recognize and prioritize the subgraphs that are critical to the end-to-end performance.

To this end, we design and implement *Ansor*, a framework for automated tensor program generation. Ansor utilizes a hierarchical representation to cover a large search space. This representation decouples high-level structures and low-level details, enabling flexible enumeration of high-level structures and efficient sampling of low-level details. The space is constructed automatically for a given computation definition. Ansor then samples complete programs from the search space and fine-tunes these programs with evolutionary search and a learned cost model. To optimize the performance of DNNs with multiple subgraphs, Ansor dynamically prioritizes subgraphs of the DNNs that are more likely to improve the end-to-end performance.

We evaluate Ansor on both standard deep learning benchmarks and emerging new workloads against manual libraries and state-of-the-art search frameworks. Experiment results show that Ansor improves the execution performance of DNNs on the Intel CPU, ARM CPU, and NVIDIA GPU by up to 3.8×, 2.6×, and 1.7×, respectively. For most computation definitions, the best program found by Ansor is outside the search space of existing search-based approaches. The results also show that, compared with existing search-based approaches, Ansor has a more efficient search algorithm, generating higher-performance programs in a shorter time, despite its larger search space. Besides, Ansor enables automatic extension to new operators without requiring manual templates.

In summary, this paper makes the following contributions:

- A mechanism to generate a large search space with a hierarchical representation, from which diverse tensor programs can be sampled.
- An evolutionary strategy with a learned cost model to fine-tune the performance of tensor programs.
- A scheduling algorithm based on gradient descent to prioritize important subgraphs when optimizing the end-to-end performance of one or more DNNs.
- An implementation and comprehensive evaluation of the Ansor system demonstrating that the above techniques outperform state-of-the-art systems on a variety of deep learning models and hardware platforms.

2 Background

The deep learning ecosystem is embracing a rapidly growing diversity of hardware platforms including CPUs, GPUs, FPGAs, and ASICs. In order to deploy DNNs on these platforms, high-performance tensor programs are needed for the operators used in DNNs. The required operator set typically contains a mixture of standard operators (*e.g.*, matmul, conv2d) and novel operators invented by machine learning researchers (*e.g.*, capsule conv2d [21], dilated conv2d [52]).

To deliver portable performance of these operators on a wide range of hardware platforms in a productive way, multiple compiler techniques have been introduced (*e.g.*, TVM [10], Halide [38], Tensor Comprehensions [45]). Users define the computation in a form similar to mathematical expressions using a high-level declarative language, and the compiler generates optimized tensor programs according to the definition. Figure 1 shows the computation definition of matrix multiplication in the TVM tensor expression language. Users mainly need to define the shapes of the tensors and how each element in the output tensor is computed.

Matrix Multiplication $C_{i,j} = \sum_k A_{i,k} B_{k,j}$

```
C = compute((N, M), lambda i, j: sum(A[i, k]*B[k, j], [k]))
```

Figure 1: The computation definition of matrix multiplication.

However, automatically generating high-performance tensor programs from a high-level definition is extremely difficult. Depending on the architecture of the targeted platform, the compiler needs to search in an extremely large and complicated search space containing combinatorial choices of optimizations (*e.g.*, tile structure, tile size, vectorization, parallelization). Finding high-performance programs requires the search strategy to cover a large enough space and explore it efficiently. We describe two recent and effective approaches in this section and other related work in §8.

Template-guided search. In template-guided search, the search space is defined by manual templates. As shown in the Figure 2 (a), the compiler (*e.g.*, TVM) requires the user to manually write a template for a computation definition. The template defines the structure of the tensor programs with some tunable parameters (*e.g.*, tile size, and unrolling factor). The compiler then searches for the best values of these parameters for a specific input shape configuration and a specific hardware target. This approach has achieved good performance on common deep learning operators. However, developing templates requires substantial efforts. For example, the code repository of TVM already contains more than 15K lines of code for these templates. This number continues to grow as new operators and new hardware platforms emerge. Besides, constructing a quality template requires expertise in both tensor operators and hardware. It takes non-trivial research efforts [29, 50, 53] to develop quality templates. Despite the huge efforts in developing templates, existing manual templates only cover limited program structures because manually enumerating all optimization choices for all operators is prohibitive.

Sequential construction based search. This approach defines the search space by decomposing the program construction into a fixed sequence of decisions. The compiler then uses an algorithm such as beam search [31] to search for good decisions (*e.g.*, Halide auto-scheduler [2]). In this approach, the compiler constructs a tensor program by sequentially unfolding all nodes in the computational graph. For each node, the compiler makes a few decisions on how to transform it into low-level tensor programs. When all nodes are unfolded, a complete tensor program is constructed. This approach

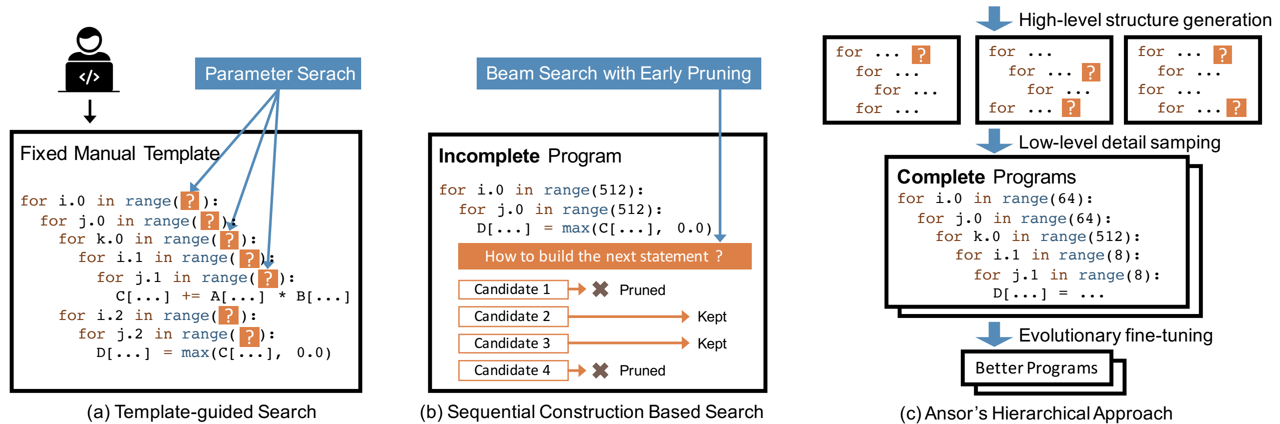


Figure 2: Search strategy comparison. The pseudo-code shows tensor programs with loop nests. The orange question marks denote low-level parameters.

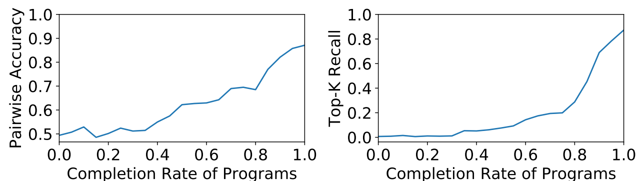


Figure 3: Pairwise comparison accuracy and top-k recall curve on random partial programs. In both diagrams, higher values are better.

uses a set of general unfolding rules for every node, so it can search automatically without requiring manual templates. Because the number of possible choices of each decision is large, to make the sequential process feasible, this approach keeps only top-k candidate programs after every decision. The compiler estimates and compares the performance of candidate programs with a learned cost model to select the top-k candidates; while other candidates are pruned. During the search, the candidate programs are incomplete because only part of the computational graph is unfolded or only some of the decisions are made. Figure 2 (b) shows this process.

However, estimating the final performance of incomplete programs is difficult in several respects: (1) the cost model trained on complete programs cannot accurately predict the final performance of incomplete programs. The cost model can only be trained on complete programs because we need to compile programs and measure their execution time to get the labels for training. Directly using this model to compare the final performance of incomplete programs will result in poor accuracy. As a case study, we train a cost model on 20,000 random complete programs from our search space and use the model to predict the final performance of incomplete programs. The incomplete programs are obtained by masking fractions of the 20,000 complete programs. We use two ranking metrics for evaluation: the accuracy of pairwise comparison and the recall@k score of top-k programs [1]. As shown in Figure 3, the two curves start from 50% and 0% respectively, meaning that random guess with zero information gives 50% pairwise comparison accuracy and 0% top-k recall. The two curves go up quickly as the programs become complete, which means the cost model performs very well for complete programs but fails to accurately predict the final performance of incomplete programs. (2) The fixed order of sequential decisions limits the design of the search space. For example, some optimization needs to add new nodes to the computational graph (*e.g.*, adding cache nodes, using `rfactor` [42]). The number of decisions for different programs becomes different. It is thus hard to align the incomplete programs for a fair comparison. (3) Sequential construction based search is not scalable. Enlarging the search space needs to add more sequential construction steps, but this will result in a worse accumulated error.

Ansor's hierarchical approach As shown in Figure 2(c), Ansor is backed by a hierarchical search space that decouples high-level structures and low-level details. Ansor constructs the space automatically, eliminating the manual efforts in developing templates. Ansor then samples complete programs from the space and performs fine-tuning on complete programs, which avoids the inaccurate estimation of incomplete programs. Figure 2 shows the key difference between Ansor's approach and existing approaches. Next, we give an overview of all components of Ansor in §3.

3 Design Overview

Ansor is an automated tensor program generation framework. Figure 4 shows the overall architecture of Ansor. The input of Ansor is a set of to be optimized DNNs. Ansor has three major components: (1) a program sampler that constructs a large search space and samples diverse programs from it; (2) a performance tuner that fine-tunes the performance of sampled programs; (3) a task scheduler that allocates time resources for optimizing multiple subgraphs in the DNNs.

Program sampler. One key challenge Ansor has to address is to generate a large search space for a given computa-

[1] recall@k of top-k = $\frac{|G \cap P|}{k}$, where G is the set of top-k program according to the ground truth and P is the set of top-k programs predicted by the model.

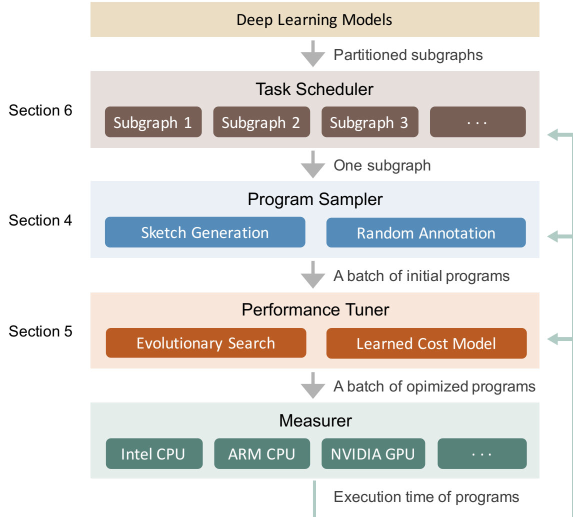


Figure 4: System Overview. The gray arrows show the flow of extracting subgraphs from deep learning models and generating optimized programs for them. The green arrows mean the measurer returns profiling data to update the status of all components in the system.

tional graph. To cover diverse tensor programs with various high-level structures and low-level details, Ansor utilizes a hierarchical representation of the search space with two levels: *sketch* and *annotation* (§4). Ansor defines the high-level structures of programs as sketches and leave billions of low-level choices (*e.g.*, tile size, parallel, unroll annotations) as annotations. This representation allows Ansor to enumerate high-level structures flexibly and sample low-level details efficiently. Ansor includes a program sampler that randomly samples programs from the space to provide comprehensive coverage of the search space.

Performance tuner. The performance of randomly sampled programs is not necessarily good. The next challenge is to fine-tune them. Ansor employs evolutionary search and a learned cost model to perform fine-tuning iteratively (§5). At each iteration, Ansor uses re-sampled new programs as well as good programs from previous iterations as the initial population to start evolutionary search. Evolutionary search fine-tunes programs by mutation and crossover which perform out-of-order rewrite and break the limitation of sequential construction. Querying the learned cost model is orders of magnitude faster than actual measurement, so we can evaluate thousands of programs in seconds via the cost model.

Task scheduler. Using program sampling and performance fine-tuning allows Ansor to find high-performance tensor programs for a computational graph. Intuitively, treating a whole DNN as a single computational graph and generating a full tensor program for it could potentially achieve the optimal performance. This, however, is inefficient because it has to deal with the unnecessary exponential explosion of the search space. Typically, the compiler partitions the large computational graph of a DNN into several small subgraphs [10]. This partition has a negligible effect on the performance thanks to the layer-by-layer construction nature of DNNs. This brings the final challenge of Ansor: how to allocate time resources when generating programs for multiple subgraphs. The task scheduler (§6) in Ansor uses a scheduling algorithm based on gradient descent to allocate resources to the subgraphs that are more likely to improve the end-to-end DNN performance.

4 Program Sampling

The search space an algorithm explores determines the best programs it can find. The considered search spaces in existing approaches are limited by the following factors: (1) Manual enumeration (*e.g.*, TVM [11]). It is impractical to manually enumerate all possible choices by templates, so existing manual templates only cover a limited search space heuristically. (2) Aggressive early pruning (*e.g.*, Halide auto-scheduler [2]). Aggressive early pruning based on evaluating incomplete programs prevents the search algorithm from exploring certain regions in the space.

In this section, we introduce techniques to push the boundary of the considered search space by addressing the above limitations. To solve (1), we automatically expand the search space by recursively applying a set of flexible derivation rules. To avoid (2), we randomly sample complete programs in the search space. Since random sampling gives an equal chance to every point to be sampled, our search algorithm can potentially explore every program in the considered space. We do not rely on random sampling to find the optimal program, because every sampled program is later fined-tuned (§5).

To sample programs that can cover a large search space, we define a hierarchical search space with two levels: *sketch* and *annotation*. We define the high-level structures of programs as sketches and leave billions of low-level choices (*e.g.*, tile size, parallel, unroll annotations) as annotations. At the top level, we generate sketches by recursively applying a few derivation rules. At the bottom level, we randomly annotate these sketches to get complete programs. This representation summarizes a few basic structures from billions of low-level choices, enabling the flexible enumeration of high-level structures and efficient sampling of low-level details.

While Ansor covers both CPU and GPU, we explain the sampling process for CPUs in the rest of this section as an example. The sampling rules for GPUs are mostly the same with minor modifications.

4.1 Sketch Generation

The first column in Figure 5 shows two examples of the input of our problem. The input has three equivalent forms: the mathematical expression, the corresponding naive program got by directly expanding the loop indices, and the corresponding computational graph (directed acyclic graph, or DAG).

Figure 5: Examples of generated sketches and sampled programs. This figure shows two example inputs, three generated sketches and four sampled programs.

To generate sketches for a DAG with multiple nodes, we visit all the nodes in a topological order and build the structure iteratively. For computation nodes that are compute-intensive and have a lot of data reuse opportunities (*e.g.*, conv2d, matmul), we build basic tile and fusion structures for them as the sketch. For simple element-wise nodes (*e.g.*, ReLU, element-wise add), we can safely inline them. Note that new nodes (*e.g.*, caching nodes, layout transform nodes) may also be introduced to the DAG during the sketch generation.

We propose a derivation-based enumeration approach to generate all possible sketches by recursively applying several basic rules. This process takes a DAG as an input and returns a list of sketches. We define the State $\sigma = (S, i)$, where S is the current partially generated sketch for the DAG, and i is the index of the current working node. The nodes in a DAG are sorted in a topological order from output to input. The derivation begins from the initial naive program and the last node, or the initial state $\sigma = (naive\ program, index\ of\ the\ last\ node)$. Then we try to apply all derivation rules to the states recursively. For each rule, if the current state satisfies the application condition, we apply the rule to $\sigma = (S, i)$ and get $\sigma' = (S', i')$ where $i' \leq i$. This way the index i (working node) decreases monotonically. A state becomes a terminal state when $i = 0$. During enumeration, multiple rules can be applied to one state to generate multiple succeeding states. One rule can also generate multiple possible succeeding states. So we maintain a queue to store all intermediate states. The process ends when the queue is empty. All $\sigma.S$ in terminal states form a sketch list at the end of the sketch generation.

Table 1 lists derivation rules we used for the CPU. We first provide the definition of the used predications and then describe the functionality of each rule. We statically analyze the computation definitions to get the values for these predications. $IsStrictInliable(S, i)$ indicates if the node i in S is a simple element-wise operator that can always be inlined (*e.g.* element-wise add, ReLU). $HasDataReuse(S, i)$ indicates if the node i in S is a compute-intensive operator and has plentiful data reuse opportunity (*e.g.*, mat-

mul, conv2d). $HasFusibleConsumer(S,i)$ indicates if the node i in S has only one consumer j and node j can be fused into node i (*e.g.*, matmul + bias_add, conv2d + relu). $HasMoreReductionParallel(S,i)$ indicates if the node i in S has little parallelism in space dimensions but has ample parallelism opportunity in reduction dimensions. (*e.g.* computing 2-norm of a matrix, matmul $C_{2\times 2} = A_{2\times 512} \cdot B_{512\times 2}$). Next we introduce the functionality of each derivation rule.

Rule 1 just simply skips a node if it is not strictly inlinable. Rule 2 always inlines strictly inlinable nodes. Since the conditions of rule 1 and rule 2 are mutually exclusive, a state with $i > 1$ can always satisfy one of them and continue to derive.

Rules 3, 4, and 5 deal with the multi-level tiling and fusion for nodes that have data reuse. Rule 3 performs multi-level tiling for data reusable nodes. For CPU, we use a "SSRSRS" tile structure, where "S" stands for one tile level of space loops and "R" stands for one tile level of reduction loops. For example, in the matmul $C(i,j) = \sum_k A[i,k] \times B[k,j]$, i and j are space loops and k is a reduction loop. The "SSRSRS" tile structure for matmul expands the original 3-level loop (i,j,k) into a 10-level loop $(i_0, j_0, i_1, j_1, k_0, i_2, j_2, k_1, i_3, j_3)$. Although we do not permute the loop order, this multi-level tiling can also cover reorder cases. For example, the above 10-level loop can be specialized to just a simple reorder (k_0, j_2, i_3) by setting the length of other loops to one. Rule 4 performs multi-level tiling and also fuse the fusible consumers. For example, we fuse the element-wise nodes (*e.g.*, ReLU, bias add) into the tiled nodes (*e.g.*, conv2d, matmul). Rule 5 adds a caching node if the current data-reusable node does not have a fusible consumer. For example, the final output node in a DAG does not have any consumer, so it directly writes results into main memory by default and this is inefficient due to the high latency of memory accesses. By adding a cache node, we introduce a new fusible consumer into the DAG, then rule 4 can be applied to fuse this newly added cache node into the final output node. With the cache node fused, now the final output node writes its results into a cache block, and the cache block will be written to the main memory at once when all data in the block is computed.

Rule 6 can use `rfactor` [42] to factorize a reduction loop into a space loop to bring more parallelism.

Figure 5 shows three examples of the generated sketches. The sketches are different from the manual templates in TVM. Because the manual templates specify both high-level structures and low-level details while sketches only define high-level structures. For the example input 1, the sorted order of the four nodes in the DAG is (A,B,C,D). To derive the sketches for the DAG, we start from output node $D(i=4)$ and apply rules to the nodes one by one. Specifically, the derivation for generated sketch 1 is:

$$Input\ 1 \rightarrow \sigma(S_0, i=4) \xrightarrow{\text{Rule 1}} \sigma(S_1, i=3) \xrightarrow{\text{Rule 4}} \sigma(S_2, i=2) \xrightarrow{\text{Rule 1}} \sigma(S_3, i=1) \xrightarrow{\text{Rule 1}} Sketch\ 1$$

For the example input 2, the sorted order of the five nodes is (A,B,C,D,E). Similarly, we start from the output node $E(i=5)$ and apply rules recursively. The generated sketch 2 is derived by:

$$Input\ 2 \rightarrow \sigma(S_0, i=5) \xrightarrow{\text{Rule 5}} \sigma(S_1, i=5) \xrightarrow{\text{Rule 4}} \sigma(S_2, i=4) \xrightarrow{\text{Rule 1}} \sigma(S_3, i=3) \xrightarrow{\text{Rule 1}} \sigma(S_4, i=2) \xrightarrow{\text{Rule 2}} \sigma(S_5, i=1) \xrightarrow{\text{Rule 1}} Sketch\ 2$$

Similarly, the generated sketch 3 is derived by:

$$Input\ 2 \rightarrow \sigma(S_0, i=5) \xrightarrow{\text{Rule 6}} \sigma(S_1, i=4) \xrightarrow{\text{Rule 1}} \sigma(S_2, i=3) \xrightarrow{\text{Rule 1}} \sigma(S_3, i=2) \xrightarrow{\text{Rule 2}} \sigma(S_4, i=1) \xrightarrow{\text{Rule 1}} Sketch\ 3$$

While the presented rules are practical enough to cover the structures for most operators, there are always exceptions. For example, some special algorithms (*e.g.*, Winograd convolution [27]) and accelerator intrinsics (*e.g.*, TensorCore [34]) require special tile structures to be effective. Although the template-guided search approach (in TVM) can craft a new template for every new case, it needs a great amount of design effort. It is not straightforward for the sequential construction based search with a fixed predefined order to expand its construction sequence either. On the other hand, the derivation-based sketch generation in Ansor is flexible enough to generate the required structures for emerging algorithms and hardware, as we allow users to register new derivation rules and integrate them seamlessly with existing rules.

4.2 Random Annotation

The sketches generated by the previous subsection are incomplete programs because they only have tile structures without specific tile sizes and loop annotations, such as parallel, unroll, and vectorization. In this subsection, we annotate sketches to be complete programs for fine-tuning and evaluation.

Given a list of generated sketches, we randomly pick one sketch, randomly fill out tile sizes, parallelize some outer loops, vectorize some inner loops, and unroll a few inner loops. We also randomly change the computation location of some nodes in the program to make a slight tweak to the tile structure. If some special algorithms (*e.g.*, Winograd convolution [27]) require special annotation policy to be effective, we allow users to give simple hints in the computation definition to adjust the annotation policy. Finally, since it is valid to arbitrarily change the layout of constant tensors, we rewrite the layouts of the constant tensors according to the multi-level tile structure to make them most cache-friendly and eliminate layout transformation overheads between nodes. This optimization is effective because the weight tensors of convolution or dense layers are constants for inference applications.

Examples of random sampling are shown in Figure 5. The sampled program might have a different number of loops than the sketch because the loops with length one are simplified.

No	Rule Name	Condition	Application
1	Skip	$\neg IsStrictInlinable(S,i)$	$S' = S; i' = i-1$
2	Always Inline	$IsStrictInlinable(S,i)$	$S' = Inline(S,i); i' = i-1$
3	Multi-level Tiling	$HasDataReuse(S,i)$	$S' = MultiLevelTiling(S,i); i' = i-1$
4	Multi-level Tiling with Fusion	$HasDataReuse(S,i) \wedge HasFusibleConsumer(S,i)$	$S' = FuseConsumer(MultiLevelTiling(S,i),i); i' = i-1$
5	Add Cache Stage	$HasDataReuse(S,i) \wedge \neg HasFusibleConsumer(S,i)$	$S' = AddCacheWrite(S,i); i = i'$
6	Reduction Factorization	$HasMoreReductionParallel(S,i)$	$S' = AddRfactor(S,i); i' = i-1$
...	User Defined Rule	...	...

Table 1: Derivation rules used to generate sketches. The condition runs on the current state $\sigma = (S,i)$. The application derives the next state $\sigma' = (S',i')$ from the current state σ. Note that some function (*e.g.*, $AddRfactor$, $FuseConsumer$) can return multiple possible values of S'. In this case we collect all possible S', and return multiple next states σ' for a single input state σ.

5 Performance Fine-tuning

The programs sampled by the program sampler have good coverage of the search space, but their qualities are not guaranteed. This is because the optimization choices, such as tile structure and loop annotations, are all randomly sampled. In this section, we introduce the performance tuner that fine-tunes the performance of the sampled programs via evolutionary search and a learned cost model.

The fine-tuning is performed iteratively. At each iteration, we first use evolutionary search to find a small batch of promising programs according to a learned cost model. We then measure these programs on hardware to get the actual execution time cost. Finally, the profiling data got from measurement is used to re-train the cost model to make it more accurate.

The evolutionary search uses randomly sampled programs as well as high-quality programs from the previous measurement as the initial population and applies mutation and crossover to generate the next generation. The learned cost model is used to predict the *fitness* of each program, which is the throughput of one program in our case. We run evolution for a fixed number of generations and pick the best programs found during the search. We leverage a learned cost model because the cost model can give relatively accurate estimations of the fitness of programs while being orders of magnitudes faster than the actual measurement. It allows us to compare tens of thousands of programs in the search space in seconds, and pick the promising ones to do actual measurement.

5.1 Evolutionary Search

Evolutionary search [49] is a generic meta-heuristic algorithm inspired by biological evolution. By iteratively mutating high-quality programs, we can generate new programs with potentially higher quality. The evolution starts from the sampled initial generation. To generate the next generation, we first select some programs from the current generation according to certain probabilities. The probability of selecting a program is proportional to its fitness predicted by the learned cost model (§5.2), meaning that the program with a higher performance score has a higher probability to be selected. For the selected programs, we randomly apply one of the evolution operations to generate a new program. Basically, for every decision we made during sampling (§4.2), we design corresponding evolution operations to rewrite and fine-tune it. We highlight some of them as follows.

Tile size mutation. This operation scans the program and randomly selects a tiled loop. For this tiled loop, it divides a tile size of one tile level by a random factor and multiplies this factor to another level. Since this operation keeps the product of tile sizes equal to the original loop length, the mutated program is always valid.

Parallel, vectorization mutation. This operation scans the program and randomly selects a loop annotated with parallel or vectorization. For this loop, this operation changes the granularity by either fusing its adjacent loop levels or splitting it by a factor.

Node-based crossover. Crossover is an operation to generate new offspring by combining the genes from two or more parents. The genes of a program in Ansor are its rewriting steps. Every program generated by Ansor is rewritten from its initial naive implementation. Ansor preserves a complete rewriting history for each program during sketch generation and random annotation. We can treat rewriting steps as the genes of a program because they describe how this program is formed from the initial naive one. Based on this, we can generate a new program by combining the rewriting steps of two existing programs. However, arbitrarily combining rewriting steps from two programs might break the dependencies in steps and create an invalid program. For example, a rewriting step annotates vectorization to the innermost loop, and the innermost loop may be generated by a previous tiling step. As a result, the granularity of crossover operation in Ansor is based on nodes in the DAG, because the rewriting steps for different nodes usually have less dependency. Ansor uses the cost model to estimate the performance scores of each node in the two programs and merges the rewriting steps of nodes with higher scores. When there are dependencies between nodes, Ansor try to analyze and adjust the steps with simple heuristics. Ansor further verifies the merged programs to guarantee the functional correctness. The verification is simple because Ansor only uses a small set of loop transformation rewriting steps, and the underlying code generator can check the correctness by dependency analysis.

The evolutionary search leverages mutation and crossover to generate a new set of candidates repeatedly for several

rounds and outputs a small set of programs with the highest scores. These programs will be compiled and measured on the target hardware to obtain the real running time cost. The collected measurement data is then used to update the cost model. In this way, the accuracy of the learned cost model is gradually improved to match the target hardware. Consequently, the evolutionary search gradually generates higher-quality programs for the target hardware platform.

5.2 Learned Cost Model

A cost model is necessary for estimating the performance of programs during the search. We adopt a learned cost model similar to related works [2, 11] with newly designed program features. A learned cost model has great portability because a single model design can be reused for different hardware targets by feeding in different training data.

Since our target programs are mainly data parallel tensor programs, which are made by multiple interleaved loop nests with several assignment statements as the innermost statements, we train the cost model to predict the score of one innermost non-loop statement in a loop nest. For a full program, we make predictions for each innermost non-loop statement and add the predictions up as the score. We build the feature vector for an innermost non-loop statement by extracting features in the context of a full program. The extracted features include arithmetic features and memory access features. A detailed list of extracted features is in the Appendix B.

We use weighted squared error as the loss function. Because we mostly care about identifying the well-performing programs from the search space, we put more weight on the programs that run faster. Specifically, the loss function of the model f on a program P with throughput y is $loss(f,P,y) = w_p(\sum_{s\in S(P)} f(s) - y)^2 = y(\sum_{s\in S(P)} f(s) - y)^2$ where $S(P)$ is the set of innermost non-loop statements in P. We directly use the throughput y as weight. We train a gradient boosting decision tree [8] as the underlying model f. A single model is trained for all tensor programs coming from all DAGs, and we normalize the throughput of all programs come from the same DAG to be in the range of $[0,1]$.

6 Task Scheduler

A DNN can be partitioned into many independent subgraphs (*e.g.*, conv2d + relu). For some subgraphs, spending time in tuning them does not improve the end-to-end DNN performance significantly. This is due to two reasons: either (1) the subgraph is not a performance bottleneck, or (2) tuning brings only minimal improvement in the subgraph's performance.

To avoid wasting time on tuning unimportant subgraphs, Ansor dynamically allocates different amounts of time resources to different subgraphs. Take ResNet-50 for example, it has 29 unique subgraphs among all 50 convolution layers. Most of these subgraphs are convolution layers with different shapes configurations (input size, kernel size, stride, etc). We need to generate different programs for different convolution layers because the best tensor program depends on these shape configurations. We define a task as a process performed to generate high-performance programs for a subgraph. It means that optimizing a single DNN requires finishing dozens of tasks (*e.g.*, 29 tasks for ResNet-50). In reality, we may even want to optimize a set of DNNs, which leads to even more tasks. A subgraph can also appear multiple times in a DNN or across different DNNs.

Ansor's task scheduler allocates time resources to tasks in an iterative manner. At each iteration, Ansor selects a task, generates a batch of promising programs for the subgraph, and measures the program on hardware. We define such an iteration as one unit of time resources. When we allocate one unit of time resources to a task, the task obtains an opportunity to generate and measure new programs, which also means the chance to find better programs. We next present the formulation of the scheduling problem and our solution.

6.1 Problem Formulation

When tuning a DNN or a set of DNNs, a user can have various types of goals, for example, reducing a DNN's latency, meeting latency requirements for a set of DNNs, minimizing tuning time when tuning no longer improves DNN performance significantly. We thus provide users a set of objective functions to express their goals. Users can also provide their own objective functions.

Suppose there are n tasks in total. Let $t \in \mathbf{Z}^n$ be the allocation vector, where t_i is the number of time units spent on task i. Let the minimum subgraph latency task i achieves be a function of the allocation vector $g_i(t)$. Let the end-to-end cost of the DNNs be a function of the latency of the subgraphs $f(g_1(t), g_2(t), ..., g_3(t))$. Our objective is to minimize the end-to-end cost:

$$\text{minimize } f(g_1(t), g_2(t), ..., g_3(t))$$

To minimize the end-to-end latency of a single DNN, we can define $f(g_1, g_2, ..., g_n) = \sum_{i=1}^{n} w_i \times g_i$, where w_i is the number of appearances of task i in the DNN. This formulation is straightforward because f is an approximation of the end-to-end DNN latency.

When tuning a set of DNNs, there are several options. Table 2 shows a number of example objective functions for tuning multiple DNNs. Let m be the number of DNNs, $S(j)$ is the set of tasks that belong to DNN j. f_1 adds up the latency of every DNN, which means to optimize the cost of a pipeline that sequentially runs all DNNs once. In f_2, we define L_j as the latency requirement of DNN j, meaning that we do not want to spend time on a DNN if its latency has already met the requirement. In f_3, we define B_j as the reference latency of a DNN j. As a result, our goal is to maximize the geometric mean of speedup against the given reference latency. Finally in f_4, we define a function $ES(g_i,t)$ that returns an

$f_1 = \sum_{j=1}^{m} \sum_{i \in S(j)} w_i \times g_i(t)$
$f_2 = \sum_{j=1}^{m} \max(\sum_{i \in S(j)} w_i \times g_i(t), L_j)$
$f_3 = -(\prod_{j=1}^{m} \frac{B_j}{\sum_{i \in S(j)} w_i \times g_i(t)})^{\frac{1}{m}}$
$f_4 = \sum_{j=1}^{m} \sum_{i \in S(j)} w_i \times \max(g_i(t), ES(g_i, t))$

Table 2: Examples of objective functions for multiple neural networks

early stopping value by looking at the history of latency of task i. It can achieve the effect of per task early stopping.

6.2 Optimizing with Gradient Descent

We propose a scheduling algorithm based on gradient descent to efficiently optimize the objective function. Given the current allocation t, the idea is to approximate the gradient of the objective function $\frac{\partial f}{\partial t_i}$ in order to choose the task i such that $i = \text{argmax}_i |\frac{\partial f}{\partial t_i}|$. We approximate the gradient by doing optimistic guess and considering the similarity between tasks. The derivation can be found in the Appendix A. We approximate the gradient by

$$\frac{\partial f}{\partial t_i} \approx \frac{\partial f}{\partial g_i}(\alpha \frac{g_i(t_i) - g_i(t_i - \Delta t)}{\Delta t} + (1-\alpha)(\min(-\frac{g_i(t_i)}{t_i}, \beta \frac{C_i}{\max_{k \in N(i)} V_k} - g_i(t_i))))$$

where Δt is a small backward window size, $g_i(t_i)$ and $g_i(t_i - \Delta t)$ are known from the history allocations. $N(i)$ is the set of similar tasks of i, C_i is the number of floating point operations in task i and V_k is the number of floating point operation per second we can achieve in task k. The parameter α and β controls the weight to trust some predictions.

To run the algorithm, Ansor starts from $t = \mathbf{0}$ and warms up with a round of round-robin to get an initial allocation vector $t = (1, 1, ..., 1)$. After the warm-up, at each iteration, we compute the gradient of each task and pick $\text{argmax}_i |\frac{\partial f}{\partial t_i}|$. Then we allocate the resource unit to task i and update the allocation vector $t_i = t_i + 1$. The optimization process continues until we run out of the time budget. To encourage exploration, we adopt a ε-greedy strategy [43], which preserves a probability of ε to randomly select a task.

Take the case of optimizing for a single DNN's end-to-end latency for example, Ansor prioritizes a subgraph that has a high initial latency because our optimistic guess says we can reduce its latency quickly. Later, if Ansor spends many iterations on it without observing a decrease in its latency, Ansor leaves the subgraph because its $|\frac{\partial f}{\partial t_i}|$ decreases.

7 Evaluation

The core of Ansor is implemented in C++ with about 10K lines of code (3K for the search policy, 7K for other infrastructures). Ansor generates programs in its own intermediate representation (IR). These programs are then lowered to TVM IR for code generation targeting various hardware platforms. Ansor only utilizes TVM as a deterministic code generator.

We evaluate Ansor on three levels: single operator, subgraph, and entire neural network. For each level of evaluation, we compare Ansor against the state-of-the-art search frameworks and hardware-specific manual libraries.

The generated tensor programs were benchmarked on three hardware platforms: an Intel CPU (20-core Platinum 8269CY@3.1 GHz), an NVIDIA GPU (V100), and an ARM CPU (4-core Cortex-A53@1.4GHz on the Raspberry Pi 3b+). We use float32 as the data type for all evaluations.

7.1 Single Operator Benchmark

We first evaluate Ansor on a set of common deep learning operators, including 1D, 2D and 3D convolution (C1D, C2D, C3D respectively), matrix multiplication (GMM), group convolution (GRP), dilated convolution (DIL) [52], depth-wise convolution (DEP) [22], transposed 2D convolution (T2D) [37], capsule 2D convolution (CAP) [21] and matrix 2-norm (NRM). For each operator, we select 4 different shape configurations from common DNNs and evaluate them with two batch sizes (1 and 16). In total, there are 10 operators $\times 4$ shape configurations $\times 2$ batch size (= 80) test cases. We run these test cases on the Intel CPU.

We include PyTorch [36], Halide auto-scheduler [2], FlexTensor [53] and AutoTVM [11] as baseline frameworks. PyTorch is backed by the vendor-provided kernel library MKL-DNN [24]. Halide auto-scheduler is a sequential construction based search framework for Halide. AutoTVM and FlexTensor are template-guided search frameworks based on TVM. Since Halide auto-scheduler does not have a pre-trained cost model for AVX-512, we disabled AVX-512 for search-based frameworks, while the MKL-DNN in PyTorch utilizes AVX-512 by default.

For each test case in this evaluation, we let search frameworks (*i.e.*, Halide auto-scheduler, FlexTensor, AutoTVM, and Ansor) run search or auto-tuning with at most $1,000$ measurement trials. This means each framework can measure at most 80×1000 programs for auto-tuning in this evaluation. Using the same number of measurement trials makes it a fair comparison without involving implementation details. For a single operator, $1,000$ measurement trials are typically enough for the search to converge in these frameworks.

Figure 6 shows the normalized performance. For each operator, we compute the geometric mean of the throughputs on four shapes and normalize the geometric means of all frameworks relative to the best one. As shown in the figure, Ansor performs the best on 19 out of 20 test cases. Ansor outperforms existing search frameworks by $1.1 - 32.7\times$. The performance improvements of Ansor come from both its large search space and effective exploration strategy. For most operators, we found the best program generated by Ansor is out-

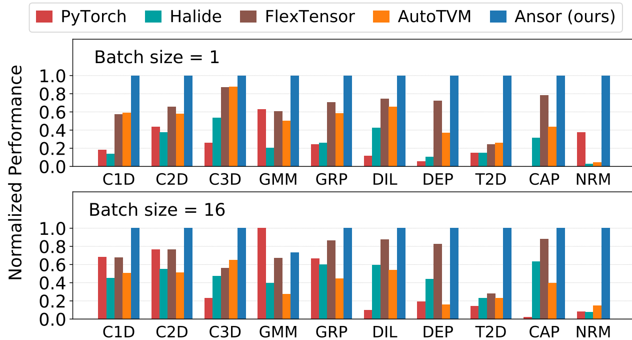


Figure 6: Single operator performance benchmark on a 20-core Intel-Platinum-8269CY. The y-axis is the throughput normalized to the best throughput for each operator.

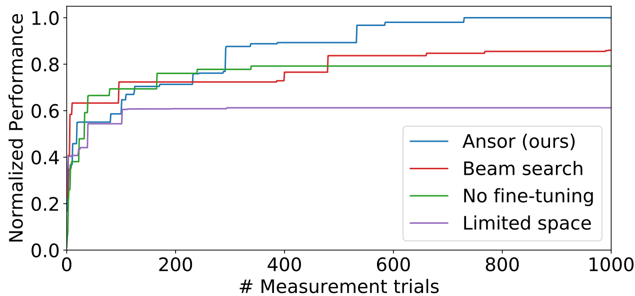


Figure 7: Ablation study of four variants of Ansor on a convolution operator. The y-axis is the throughput relative to the throughput of the best program.

side the search space of existing search frameworks because Ansor is able to explore more optimization combinations. For example, the significant speedup on NRM is because Ansor can parallelize reduction loop, while other frameworks do not. The large speedup on T2D is because Ansor can use correct tile structures and unrolling strategies to let the code generator simplify the multiplication of zeros in strided transposed convolution. In contrast, other frameworks fail to capture many effective optimizations in their search space, making them unable to find the programs that Ansor does. For example, the unfolding rules in Halide does not split the reduction loop in GMM and does not split reduction loops in C2D when padding is computed outside of reduction loops. The manual templates in AutoTVM have limited tile structures, as they cannot cover the structure of "Generated Sketch 1" in Figure 5. The manual template in FlexTensor does not change the computation location of padding and has a fixed unrolling policy. Finally, for the only case (GMM with batch size 16) that Ansor performs worse than PyTorch, it is due to the disabling of AVX-512. Ansor can match PyTorch after utilizing AVX-512.

Ablation study. We run four variants of Ansor on a convolution operator and report the performance curve. We pick the last convolution operator in ResNet-50 with batch size=16 as the test case, because it has a large enough interesting search space to evaluate the search algorithms. Other operators share a similar pattern. In Figure 7, each curve is the median of 5 runs. "Ansor (ours)" uses all our introduced techniques. "Beam Search" means we prune incomplete programs with the cost model during the sampling process and do not use fine-tuning. "No fine-tuning" is based on "Ansor (ours)" but disables fine-tuning and only relies on random sampling. "Limited space" is also based on "Ansor (ours)" but limits the search space to make it similar to the space in existing manual templates. As demonstrated by Figure 7, dropping either the large search space or efficient fine-tuning decreases the final performance significantly. The aggressive early pruning in "Beam search" throws away incomplete programs with good final performance due to inaccurate estimation.

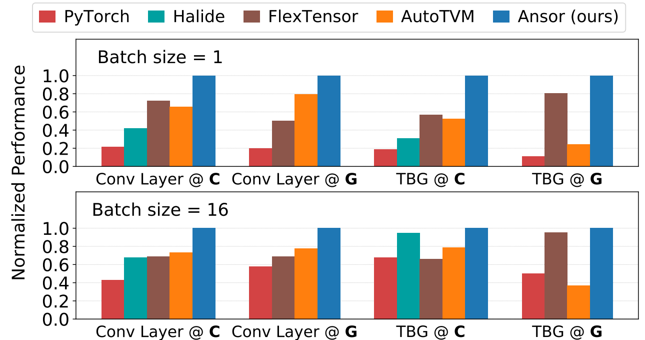


Figure 8: Subgraph performance benchmark on a 20-core Intel-Platinum-8269CY and an NVIDIA V100. "@**C**" denotes CPU results and "@**G**" denotes GPU results. The y-axis is the throughput normalized to the best throughput for each subgraph.

7.2 Subgraph Benchmark

We perform subgraph benchmark on two common subgraphs in DNNs. The "ConvLayer" is a subgraph consisting of 2D convolution, batch normalization [25] and ReLU activation, which is a common pattern in convolutional neural networks. The "TBG" is a subgraph consisting of two matrix transposes and one batch matrix multiplication, which is a common pattern in the multi-head attention [46] in language models. Similar to single operator benchmark, we select four different shape configurations and two batch sizes and run auto-tuning with up to $1,000$ measurement trails per test case. We use the same set of baseline frameworks and run the benchmark on the Intel CPU and the NVIDIA GPU. We do not report the performance of Halide auto-scheduler on GPU because its GPU support is still in an experimental stage.

Figure 8 shows that Ansor outperforms manual libraries and other search frameworks by $1.1-1.8\times$. Ansor can generate high-performance programs consistently for these subgraphs on both platforms. In comparison, other frameworks perform poorly on certain cases. For example, the template in FlexTensor is mainly designed for a single operator, so it lacks the ability to perform GPU kernel fusion in some cases. Relatively, FlexTensor performs worse on "ConvLayer@G" than on "TBG@G"" because it cannot fuse batch normalization

and ReLU into the convolution operator.

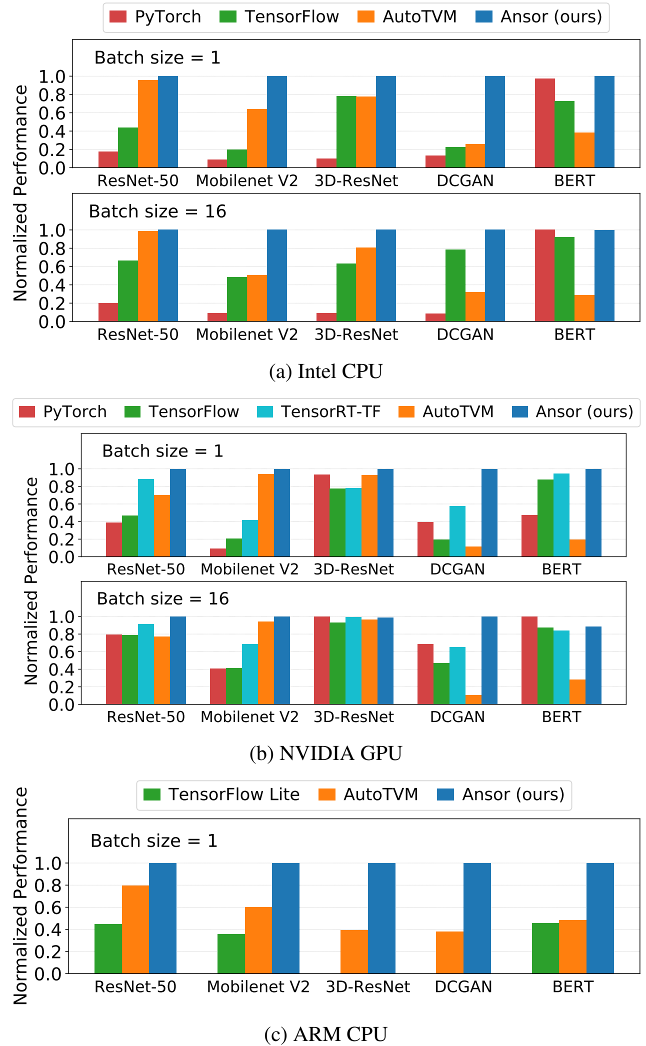
7.3 End-to-End Network Benchmark

We benchmark the end-to-end inference execution time of several DNNs, which include ResNet-50 [20] and MobileNet-V2 [40] for image classification, 3D-ResNet-18 [19] for action recognition, DCGAN [37] generator for image generation, and BERT [14] for language understanding. We benchmark these DNNs on three hardware platforms. For the server-class Intel CPU and NVIDIA GPU, we report the results for batch size 1 and batch size 16. For the ARM CPU in the edge device, real-time feedback is typically desired, so we only report the results for batch size 1.

We include PyTorch, TensorFlow, TensorRT (TensorFlow integration) [35], TensorFlow Lite and AutoTVM as baseline frameworks. We do not include Halide auto-scheduler or FlexTensor because they lack the support of widely-used deep learning model formats (*e.g.*, the formats in PyTorch and TensorFlow) and high-level graph optimizations. As a result, we expect that the end-to-end execution time they can achieve will be the sum of the latency of all subgraphs in a DNN. In contract, AutoTVM can optimize a whole DNN with its manual templates and various graph-level optimizations (*e.g.*, graph-level layout search [29], graph-level constant folding [39]). We let both AutoTVM and Ansor run auto-tuning with up to $1000 \times n$ measurement trials on each DNN, where n is the number of subgraphs in the DNN. They have similar search overhead, so it roughly takes the same amount of time for them to do the same number of measurements. We set the objective of the task scheduler as minimizing the total latency of one DNN and generate programs for these test cases one by one. On the other hand, PyTorch, TensorFlow, TensorRT, and TensorFlow Lite are all backed by static kernel libraries (MKL-DNN on Intel CPU, CuDNN on NVIDIA GPU, and Eigen on ARM CPU) and do not need auto-tuning. We enabled AVX-512 for all frameworks on the CPU in this network benchmark.

Figure 9 shows the results on the Intel CPU, NVIDIA GPU and ARM CPU [2]. Compared with search-based AutoTVM, Ansor matches or outperforms it on all cases with $1.0-9.4\times$ speedup. Compared with the best alternative, Ansor improves the execution performance of DNNs on the Intel CPU, ARM CPU, and NVIDIA GPU by up to $3.8\times$, $2.6\times$, and $1.7\times$, respectively. Overall, Ansor performs the best or equally the best on 24 out of 25 cases. The only exception is BERT with batch size 16 on the GPU. This is because BERT consists of many matrix multiplications with large input sizes. It is hard for compilation-based approaches to beat manually-written assembly code on large matrix multiplications [5, 10, 45], as the code has been hand-optimized for decades.

[2] 3D-ResNet and DCGAN are not yet supported by TensorFlow Lite on the ARM CPU.



(a) Intel CPU

(b) NVIDIA GPU

(c) ARM CPU

Figure 9: Network inference performance benchmark on three hardware platforms. The y-axis is the throughput relative to the best throughput for each network.

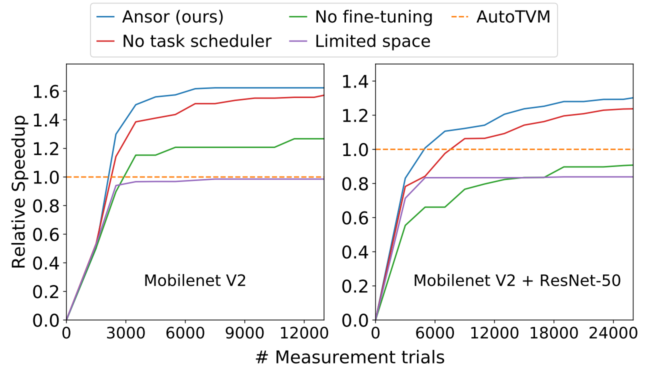


Figure 10: Network performance auto-tuning curve. The y-axis is the speedup relative to AutoTVM.

Ablation study. We run variants of Ansor on two test cases in Figure 10. In the left figure, we run four variants of Ansor to generate programs for a single mobilenet-V2. In the right figure, we run these variants for both mobilenet-V2 and ResNet-

50. We set the objective function of the task scheduler to be the geometric mean of speedup against AutoTVM. As shown in Figure 10, "No task scheduler" means we use a round-robin strategy to allocate equal time resources to all subgraphs. "Limited space" is based on "Ansor (ours)" but limits the search space. "No fine-tuning" is also based on "Ansor (ours)" but disables fine-tuning and relies on random sampling only. As can be seen in Figure 10, "Limited space" performs the worst in terms of the final achieved performance, proving that the best programs are not included in the limited space. The final achieved performance can be improved by enlarging the search space, as depicted in "No fine-tuning". However, in the right figure, randomly assigning tile sizes and annotations still cannot beat AutoTVM in the given time budget. After enabling fine-tuning, "No task scheduler" outperforms AutoTVM in both cases. Finally, "Ansor (ours)" employs the task scheduler to prioritize performance bottlenecks (*e.g.*, subgraphs contain 3x3 convolution), so it performs the best in both search efficiency and the final achieved performance.

Search time. Ansor searches efficiently and can outperform or match AutoTVM with less search time. AutoTVM does not have a task scheduler so it generates programs for all subgraphs sequentially with a predefined budget of measurement trials. To get the reference results in Figure 10, it requires around $30,000$ measurement trials for mobilenet-V2 and $50,000$ measurement trials for mobilenet-V2 and ResNet-50. However, Ansor can match its performance on these two cases with $10\times$ less measurement trials, thanks to the task scheduler, efficient fine-tuning and comprehensive coverage of effective optimizations. As a reference, depending on the target platforms and the complexity of the subgraph, it takes about one to two seconds to compile one program and measure it with other search overhead amortized. Therefore, it takes several hours to generate programs for a DNN. This is acceptable for inference applications, because we only need to run program generation for the DNNs once before deployment. Comparing Ansor against AutoTVM on other network benchmark cases, we observe similar significant reductions in the wall clock search time up to $10\times$.

8 Related Work

Automatic tensor programs generation based on scheduling languages. Halide [38] introduces a scheduling language that can describe loop optimization primitives. This language is suitable for both manual optimization and automatic search. Halide has three versions of auto-scheduler based on different techniques [2, 28, 33]. The latest one with beam search and learned cost model performs the best among them, which is also used in our evaluation. TVM [10] utilizes a similar scheduling language and includes a template-guided search framework AutoTVM [11]. Similar to the motivation of this paper, FlexTensor [53] attempts to reduce human efforts in writing templates. It proposes more general templates targeting a set of operators so that the required number of templates could be reduced. Its templates are still manually designed, so it fails to cover certain operators and is lacking the support for some important optimizations (*e.g.*, operator fusion).

Polyhedral compilation models. Polyhedral compilation model [7, 47, 48] formulates the optimization of programs as an integer linear programming (ILP) problem. It optimizes a program with affine loop transformation that minimizes the data reuse distance between dependent statements. Tiramisu [5] and TensorComprehensions [45] are two polyhedral compilers that also target deep learning domain. Tiramisu provides a scheduling language similar to Halide language, and it needs manual scheduling. TensorComprehensions can search for GPU code automatically, but it is not yet meant to be used for compute-bounded problems [10]. It cannot outperform TVM on operators like conv2d and matmul [10, 44]. This is because of the lack of certain optimizations and the inaccurate implicit cost model in the polyhedral formulation.

Graph level optimization for deep learning. Graph level optimizations treat an operator in the computational graph as a basic unit and perform optimization at graph level without changing the internal implementations of operators. The common optimizations at graph level include layout optimizations [29], operator fusion [10, 35], constant folding [39], auto-batching [30], automatic generation of graph substitution [26] and so forth. The graph-level optimizations are typically orthogonal to operator-level optimizations. It can also benefit from high-performance implementations of operators. For example, general operator fusion relies on the code generation ability of Ansor. We leave the joint optimization of Ansor and more graph level optimization as future work.

Search-based compilation and auto-tuning. Search based compilation and auto-tuning have already shown its effectiveness in domains other than deep learning. Stock [41] is a super-optimizer based on random search. Stock searches for loop-free hardware instruction sequences, while Ansor generates tensor programs with nests of loops. OpenTuner [4] is a general framework for program auto-tuning based on multi-armed bandit approaches. OpenTuner relies on user-specified search space, while Ansor constructs the search space automatically. Traditional high-performance libraries such as ATLAS [51] and FFTW [16] also utilizes auto-tuning. More recent works NeuroVectorizer [17] and AutoPhase [18, 23] use deep reinforcement learning to automatically vectorize programs and optimize the compiler phase ordering.

9 Conclusion

We proposed Ansor, an automated search framework that generates high-performance tensor programs for deep neural networks. By efficiently exploring a large search space, Ansor finds high-performance programs that are outside the search space of existing approaches. Ansor outperforms existing manual libraries and search-based frameworks on a diverse

set of neural networks and hardware platforms by up to 3.8×. All of Ansor's source code will be publicly available.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4):1–12, 2019.

[3] Hassan Abu Alhaija, Siva Karthik Mustikovela, Lars Mescheder, Andreas Geiger, and Carsten Rother. Augmented reality meets deep learning for car instance segmentation in urban scenes. In *British machine vision conference*, volume 1, page 2, 2017.

[4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014.

[5] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[6] Paul Barham and Michael Isard. Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 177–183, 2019.

[7] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, 2008.

[8] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, 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)*, pages 578–594, 2018.

[11] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pages 3389–3400, 2018.

[12] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[13] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3213–3223, 2016.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[15] Álvaro Fialho, Luis Da Costa, Marc Schoenauer, and Michèle Sebag. Analyzing bandit-based adaptive operator selection mechanisms. 2010.

[16] Matteo Frigo and Steven G Johnson. Fftw: An adaptive software architecture for the fft. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, volume 3, pages 1381–1384. IEEE, 1998.

[17] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. Neurovectorizer: end-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 242–255, 2020.

[18] Ameer Haj-Ali, Qijing Huang, William Moses, John Xiang, John Wawrzynek, Krste Asanovic, and Ion Stoica. Autophase: Juggling hls phase orderings in random

forests with deep reinforcement learning. In *Third Conference on Machine Learning and Systems (ML-Sys)*, 2020.

[19] Kensho Hara, Hirokatsu Kataoka, and Yutaka Satoh. Can spatiotemporal 3d cnns retrace the history of 2d cnns and imagenet? In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6546–6555, 2018.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[21] Geoffrey E Hinton, Sara Sabour, and Nicholas Frosst. Matrix capsules with em routing. 2018.

[22] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[23] Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. Autophase: Compiler phase-ordering for hls with deep reinforcement learning. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 308–308. IEEE, 2019.

[24] Intel. Intel® math kernel library for deep learning networks, 2017.

[25] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[26] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[27] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.

[28] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in halide. *ACM Transactions on Graphics (TOG)*, 37(4):139, 2018.

[29] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing {CNN} model inference on cpus. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 1025–1040, 2019.

[30] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.

[31] Mark F. Medress, Franklin S Cooper, Jim W. Forgie, CC Green, Dennis H. Klatt, Michael H. O'Malley, Edward P Neuburg, Allen Newell, DR Reddy, B Ritea, et al. Speech understanding systems: Report of a steering committee. *Artificial Intelligence*, 9(3):307–316, 1977.

[32] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, et al. A hardware–software blueprint for flexible deep learning specialization. *IEEE Micro*, 39(5):8–16, 2019.

[33] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):83, 2016.

[34] Nvidia. Nvidia tensor cores, 2017.

[35] Nvidia. Nvidia tensorrt: Programmable inference accelerator, 2017.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[37] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[38] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[39] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Josh Pollock, Logan Weber, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. Relay: A high-level ir for deep learning. *arXiv preprint arXiv:1904.08368*, 2019.

[40] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[41] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316, 2013.

[42] Patricia Suriana, Andrew Adams, and Shoaib Kamil. Parallel associative reductions in halide. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 281–291. IEEE, 2017.

[43] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[44] Philippe Tillet, HT Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[45] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[47] Sven Verdoolaege. Presburger formulas and polyhedral compilation. 2016.

[48] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–23, 2013.

[49] Pradnya A Vikhar. Evolutionary algorithms: A critical review and its future prospects. In *2016 International conference on global trends in signal processing, information computing and communication (ICGTSPICC)*, pages 261–265. IEEE, 2016.

[50] Leyuan Wang, Zhi Chen, Yizhi Liu, Yao Wang, Lianmin Zheng, Mu Li, and Yida Wang. A unified optimization approach for cnn model inference on integrated gpus. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

[51] R Clinton Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 38–38. IEEE, 1998.

[52] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

[53] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.

A Gradient Approximation for the Task Scheduler

Now we show how to approximate the gradient for the objective function f. First, do the approximation $g_i(t) \approx g_i(t_i)$. This means we assume the best cost of task i depends only on the resource units spent on it. This may not be true because all tasks share a cost model. Different resource allocations lead to different collections of training data, which then leads to different cost models. Here we make this approximation to continue derivation:

$$\begin{aligned}
\frac{\partial f}{\partial t_i} &= \frac{\partial f}{\partial g_i}\frac{\partial g_i}{\partial t_i} \\
&\approx \frac{\partial f}{\partial g_i}\left(\alpha\frac{g_i(t_i)-g_i(t_i-\Delta t)}{\Delta t} + (1-\alpha)\frac{g_i(t_i+\Delta t)-g_i(t_i)}{\Delta t}\right) \\
&\approx \frac{\partial f}{\partial g_i}\left(\alpha\frac{g_i(t_i)-g_i(t_i-\Delta t)}{\Delta t} + (1-\alpha)(g_i(t_i+1)-g_i(t_i))\right)
\end{aligned}$$

In this expression, Δt is a small backward window size, $g_i(t_i)$ and $g_i(t_i-\Delta t)$ are known from the history allocations. But $g_i(t_i+1)$ is unknown because we have not allocated t_i+1 units of resource to this task. So we have to predict this value. The parameter α controls the weight to trust the prediction. We predict $g_i(t_i+1)$ in two ways. First, we have an optimistic guess that if we spend extra t_i, we can decrease the latency of task i to 0. This means $g_i(t_i+1) \approx g_i(t_i) - \frac{g_i(t_i)}{t_i}$. Second, if subgraphs are structurally similar, their latency is also similar per floating point operation. Considering both factors, we have the following approximation:

$$g_i(t_i+1) \approx \min\left(g_i(t_i) - \frac{g_i(t_i)}{t_i}, \beta\frac{C_i}{\max_{k\in N(i)} V_k}\right)$$

where $N(i)$ is the set of similar tasks of i, C_i is the number of floating point operations in task i and V_k is the number of

floating point operation per second we can achieve in task k. The parameter β controls the weight to trust the prediction based on similarity.

B The List of Extracted Features

We extract the following features for one innermost non-loop statement in the context of a full tensor program. The features include categorical features and numerical features. We use one-hot encoding to encode category features. The length of a feature vector including all the listed features for one statement is 164. We use the same set of features for both CPU and GPU.

- **Numbers of float operationrs**. The numbers of addition, subtraction, division, modulo operation, comparison, intrinsic math function call (e.g. exp, sqrt) and other math function call respectively, with floating point operands.
- **Number of integer operators.** Similar to the above one, but for operations with integer operands.
- **Vectorization related features.** The length of the innermost vectorized loop. The type of vectorization position (InnerSpatial, MiddleSpatial, OuterSpatial, InnerReduce, MiddleReduce, OuterReduce, Mixed, None). The product of the lengths of all vectorized loops. The number of vectorized loops.
- **Unrolling related features.** Similar to the vectorization related features, but for unrolling.
- **Parallelization related features.** Similar to the vectorization related features, but for parallelization.
- **GPU thread binding related features.** The lengths of blockIdx.x, blockIdx.y, blockIdx.z, threadIdx.x, threadIdx.y, threadIdx.z and virtual threads [10].
- **Arithmetic intensity curve.** Arithmetic intensity is defined as $\frac{\text{The number of floating point operations}}{\text{The number of bytes accessed}}$. We compute the arithmetic intensity for each loop level and draw a curve with linear interpolation. Then we sample 10 points from this curve.
- **Buffer Access Feature** For each buffer this statement accesses, we extract features for it. While different statement can access different numbers of buffers, we perform feature extraction for at most five buffers. We pad zeros if a statement accesses less then five buffers and remove small buffers if a statement accesses more than five buffers.
  - **Access type**. The type of access (read, write, read + write).
  - **Bytes**. The total number of bytes accessed by this statement.
  - **Unique bytes**. The total number of unique bytes accessed by this statement.
  - **Lines**. The total number of cache lines accessed by this statement.
  - **Unique lines**. The total number of unique cache lines accessed by this statement.
  - **Reuse type**. The type of data reuse (LoopMultipleRead, SerialMultipleRead, NoReuse).
  - **Reuse distance**. The distance between data reuse in terms of number of for loop iterations and total accessed bytes.
  - **Reuse counter**. The number of the happening of data reuse.
  - **Stride**. The stride of access.
  - **Accessed bytes divided by reuse**. We compute the following values: $\frac{\text{Bytes}}{\text{Reuse counter}}$, $\frac{\text{Unique bytes}}{\text{Reuse counter}}$, $\frac{\text{Lines}}{\text{Reuse counter}}$, $\frac{\text{Unique lines}}{\text{Reuse counter}}$.
- **Allocation related features** The size of the allocated buffer for the output buffer of this statement. The number of allocations.
- **Other features** The number of outer loops. The product of the lengths of outer loops. The value of the "auto_unroll_max_step"' specified by the pragma in outer loops.