Accelerated Auto-Tuning of GPU Kernels for Tensor Computations

Chendi Li[*]
chendi.li@utah.edu
University of Utah
Salt Lake City, Utah, USA

Yufan Xu[*]
yf.xu@utah.edu
University of Utah
Salt Lake City, Utah, USA

Sina Mahdipour Saravani
sina@cs.utah.edu
University of Utah
Salt Lake City, Utah, USA

P. Sadayappan
saday@cs.utah.edu
University of Utah
Salt Lake City, Utah, USA

ABSTRACT

TVM is a state-of-the-art auto-tuning compiler for the synthesis of high-performance implementations of tensor computations. However, an extensive search in the vast design space via thousands of compile-execute trials is often needed to identify high-performance code versions, leading to high auto-tuning time. This paper develops new performance modeling and design space exploration strategies to accelerate the code optimization process within TVM. Experimental evaluation on a number of matrix-matrix multiplication and 2D convolution kernels demonstrates about an order-of-magnitude improvement in auto-tuning time to achieve the same level of code performance.

CCS CONCEPTS

• **Software and its engineering → Compilers**; • **Computing methodologies → Parallel computing methodologies**; **Model development and analysis**; **Neural networks**.

KEYWORDS

Auto-tuning, Design space exploration, GPU kernel optimization, Neural networks, Performance modeling, Tile-size optimization

ACM Reference Format:
Chendi Li, Yufan Xu, Sina Mahdipour Saravani, and P. Sadayappan. 2024. Accelerated Auto-Tuning of GPU Kernels for Tensor Computations. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650200.3656626

1 INTRODUCTION

TVM [8] is a state-of-the-art auto-tuning compiler for the synthesis of high-performance implementations of tensor computations for both CPU and GPU platforms. Starting from a high-level representation of a tensor operator specification, a multi-level tiled parallel loop code is synthesized, and an auto-tuning search is performed in the huge configuration space of possible tile sizes and other mapping parameters. The performance of the layer-specific customized kernels synthesized by TVM/Ansor is impressive. Fig. 1 shows examples of performance improvement over Nvidia's cuDNN library for some convolution layers from Resnet-50 [12], on two different GPUs – an Nvidia RTX 3090 and an Nvidia RTX 4090.

However, in order to generate a high-performance code version, a significant number of compile-and-execute explorations are generally required. It is recommended [28] that 1000 measurements be used for each auto-tuned layer of a DNN, which generally takes between 15-30 minutes of wallclock time. Further, even with 1000 measurements, there can be considerable variability in the achieved performance of the best kernel configuration found via auto-tuning search because of the non-deterministic nature of the exploration of candidate configurations in the auto-tuning process. Fig. 2 shows scatter-plots for performance versus trials for two TVM/Ansor auto-tuning runs on one of the Resnet convolution layers from Fig. 1. As can be seen, the performance of the evaluated candidate configurations varies quite significantly from sample to sample for each run and the highest achieved performance across the two runs differs by around 20%. This suggests that more than 1000 trials or multiple runs of 1000 trials may be needed to increase confidence that the highest possible performance has been achieved.

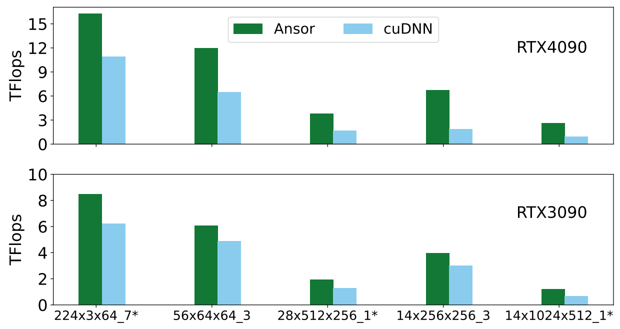

Figure 1: TVM/Ansor can generate high performance customized layer-specific kernels that outperform cuDNN.

[*]Both authors contributed equally to this research.

ICS '24, June 04–07, 2024, Kyoto, Japan
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0610-3/24/06
https://doi.org/10.1145/3650200.3656626

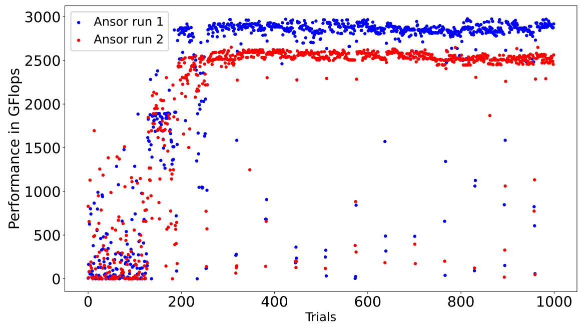

Figure 2: TVM/Ansor: Performance variability across runs.

Some recent efforts have developed strategies to reduce auto-tuning time compared to TVM [11, 18, 33], but none, to our knowledge, have demonstrated comparable code performance to TVM/Ansor across a wide range of tensor computations. In this paper, we address the following challenge: **Can significant reduction in the number of auto-tuning measurements be achieved over the state-of-the-art TVM/Ansor system without sacrificing performance?**

We successfully address the challenge, achieving comparable performance of the generated kernels with an order-of-magnitude reduction in the number of auto-tuning measurements compared to TVM/Ansor. Our developments have been conducted in the TVM/Ansor framework and have been made publicly available.[1].

The paper makes the following contributions:

- **New Features for Proxy Performance Model:** TVM/Ansor uses an XGBoost regression model for predicting the performance of candidate code configurations, using a large feature vector of 164 components derived from the code's AST (Abstract Syntax Tree). We develop an alternate set of 7 features that are based on analytical modeling of metrics such as data reuse at the shared-memory and register levels, instruction-level and warp-level concurrency, and estimates of load-imbalance across SMs (Streaming Multiprocessors)[2]. The development and experimental evaluation of the effectiveness of an XGBoost regression model using these analytical features is described in Sec. 4.
- **New Dynamic Search Space Exploration Strategy:** TVM/Ansor uses a Genetic Algorithm to explore the vast design space of schedules using the proxy performance model to identify a small number of the most promising candidates to generate code and compile and execute it on the target platform. We develop an alternate search strategy based on a dynamic gradient descent in a multi-dimensional space of the tile-size vector for code configurations. The development and experimental evaluation of effectiveness of our new search space exploration strategy is discussed in Sec. 5.
- **Demonstration of Order-of-Magnitude Speedup over TVM/Ansor Auto-tuning:** By combining the new analytical features for the proxy performance model and the new dynamic gradient descent search space exploration strategy, we demonstrate that comparable code performance can be achieved in 1-2 minutes, compared to 10-20 minutes for 1000 trials with TVM/Ansor, on a range of benchmarks for different sizes of matrix multiplication from transformers and all convolution operators from two CNN pipelines (ResNet [12] and Yolo9000 [17]). These results are presented in Sec. 6.

[1]https://github.com/HPCRL/Ansor-AF-DS

[2]The developed approaches are also applicable to GPUs from other vendors, but our experiments use Nvidia GPUs and we use CUDA terminology in the paper.

2 OVERVIEW OF PAPER

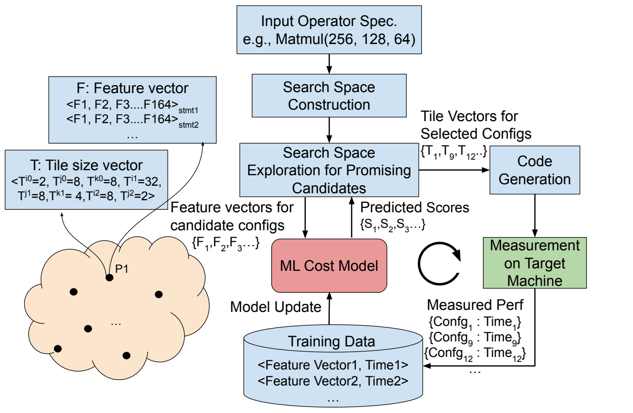
Input Operator Spec.
e.g., Matmul(256, 128, 64)
Search Space Construction
F: Feature vector
<F1, F2, F3....F164>stmt1
<F1, F2, F3....F164>stmt2
…
T: Tile size vector
$<T^{i0}=2, T^{j0}=8, T^{k0}=8, T^{i1}=32, T^{j1}=8, T^{k1}=4, T^{i2}=8, T^{j2}=2>$
Search Space Exploration for Promising Candidates
Tile Vectors for Selected Configs
$\{T_1, T_9, T_{12}..\}$
Code Generation
Feature vectors for candidate configs
$\{F_1, F_2, F_3...\}$
Predicted Scores
$\{S_1, S_2, S_3...\}$
P1
ML Cost Model
Measurement on Target Machine
Model Update
Measured Perf
$\{Confg_1 : Time_1\}$
$\{Confg_9 : Time_9\}$
$\{Confg_{12} : Time_{12}\}$
Training Data
<Feature Vector1, Time1>
<Feature Vector2, Time2>
…

Figure 3: Overview of GPU kernel optimization via TVM/Ansor auto-tuning.

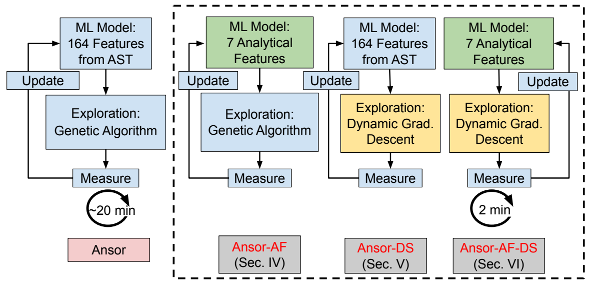
ML Model: 164 Features from AST
Update
Exploration: Genetic Algorithm
Measure
~20 min
Ansor
ML Model: 7 Analytical Features
Update
Exploration: Genetic Algorithm
Measure
Ansor-AF (Sec. IV)
ML Model: 164 Features from AST
Update
Exploration: Dynamic Grad. Descent
Measure
Ansor-DS (Sec. V)
ML Model: 7 Analytical Features
Update
Exploration: Dynamic Grad. Descent
Measure
2 min
Ansor-AF-DS (Sec. VI)

Figure 4: Overview of our developments within Ansor/TVM.

In this section, we provide a high-level overview of the work described in this paper. Fig. 3 shows the workflow for GPU kernel optimization in TVM/Ansor (also called TVM Auto-scheduler, after the incorporation of the Ansor [29] auto-tuner within the Apache TVM framework [1]). For an input operator specification (e.g., matrix-matrix multiplication computation expressed in an abstract high-level form, with specified sizes of the matrix operands), a search space with parametric multi-level tiles is created. Each configuration in this search space (e.g., P1 in the figure) is characterized by a tile-size vector *T* and also by a feature vector *F*. Ansor generates a 164-component feature vector extracted from the AST (Abstract Syntax Tree) for the multi-level tiled schema that includes statements representing the movement of data between global and shared memory and between shared memory to registers and computational statements for performing the arithmetic operations. The feature vector *F* is used for training an online Machine Learning model (XGBoost [7]) that assists the exploration of the search space by providing a predicted performance for any queried candidate configuration. The auto-tuning search proceeds iteratively over a number of batches – typically 64 configurations. The search strategy used by Ansor is a Genetic Algorithm where each member of a population represents one code configuration. The proxy ML model is used to evaluate the fitness of the members of the population. After evolution over a few generations, a set of the fittest members is identified and GPU code is generated, compiled, and executed for each of them. These measurements are used to retrain the ML model with a larger training set that keeps growing with each batch of execution.

Fig. 4 summarizes our developments within the TVM/Ansor framework to accelerate auto-tuning. Starting with the baseline Ansor implementation, we first attempt to improve the accuracy/effectiveness of the proxy performance model by developing features based on analytical modeling of three key factors that affect performance: i) data movement (both between global memory and shared memory and between shared-memory and registers), ii) concurrency/occupancy (modeling both Instruction-Level Parallelism and Warp-Level Parallelism), and iii) load-imbalance between the Streaming Multiprocessors. We use the label Ansor-AF to refer to the version of Ansor that uses these (seven) features instead of the original 164 features for the XGBoost ML performance model. Details on the features and an experimental evaluation of the effectiveness of Ansor-AF are presented in Sec. 4.

We next develop a new search strategy based on dynamic gradient descent in the multi-dimensional tile-space. We replace Ansor's GA-based search-space exploration with a new exploration strategy where the proxy ML model is used to predict the performance of neighbor points in the tile space of valid configurations. Neighbor configurations with the best-predicted performance are compiled and executed, and if any of them performs better, the locus of search moves to the best-performing neighbor. This variant of Ansor is called Ansor-DS. Details on the dynamic gradient search algorithm and the experimental evaluation of Ansor-DS are discussed in Sec. 5.

Both of the above strategies are then combined to create a version of Ansor labeled Ansor-AF-DS, which is evaluated on two different GPU platforms – Nvidia 3090 and Nvidia 4090. The details of the evaluation are provided in Sec. 6.

3 BACKGROUND ON GPU KERNEL OPTIMIZATION BY TVM/ANSOR AUTO-TUNING

3.1 Ansor kernel template and search space

The kernel template generated by Ansor is represented as a nested loop code schema with symbolic loop extents by using a set of predefined rules. Ansor constructs the iteration space of the specified computation and applies multi-level tiling to form the kernel search space for the GPU. We use the matrix multiplication kernel template shown in Listing 1 to explain Ansor's search space for GPU kernel. The same process generalizes to other tensor computations such as convolution. In Listing 1, the non-reduction loops **I** and **J** are each subjected to a 4-level tiling transformation to form a 5-dimensional search space for each iterator, as shown in the Listing 2. The product of the 5 loop extents for the tiling loops corresponding to any iteration dimension must be equal to the untiled problem size. To match the execution model of the GPU, the template generated by Ansor has outer loop bands that map to the grid-level, thread-block-level, and thread-level mapping, respectively. These loop bands can be executed in a fully parallel manner without any dependencies.

```
 1 // blockIdx.x i.0@j.0@ (None)
 2 // threadIdx.x i.2@j.2@ (None)
 3   for i.0 (None)
 4     for j.0 (None)
 5       for i.1 (None)
 6         for j.1 (None)
 7           for i.2 (None)
 8             for j.2 (None)
 9               // thread level code Line 10 - 25
10               for k.0 (None)
11                 // shared memory buffer loading
12                 // Line 11 - 13
13                 B.shared[...] = B[...]
14                 A.shared[...] = A[...]
15                 __syncthreads();
16                 // register level
17                 for k.1 (None)
18                   for i.3 (None)
19                     for j.3 (None)
20                       for k.2 (None)
21                         for i.4 (None)
22                           for j.4 (None)
23                             matmul.local = ...
24               // store output to global memory
25                matmul[...] = matmul.local[...]
```

Listing 1: Ansor kernel template for Matrix Multiplication.

The code generation eventually collapses the multi-dimensional grid level and thread block level iterators to create a 1D linearized grid shape and thread block shape.

Deep learning operators like convolution and matrix multiplication in transformer layers can exhibit a high degree of data reuse depending on data placement in the memory hierarchy. Ansor's schema exploits the data reuse potential of the operator and places data efficiently in GPU shared memory while keeping slices of the output tensors stationary in GPU registers. After the application of three-level tiling on the reduction iteration, the loop **K** is split into two loop groups. The outer loop k_0 at line 10 in the Listing 1 represents a sequential loop in each thread's code and controls the sequencing of shared-memory-level caching and register-level computation. The remaining innermost two loops for all iteration $i_3, i_4, j_3, j_4, k_1, k_2$ are used for register-level tiling, and a choice of loop permutations via degenerate unit tile sizes (explained later). Auto-tuning searches over tile size parameters in the schema. Since the search space is extremely large, a dynamically constructed machine learning model is used to limit the total number of instances of code generated and compiled/executed during the auto-tuning process.

$$I \longrightarrow \langle i_0, i_1, i_2, i_3, i_4 \rangle, where \prod_{n=0}^{4} i_n = I$$

$$J \longrightarrow \langle j_0, j_1, j_2, j_3, j_4 \rangle, where \prod_{n=0}^{4} j_n = J$$

$$K \longrightarrow \langle k_0, k_1, k_2 \rangle, where \prod_{n=0}^{2} k_n = K$$

Listing 2: Search space of matrix multiplication GPU kernel in Ansor.

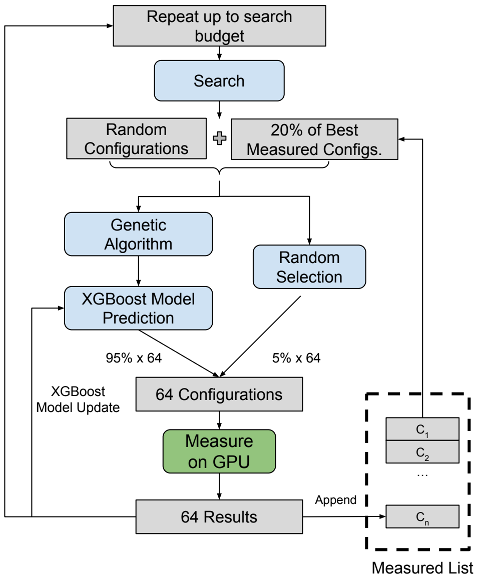

Figure 5: Ansor's search algorithm.

3.2 Search algorithm in Ansor

In the above two subsections, we described the configuration search space associated with the kernel template in Ansor [29]. In this subsection, we describe the search-space exploration algorithm in Ansor. Figure5 represents Ansor's search algorithm as a flow graph. In Ansor's tuning process, the search starts from randomly sampled initial code configurations. Ansor uses a genetic algorithm to generate a new batch of configurations from a set of explored and measured code configurations from the current search process with certain probabilities. It randomly mutates one or more parts of the existing code configuration by changing values like tile size or unroll factors. During the search process, the machine learning model (XGBoost [7]) is periodically updated with performance data from measured code configurations as training inputs. The machine learning model evaluates numerous newly generated code configurations generated by the genetic algorithm and selects highly scored candidates to form the next batch for execution. To avoid getting stuck at local minima, Ansor chooses 5% of configurations randomly and 95% from the group with high predicted performance. The new code configurations are executed on the GPU, and by default, the number of measurements per batch is 64. The search process iteratively generates and measures code, and updates the prediction model.

Ansor extracts training features from concrete configuration candidates based on the pre-generated template. Each statement in Ansor's configuration is represented as an n-dimensional feature vector. There are 164 features in a feature vector, categorized into five general groups, including computation operation counts, memory footprints, buffer allocations, etc. Ansor accumulates scores from each statement in a code configuration and uses the accumulated 164-component vector as the feature vector associated with the code configuration.

4 ANSOR-AF: ML PERFORMANCE MODELING WITH ANALYTICAL FEATURES

In this section, we present the details of our feature selection for performance modeling. Instead of the large set of 164 features extracted from the AST (Abstract Syntax Tree) of the multi-level tiled code schema by Ansor for training an XGBoost regression model, we analytically generate a very small number (seven) of features, which represent important factors that affect the performance of GPU kernels.

4.1 Analytical performance modeling features

The set of features were selected to cover three key factors that have a significant impact on GPU kernel performance:

4.1.1 **Data Movement.** The aggregate bandwidth between global memory to GPU cores is orders of magnitude lower than the peak arithmetic throughput of GPUs and the latency is in the order of hundreds of clock cycles. Three of the analytical metrics pertain to data movement, computed as *Operational Intensities (OI)*, ratios of the total number of arithmetic operations to the number of data-movement transactions at different levels of the memory hierarchy.

- **OI_Global_Mem**: OI for data movement from global memory to shared-memory. The total number of operations is computed as a product of trip-counts of loops surrounding the compute statements. The number of global memory read transactions is estimated by multiplying the sizes of shared-memory buffers for the input arrays by the product of trip-counts of loops surrounding the data buffering instructions in the TVM IR.
- **OI_Shared_Mem**: OI for data movement from shared memory to registers. The total number of shared memory transactions is accumulated for all launched thread blocks in the kernel. When the threads in a warp load from shared memory, multiple threads can request data from the same shared memory bank, resulting in bank conflicts. The degree of the bank conflicts in a warp proportionately increases the number of cycles needed to complete the warp-level load instruction. We estimate the degree of bank conflict from the stride of the data access by adjacent threads in a warp.
- **Reg_Reuse_Factor**: OI for data movement from registers to global memory. The GPU code generated by Ansor keeps elements of the output array stationary in thread registers and writes them out directly to global memory after all arithmetic operations have been performed on it.

4.1.2 **Concurrency.** A high degree of parallelism is essential for the high performance of GPU kernels to effectively overlap the high latency for data movement in the memory hierarchy for some instructions while arithmetic operations are concurrently executed for other instructions.

- **ILP**: Instruction Level Parallelism. This feature estimates the degree of intra-thread parallelism in the kernel. GPUs issue instructions without blocking if there is no dependence on any previous instruction. The load and compute operations on the different output elements in a register tile are completely independent, and thus, the sizes of the register tiles enable the estimation of ILP.
- **WLP**: Warp Level Parallelism. WLP represents the number of concurrently active warps in each SM (Streaming Multiprocessor)

Table 1: Configurations of matrix multiplication problem sizes from Bert [10, 21] (left), and 2D convolution operators from YOLO-9000 [17] (middle), and ResNet-50 [12] (right) ; F: # output channels; H, W: input image height and width; C: #input channels; R,S: kernel height and width; kernel stride = 1/2 (2 if * follows the kernel name, 1 otherwise).

Layer	M	N	K
M0	512	64	1024
M1	512	4096	1024
M2	512	64	768
M3	512	3072	768
M4	512	1024	4096
M5	512	768	3072

Layer	F	C	H/W	R/S
Y0	32	3	544	3
Y1	64	32	272	3
Y2	128	64	136	3
Y3	64	128	136	1
Y4	256	128	68	3
Y5	128	256	68	1
Y6	512	256	68	3
Y7	512	256	34	3
Y8	256	512	34	1
Y9	1024	512	17	3
Y10	512	1024	17	1

Layer	F	C	H/W	R/S
R0*	64	3	224	7
R1	64	64	56	1
R2	64	64	56	3
R3	256	64	56	1
R4*	128	256	56	1
R5	128	128	28	3
R6	512	128	28	1
R7*	256	512	28	1
R8	256	256	14	3
R9	1024	256	14	1
R10*	512	1024	14	1
R11	512	512	7	3
R12	2048	512	7	1

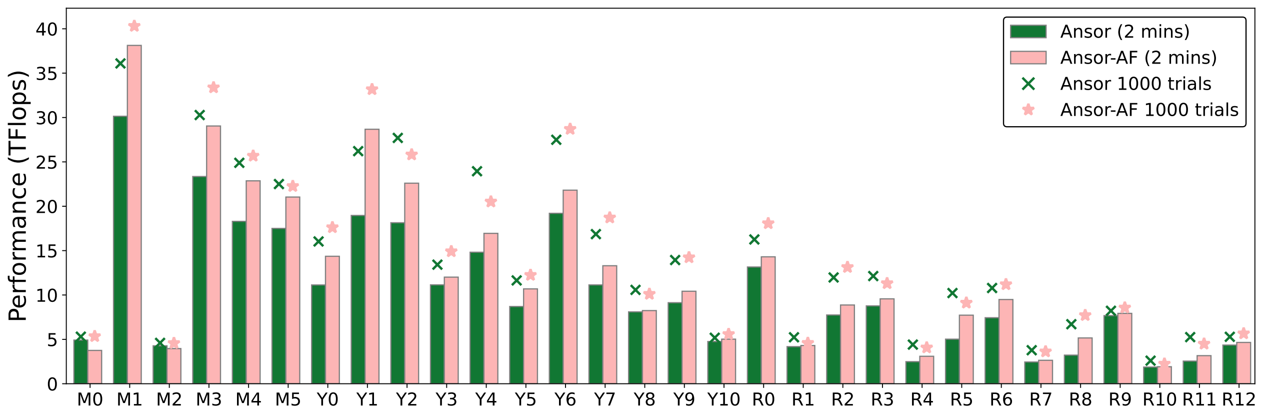

Figure 6: The effect of Analytical Features performance prediction model to Ansor (Ansor-AF), compared to the default Ansor.

in the GPU. It is primarily determined by resource constraints – the register usage per thread and the shared-memory usage per thread block. Resource usage is estimated by knowledge of the register-level and shared-memory buffer sizes for all three arrays, which are obtained as products of appropriate tile sizes.

- **Estimated_Occupancy**: the ratio of the actual number of active warps to the upper limit on the number of concurrently loadable warps per SM allowed by the hardware. The number of active warps is estimated from register and shared-memory usage.

4.1.3 **Load Imbalance.** The final analytically estimated feature estimates SM efficiency with respect to *tail effects*, where only a subset of SM's are active towards the end of the computation because the total number of thread blocks to be executed is not a perfect multiple of the number of SM's.

- **Wave_Efficiency** estimates the utilization ratio of GPU Streaming Multiprocessors (SMs). If a GPU has 128 Streaming Multiprocessors (SMs), and the chosen code configuration requires 320 thread blocks, assuming equal time for execution of each thread block and that only one thread block can be concurrently loaded to an SM, the entire execution will require three *waves*. The first 256 thread blocks would fully utilize the hardware units and form two full waves, but in the last wave only 64 SMs are active and the other half are idle. Since 3 waves are executed but only 2.5 waves represent the useful work, the *Wave_Efficiency* is computed as the ratio $\frac{2.5}{3} = 0.83$. A higher wave efficiency results in better utilization of GPU resources.

The kernel performance prediction model is a regression model based on the above seven analytical features.

$$Regessor(OI\_Global\_Mem, OI\_Shared\_Mem, Reg\_Reuse\_Factor, ILP, WLP, Estimated\_Occupancy, Wave\_Efficiency) \longrightarrow Perf$$

For each explored candidate code configuration, the above seven analytical features are computed from tile size information in the TVM IR. These features are used with the online XGBoost [7] decision tree model used by Ansor. Thus, to create Ansor-AF, we simply replace the default 164-sized feature vector computed by Ansor by these seven analytical features

4.2 Performance model evaluation

In this section, we present the results of experiments to perform an evaluation of the effectiveness of our seven analytical features in comparison with the default 164 features used by Ansor. We replaced Ansor's default 164 features for the XGBoost model by the seven analytical features, making no other changes to Ansor. We call this system Ansor-AF. Table 1 provides details on the 30 benchmarks used. The Matrix multiplication operator sizes (benchmarks M0-M5) are selected from Bert-BASE and Bert-LARGE models [10], the Conv2d operator sizes represent all distinct shapes from YOLO-9000 [17] (benchmarks Y0-Y10) and ResNet-50 [12] (benchmarks R0-R12).

Fig 6 contrasts the achieved performance (TFLOPs) for the best code generated by default Ansor with that achieved by Ansor-AF. All data in Fig 6 represents the mean from three independent runs. For each benchmark, we show the best performance achieved by Ansor and Ansor-AF in 2 minutes of wall clock time for tuning (the green and pink bars, respectively), as well as the best achieved performance after 1000 trials (green cross for Ansor and pink star for Ansor-AF). We make the following observations:

- For the vast majority of benchmarks (with the exception of M0, M2, and R10), a proxy model using the new analytical features (pink bar for Ansor-AF) finds better code versions at 2 minutes of wallclock tuning time than default Ansor (green bar).
- For the majority of the benchmarks (18 out of 30), the performance after 1000 trials for Ansor-AF (pink star) is better than Ansor (green cross).
- Overall, although the analytical features provide benefits over the default features used by Ansor, they are insufficient to achieve the goal of significantly reduced TVM auto-tuning time without significant loss of code performance. If we compare the performance of the best code generated by Ansor-AF in 2 minutes with that produced by Ansor with 1000 trials, higher performance is achieved on only two benchmarks, while performance is comparable (within 5%) or better for only 5 out of the 30 benchmarks.

5 ANSOR-DS: DYNAMIC GRADIENT DESCENT SEARCH SPACE EXPLORATION

In this section, we introduce a dynamic search algorithm to explore the configuration space in Ansor, as an alternative to its Genetic Algorithm based exploration strategy.

5.1 Search space construction

We use the matrix multiplication example in Listing 7a to explain how the search space is constructed. The search space of the example with a fixed loop order can be represented as a 5-dimensional vector $\langle i_0, j_0, k, i_1, j_1 \rangle$. The tile size of each dimension in the search space is a factor of the original problem size along that dimension. For each iteration-space dimension, the product of the tile size values at all levels must be equal to the problem size along that dimension. In the example, both i and j loops are split into two loops, and we have $\{Ti_0 \times Ti_1 == I; Tj_0 \times Tj_1 == J\}$. The tile sizes should be selected such that hardware capacity constraints are not violated, and we prune out such infeasible configurations in our search space. To enable loop permutation, the Ansor schema uses an additional level of tiling with degenerate tile size for one iterator in the search space. For example, in Listing 7b, the search space expands to a 7-dimensional vector with two additional loops under loop k as $\langle i_0, j_0, k, i_{1x}, j_{1x}, i_{1y}, j_{1y} \rangle$. We can achieve loop permutation among inner loops i_1, j_1 by setting the tile size to 1 for one of the two loop iterators along a dimension in $\langle i_{1x}, j_{1x}, i_{1y}, j_{1y} \rangle$. The degenerate loop with unit extent gets removed during code generation, thus effectively enabling loop permutation via tile size variation.

In the multi-dimensional search space, each dimension iterator is a perfect divisor of the extent along the corresponding dimension of the iteration space. We define a **hop** on one iterator in the search space to be a change from the current value to either the next possible larger or smaller value in the factor list. If the extent of I is 32, the iterator i is a perfect divisor of the iteration space of 32, i.e., one of (1,2,4,8,16,32). When the value of iterator i equals 8, the value can change to 4 or 16 in one hop. We use the number of hops to represent the number of iterators changed. In Figure 8, we use a 3D search space $\langle i, j, k \rangle$ to visualize the hop and distance of the hop. The base node is shown as a gray node and all red arrows represent *1-hop* distances from the base node. The green arrow represents a *2-hop* distance from the base node, with a total of 12 candidates in this 3D search space. There are eight *3-hop* nodes in this space, one being shown with a blue arrow.

5.2 Dynamic search algorithm

We propose a search algorithm that uses dynamic gradient descent (DGD) based on light online measurements. Our search algorithm, **DGD_Search**, described in Figure 9a, is given a search budget and a dimension hop threshold for search in the configuration space. An initial performance predictor model is built based on 64 random code configurations in the space at Line 3. The search algorithm explores candidates at a given distance (within n hops) and uses the prediction model to estimate the runtime performance of all such neighbor configurations. The proxy performance model ranks candidate configurations in order of the predicted score at Lines 11-13. The performance prediction model is updated at Line 20 after calling the DGD_Move function at Line 14 in DGD_Search, and the newly measured configurations with time information are used as model training data. This process is repeated as long as a better-measured configuration emerges during the search process under a given hop distance threshold. When a local minimum is reached, i.e., no neighbor with lower measured performance is found, search starts at a new random start point. Finally, the search ends when the specified budget of total number of measurements is exhausted.

The **DGD_Move** function in Figure 9b decides where to move, by using a *sliding window* size of three. The three candidates with the highest predicted scores are measured on the target device. If the best performance among the measured candidates exceeds the current point's performance, the locus of search moves to that point and repeats the n-hops candidates' process from the new point. To avoid exploring low-performance candidates, we use two early-stop criteria: i) a minimum threshold for model-predicted scores at Line 8, and ii) a measurement-based threshold at Line 20.

Using gradient descent with exhaustive measurement of all potential n-hops candidates can eventually find a very high-performance code configuration, but is infeasible to explore in a reasonable time

```
for(i0 = 0; i0 < I; i0+=Ti)
  for(j0 = 0; j0 < J; j0+=Tj)
    for(k = 0; k < K; k++)
      for(i1 = i0;  i1 < i0+Ti; i1++)
        for(j1 = j0; j1 < j0+Tj; j1++)
          Out[i1][j1] += A[i1][k]*B[k][j1]
```

(a)

```
for(i0 = 0; i0 < I; i0+=Ti)
  for(j0 = 0; j0 < J; j0+=Tj)
    for(k = 0; k < K; k++)
      for(i1x = i0;  i1x < ?; i1x++)
        for(j1x = j0; j1x < ?; j1x++)
          for(i1y = i0;  i1y < ?; i1y++)
            for(j1y = j0; j1y < ?; j1y++)
              Out[i1][j1] += A[i1][k]*B[k][j1]
```

(b)

Figure 7: a) Five dimensional search space for tiled matrix multiplication. b) Loop permutation via tile size choice in a higher dimensional space.

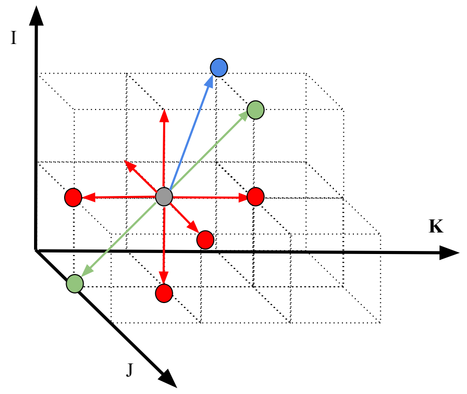

Figure 8: Illustration of tile space and 1-hop, 2-hop, 3-hop neighbors.

frame. Code generation, compilation and measurement is the most time-consuming component in the existing auto-tuning framework. The dynamic gradient descent with performance modeling can significantly reduce the number of code measurements during the search. There are multiple reasons to explore only a limited number of the n-hops candidates in an iterative manner (we use 3 as the default value in our algorithm). We use the same example from the Listing 7a to motivate the heuristics in our search algorithm. Exploring only 1-hop candidates of a 5 dimensional configuration $\langle i_0, j_0, k, i_1, j_1\rangle$ results in many infeasible configurations or those that violates hardware constraints. For example, simply changing one value in the configuration vector would result in the inner tile sizes being larger than the outer tile sizes at the same dimension, which produces an invalid code during the code generation. Therefore, changing two or more dimensions of tile size values simultaneously produces more valid candidates and improves the probability of moving to better points with lower execution time in the search space. However, the number of candidates increases rapidly when we explore 4-hop or higher dimensional neighbor candidates. The model prediction time is proportional to the number of evaluated code candidates, and longer feature extraction time is also unavoidable. The total number of distinct n-hop candidates in an m-dimensional subspace is $\binom{m}{n} \times 2^n$. The number of 1-hop candidates in the above 5-dimensional space is 10, and it has 40 unique 2-hop candidates and 160 3-hop candidates.

Figure 10 presents an example of the dynamic gradient descent search with a sliding window size of two for exploring $1 - hop$ nodes. Every point represents a code configuration in the search space, and each configuration is associated with a model-predicted score between 0 to 1 (higher is better, shown in orange color) and a measured performance of execution time (lower is better, shown in green color). In the figure, the start point is marked in grey and has a measured performance value (0.9 in green color). The algorithm uses a surrogate performance predicting model to estimate all $1 - hop$ nodes and selects the highest two values (orange color) as measurement candidates via a sliding window. In this example, the top two candidates (Sliding Window 1, with predicted scores of 0.78 and 0.75) are compiled, executed and measured. It turns out that both have higher execution times (1.2 and 1.1, shown in green). So the sliding window is moved over the next two configurations in the sorted list (Sliding Window 2, with predicted scores of 0.65 and 0.64). These two configurations are measured, and now one of them (predicted score of 0.65, measured time of 0.8) is better than the current point, resulting in the locus of search moving to that point. To reduce clutter, we only show details at all neighbors for the initial configuration (gray point) but only show the predicted score and measured time for points in the gradient descent path. The search terminates when a local minimum is reached (predicted score of 0.92, measured time of 0.3).

5.3 Evaluation of Effectiveness of Dynamic Search Algorithm

Ansor's design space exploration strategy is based on the use of a genetic algorithm (GA). In order to assess the effectiveness of our alternate gradient-descent search strategy, we designed a second experimental setup where we incorporated our search algorithm within the Ansor auto-tuning platform, but retained Ansor's default features for the XGBoost proxy performance model. We call this system Ansor-DS.

Fig. 11 presents experimental results comparing Ansor-DS against default Ansor. As was done previously in comparing Ansor-AF with Ansor (Fig. 6 in Sec. 4), we again show the best achieved tuned code performance after 2 minutes of auto-tuning, as well as after 1000 trials, for both Ansor and Ansor-DS. From the figure, we may observe:

```
def DGD_Search(search_budget, hop_threshold):
 # constrcut the init model
 model.init(sample_size=64)
 global best_measured = 0
 while i < search_budget:
   cur_point = random_sample_point()
   while cur_point != null:
     n = 1
     meas_list.clear()
     do:
       points = n_hop_cands(cur_point, n)
       scores = model.predict(points)
       sort(scores)
       cur_point, meas_list = DGD_Move(points,
                                        scores)
       n += 1
     while (n < hop_threshold
            and cur_point == null)
     # update model with measured code
     model.update(meas_list)
   # start a new point after hiting local min
   i += 1
return
```

(a)

```
def DGD_Move(points, scores, best_measured):
 base_score = scores[0]
 score_threshold = base_score * 0.6
 measure_threshold = best_measured * 0.6
 slide = 3 # slide window size for measurement
 i = 1
 meas_list = []
 while (all(scores[i: i+wd]) >= score_threshold
        and i+wd < points.size):
   local_meas_list = measure(points[i:i+wd])
   if max(local_meas_list) > base_performance:
       next_base = local_meas_list[max_index]
       meas_list.append(local_meas_list)
       break
   else:
       meas_list.append(local_meas_list)

   best_measured = max(local_meas_list)
   # early stop
   if max(local_meas_list) < measure_threshold:
       break
   i += wd
 return next_base, meas_list
```

(b)

Figure 9: Dynamic gradient descent search algorithm with points movement in the space.

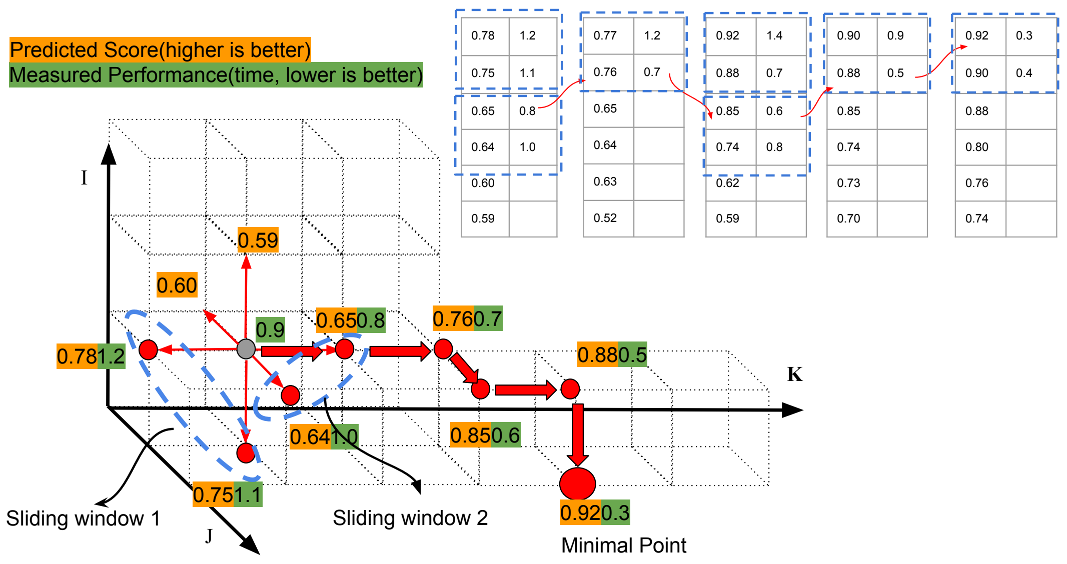

Figure 10: Illustration of dynamic gradient descent search

- For every benchmark, after 2 minutes of auto-tuning, the best achieved code performance by Ansor-DS (burgundy bars) is always comparable or better than Ansor (green bars).
- For 21 out of the 30 benchmarks, within 2 minutes of tuning, Ansor-DS finds a code version with comparable or better performance than Ansor after 1000 trials. Thus, the new dynamic gradient search can be very effective in reducing auto-tuning time in TVM/Ansor.
- If run to 1000 trials, Ansor-DS (burgundy star) achieves a performance improvement of more than 5% over Ansor (green cross) for 22 benchmarks, and comparable (within 5%) for 8 benchmarks.

6 ANSOR-AF-DS: EVALUATION OF INTEGRATED OPTIMIZATIONS

In this section, we present an evaluation of Ansor-AF-DS, integrating the two optimizations described in the previous two sections: i) replace TVM/Ansor's default 164 features by our seven analytical features (AF) for the XGBoost machine learning proxy model for

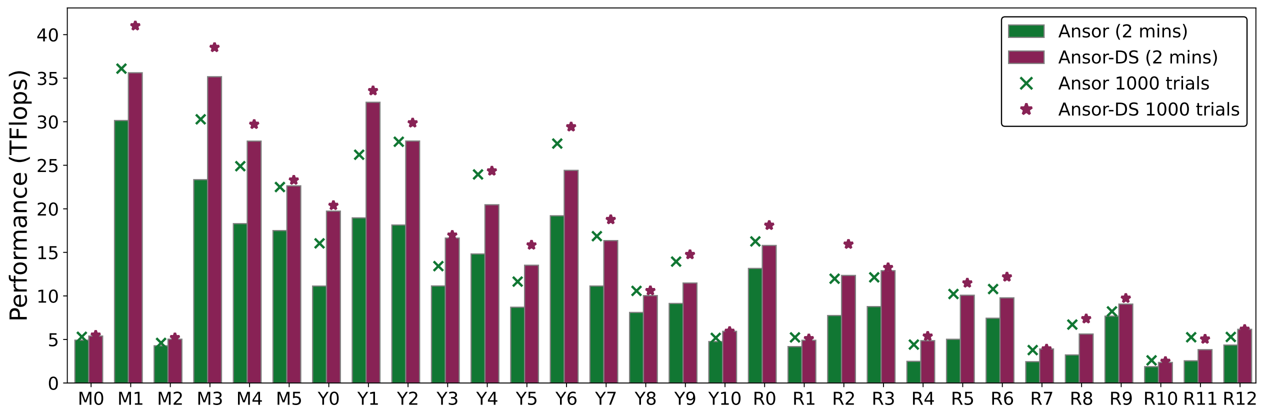

Figure 11: The effect of adding our dynamic search algorithm to Ansor (Ansor-DS), compared to the default Ansor with GA.

performance prediction, and ii) replace TVM/Ansor's Genetic Algorithm by our dynamic gradient descent search algorithm (DS) as the search-space exploration strategy.

6.1 Experimental Setup

For our evaluation of the integrated Ansor-AF-DS system, we carried out experiments on two NVIDIA GPUs – Nvidia RTX 3090 GPU with a 24-core Intel Core i9-13900K CPU, and Nvidia RTX 4090 GPU with a 16-core AMD Ryzen 9 7950X CPU. These two machines represent different GPU generations (Ampere and Ada Lovelace). Both machines use the Ubuntu 22.04 LTS operating system. In addition to Ansor [29], we also compared with Roller [33]. Ansor [29] was run with its default XGBoost [7] model for 1000 trials. For Roller [33], we ran it to select the Top50 candidates from its generated configurations. ROLLER uses analytical models based on architecture parameters. We chose hardware parameters for each machine following instructions at ROLLER official code repository [31, 32]. Three independent runs were executed for all measurements, and the arithmetic mean from the three measurements is reported. The Nvidia Nsight-Compute profiler was used to measure execution time for all code versions (Ansor baseline, Ansor-AF-DS, and ROLLER) on both GPU platforms.

6.2 Evaluation

Figure 12 presents the results of experimental evaluation on the two GPU platforms, comparing Ansor-AF-DS, Ansor [29], and Roller [33]. Each group of bars shows the achieved performance of the best code configurations found by each methodology (averaged over three runs). For Ansor and Ansor-AF-DS, stacked bars show the best achieved performance after auto-tuning for a wallclock time of 1 minute and 2 minutes, by the lighter and darker shades, respectively. The green cross marker (×) shows the best performance among all measured configurations after 1000 trials. Similarly, the red star marker (⋆) shows the best achieved performance of code versions explored by Ansor-AF-DS after 1000 trials. The Roller-Top-50 bars report the best performance among Roller's top 50 generated code candidates.

We make the following summary observations on the experimental data in Fig. 12.

- Although Roller generates kernel codes quite rapidly (the time taken for generating and evaluating the Top-50 configurations was typically around 20 seconds for matrix-multiplication benchmarks and around a minute for the convolution benchmarks), except for one benchmark (Y10), achieved performance is higher with one-minute of auto-tuning with Ansor-AF-DS.
- On the RTX 4090 (3090), for 27 (22) out of the 30 benchmarks, within 2 minutes of tuning, Ansor-AF-DS finds a code version with comparable or better performance than Ansor after 1000 trials.
- If run to 1000 trials, Ansor-AF-DS achieves a performance improvement of more than 5% over Ansor (green cross) for 25 (24) benchmarks, and comparable (within 5%) for 5 (6) benchmarks.
- Table 2 summarizes the overall effectiveness for the three groups of benchmarks (M0-M5, Y0-Y10, R0-R12) by presenting geometric means of the achieved speedup by the best kernels generated by Ansor-AF-DS (1-minute, 2-minutes, 1000-trials) over Ansor's best kernel (1000 trials), for the two GPU platforms. It may be seen that with 2 minutes of auto-tuning, on both GPUs, the kernels generated by Ansor-AF-DS achieve a speedup for all three groups of benchmarks over the best kernels generated by Ansor after 1000 trials.

Table 2: Geometric mean of speedups of best kernels generated by Ansor-AF-DS in 1 minute, 2 minutes, and after 1000 trials, versus Ansor's best kernel after 1000 trials, across the three benchmark groups.

	RTX 3090			RTX 4090		
Type	**1 Min**	**2 Min**	**1000 trials**	**1 Min**	**2 Min**	**1000 trials**
Matmul	1.01	1.04	1.09	0.99	1.04	1.13
Resnet	0.96	1.03	1.13	0.94	1.02	1.16
Yolo	0.97	1.00	1.09	1.01	1.11	1.22

We also evaluated the performance variability of the best kernels across multiple runs of Ansor-AF-DS compared to Ansor. For each

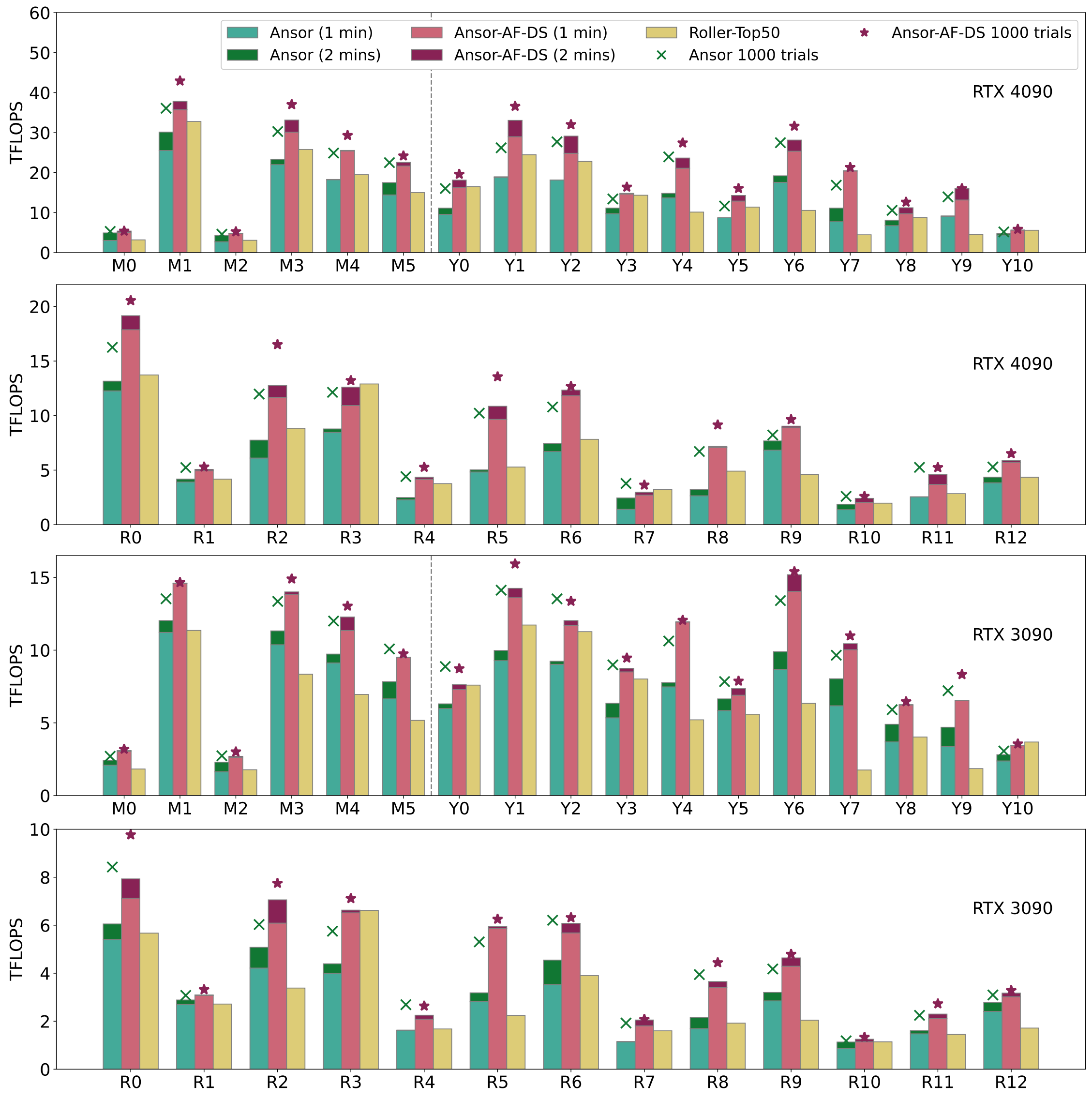

Figure 12: Performance comparison summary on RTX 4090 and RTX 3090 GPUs. The top two charts show the performance on RTX 4090, and the bottom two charts show the performance on RTX 3090.

benchmark, performance variability across multiple auto-tuning runs was computed as follows (Eq. 1):

$$Variability = \frac{\max(\text{Performance}) - \min(\text{Performance})}{\max(\text{Performance})} \tag{1}$$

Table 3 presents the geometric mean of the variability metrics across the benchmarks in the three groups (M0-M5, Y0-Y10, R0-R12).

Figure 13 presents details on the observed performance variability for each benchmark by displaying the minimum and maximum performance across three independent runs as error-bars around the mean value bars.

- On both GPUs, with very few exceptions, the performance variability with Ansor-AF-DS after 1000 trials (error bars around the orange mean-value bars) is quite low compared to Ansor after 1000 trials (green bars).
- The performance variability with Ansor-AF-DS after 2 minutes of auto-tuning is noticeably higher than Ansor-AF-DS after 1000 trials.
- The performance variability with Ansor-AF-DS after 2 minutes is sometimes worse (higher) than Ansor after 1000 trials, but it is more often lower.

7 RELATED WORK

Auto-tuning has been widely used for performance optimization. In general, auto-tuning involves the definition of a multi-dimensional design space of alternatives along with a search strategy within that space to find a high-performance configuration via execution

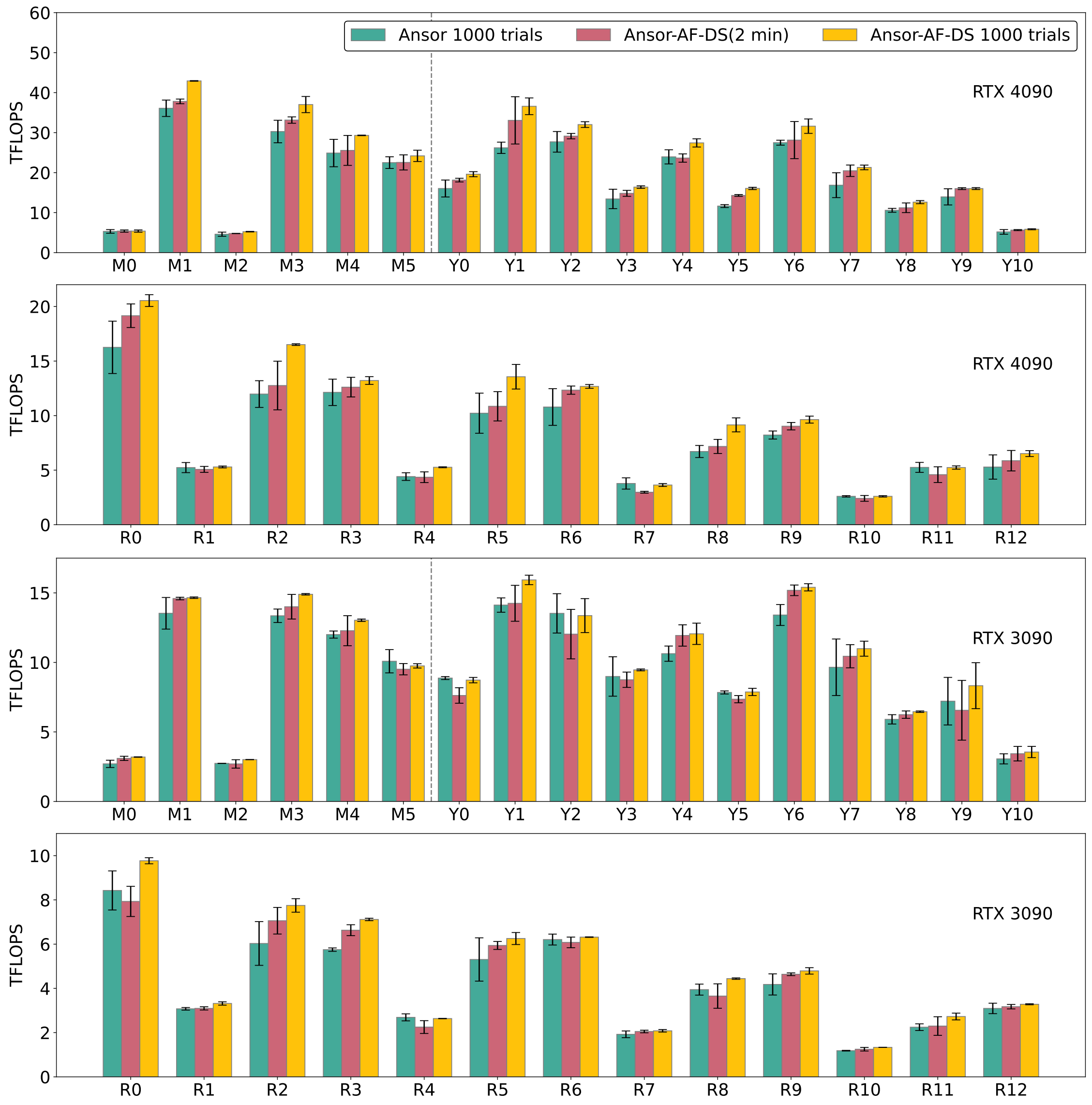

Figure 13: Performance variability summary on RTX 4090 and RTX 3090 GPUs. The top two charts show the performance on RTX 4090, and the bottom two charts show the performance on RTX 3090.

Table 3: Performance variability across runs for Ansor-AF-DS (2 min, 1000-trials) versus Ansor (1000-trials). The values in each row are geometric means over the benchmarks within each of the three groups.

	RTX 3090			RTX 4090		
	Ansor-AF-DS		Ansor	Ansor-AF-DS		Ansor
Type	**2 Min**	**1000 trials**	**1000 trials**	**2 Min**	**1000 trials**	**1000 trials**
Matmul	0.09	0.01	0.11	0.06	0.03	0.17
Resnet	0.10	0.02	0.10	0.14	0.05	0.18
Yolo	0.14	0.06	0.12	0.08	0.05	0.15

of a limited subset of points in the design space. Auto-tuning has been used extensively for performance optimization at the library level, compiler level, and application level [4]. This section discusses related work on auto-tuning at the compiler level.

One of the earliest efforts in compiler-centered auto-tuning was the CHiLL framework [19]. Focusing on the optimization of image processing pipelines, Halide [16] demonstrated the significant benefits of separated specification of the computation in an abstracted form and the use of a separate language for the specification of a space of possible schedules for the computation. TVM [8] built on the insights from Halide to develop a scheduling framework for tensor computations.

AutoTVM [8], the original auto-tuning framework of TVM before the integration of Ansor[29] (now called TVM AutoScheduler), pioneered the online-learning-based approach to optimize tensor programs [9]. Unlike the automated approach of Ansor, AutoTVM requires code-tuning templates provided by an expert to generate the auto-tuning search over tile size parameters. The performance of code generated by TVM/Ansor is generally comparable or better than AutoTVM. Due to the lower performance of the generated code and the lack of automation/generality of AutoTVM, we do not compare with it in this paper.

Tiramisu[3] and Tensor-Comprehensions[20] are two polyhedral compilers that also target the deep learning domain. Tiramisu provides a scheduling language, and uses auto-tuning to find the best tile sizes and unrolling factors for the code configuration on the target machine. Tensor Comprehensions combines auto-tuning with the PPCG [22] polyhedral code generator for GPUs. However, the performance of code generated by both these systems is generally lower than that generated by TVM/Ansor.

DOPpler [5] provides a parallel auto-tuning infrastructure to alleviate the time costs that are incurred in the auto-tuning process. This is done by performing a parallel execution enabled by an on-device measurement technique of the candidate tensor programs. DOPpler's contribution is in an orthogonal direction to ours: it reduces auto-tuning time without changing the number of measurements, while we reduce the number of measurements to reduce auto-tuning time, while maintaining comparable performance of generated code.

ROLLER [33] uses a recursive construction-based method to limit the number of candidate configurations by taking a tile abstraction view that aligns the tile shapes with features of the target accelerator, such as memory banks and size of schedulable unit, thus reducing the number of choices for the shapes. In terms of candidate evaluation, ROLLER uses a performance model based on micro measurements and analytical calculation of performance factors without full measurements of the program wall time, enabling rapid kernel generation. We present experimental results that compare against ROLLER.

Trimmer [6] focuses on detecting and removing low performance configuration candidates by preempting unpromising programs, and applying a fully connected neural network model to filter out inefficient tensor programs that are predicted to have high latency. At its core, it dynamically dedicates tuning time to operators based on their relative rate of latency improvement. Trimmer has been shown to achieve high energy cost reduction but very mild absolute performance gain.

One-Shot Tuner [18] uses an offline one-time trained Transformer neural network model to predict configuration performance instead of the XGBoost model in AutoTVM. They do not perform any online adaptation, a primary difference from our work. As they observe in their paper, they are unable to achieve comparable performance to TVM for later layers in CNN pipelines. In contrast, we are able to achieve comparable or better performance across all stages of CNN pipelines.

CNNOpt [25] uses hybrid analytical-ML modeling along with an offline trained model to achieve fast and high performance convolution GPU kernel generation. However, it is specialized to the convolutional GPU kernel, in contrast to the state-of-the-art TVM/Ansor auto-tuning framework, which supports all dense, deep learning operators. Our improvements over TVM/Ansor retain the generality in terms of the range of tensor operators that can be optimized.

Several general-purpose auto-tuning frameworks exist, such as BACO [13], OpenTuner [2], ytopt [23]. They implement search strategies to optimize high-dimensional function spaces via auto-tuning and can be used to build auto-tuning compilers by coupling a code generator and a compile/execute harness with them.

Several efforts [15, 24, 27, 30] have focused on identifying sequences of operators that can be profitably fused. The approach we develop can be extended to apply to tile-size optimization of fused kernels.

Some research efforts have sought to optimize the total time for TVM Auto-tuning of all the operators in a graph. Liu et al. [14] applied layout optimization for tensor passing through layers in Deep Learning and achieved overall inference time reduction. Other efforts [11, 26] are inspired by the idea of transfer learning and adapt such a method to locate repeated auto-schedules and reuse them among various tensor programs. The developments in this paper can be applied to auto-tuning of operator graphs.

8 CONCLUSION

In this paper we describe two primary contributions that together enable a significant reduction in auto-tuning time (an order of magnitude) with comparable or better performance of the generated GPU kernels by the state-of-the-art TVM/Ansor auto-tuning compiler: i) a new analytical-feature-based machine learning performance prediction model, and ii) a new dynamic gradient descent search-space exploration strategy. Experimental evaluation on two GPU platforms (Nvidia RTX 3090 and 4090) using a number of matrix multiplication operators from transformers and all convolution stages from two CNN pipelines demonstrate effectiveness in reducing the time for GPU kernel optimization in TVM/Ansor.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their suggestions that helped improve the paper. This work was supported in part by the U.S. National Science Foundation through award 2217154.

REFERENCES

[1] [n. d.]. Apache TVM: An End to End Machine Learning Compiler Framework for CPUs, GPUs and accelerators. https://tvm.apache.org/.

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman P. Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*. ACM, 303–316. https://doi.org/10.1145/2628071.2628092

[3] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019*. IEEE, 193–205. https://doi.org/10.1109/CGO.2019.8661197

[4] Prasanna Balaprakash, Jack J. Dongarra, Todd Gamblin, Mary W. Hall, Jeffrey K. Hollingsworth, Boyana Norris, and Richard W. Vuduc. 2018. Autotuning in High-Performance Computing Applications. *Proc. IEEE* 106, 11 (2018), 2068–2083. https://doi.org/10.1109/JPROC.2018.2841200

[5] Damian Borowiec, Gingfung Yeung, Adrian Friday, Richard Harper, and Peter Garraghan. 2023. DOPpler: Parallel Measurement Infrastructure for Auto-Tuning Deep Learning Tensor Programs. *IEEE Trans. Parallel Distributed Syst.* 34, 7 (2023), 2208–2220. https://doi.org/10.1109/TPDS.2023.3279233

[6] Damian Borowiec, Gingfung Yeung, Adrian Friday, Richard H. R. Harper, and Peter Garraghan. 2022. Trimmer: Cost-Efficient Deep Learning Auto-tuning for

Cloud Datacenters. In *IEEE 15th International Conference on Cloud Computing, CLOUD 2022, Barcelona, Spain, July 10-16, 2022*, Claudio Agostino Ardagna, Nimanthi L. Atukorala, Rajkumar Buyya, Carl K. Chang, Rong N. Chang, Ernesto Damiani, Gargi Banerjee Dasgupta, Fabrizio Gagliardi, Christoph Hagleitner, Dejan S. Milojicic, Tuan M. Hoang Trong, Robert Ward, Fatos Xhafa, and Jia Zhang (Eds.). IEEE, 374–384. https://doi.org/10.1109/CLOUD55607.2022.00061

[7] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016.* ACM, 785–794. https://doi.org/10.1145/2939672.2939785

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.* USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[9] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. (2018), 3393–3404. https://proceedings.neurips.cc/paper/2018/hash/8b5700012be65c9da25f49408d959ca0-Abstract.html

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. (2019), 4171–4186. https://doi.org/10.18653/V1/N19-1423

[11] Perry Gibson and José Cano. 2022. Transfer-Tuning: Reusing Auto-Schedules for Efficient Tensor Program Code Generation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT 2022, Chicago, Illinois, October 8-12, 2022.* ACM, 28–39. https://doi.org/10.1145/3559009.3569682

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016.* IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90

[13] Erik Orm Hellsten, Artur L. F. Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. 2023. BaCO: A Fast and Portable Bayesian Compiler Optimization Framework. (2023), 19–42. https://doi.org/10.1145/3623278.3624770

[14] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing CNN Model Inference on CPUs. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019.* USENIX Association, 1025–1040. https://www.usenix.org/conference/atc19/presentation/liu-yizhi

[15] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: Accelerating deep neural networks execution with advanced operator fusion. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021.* ACM, 883–898. https://doi.org/10.1145/3453483.3454083

[16] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 519–530. https://doi.org/10.1145/2491956.2462176

[17] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: Better, Faster, Stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017.* IEEE Computer Society, 6517–6525. https://doi.org/10.1109/CVPR.2017.690

[18] Jaehun Ryu, Eunhyeok Park, and Hyojin Sung. 2022. One-shot tuner for deep learning compilers. In *CC '22: 31st ACM SIGPLAN International Conference on Compiler Construction, Seoul, South Korea, April 2 - 3, 2022.* ACM, 89–103. https://doi.org/10.1145/3497776.3517774

[19] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary W. Hall, and Jeffrey K. Hollingsworth. 2009. A scalable auto-tuning framework for compiler optimization. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009.* IEEE, 1–12. https://doi.org/10.1109/IPDPS.2009.5161054

[20] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

[21] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA.* 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[22] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4 (2013), 54:1–54:23. https://doi.org/10.1145/2400682.2400713

[23] Xingfu Wu, Prasanna Balaprakash, Michael Kruse, Jaehoon Koo, Brice Videau, Paul D. Hovland, Valerie E. Taylor, Brad Geltz, Siddhartha Jana, and Mary W. Hall. 2023. ytopt: Autotuning Scientific Applications for Energy Efficiency at Large Scales. *CoRR* abs/2303.16245 (2023). https://doi.org/10.48550/ARXIV.2303.16245 arXiv:2303.16245

[24] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. 2022. Bolt: Bridging the Gap between Auto-tuners and Hardware-native Performance. (2022). https://proceedings.mlsys.org/paper/2022/hash/38b3eff8baf56627478ec76a704e9b52-Abstract.html

[25] Yufan Xu, Qiwei Yuan, Erik Curtis Barton, Rui Li, P. Sadayappan, and Aravind Sukumaran-Rajam. 2022. Effective Performance Modeling and Domain-Specific Compiler Optimization of CNNs for GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT 2022, Chicago, Illinois, October 8-12, 2022.* ACM, 252–264. https://doi.org/10.1145/3559009.3569674

[26] Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. 2023. TLP: A Deep Learning-Based Cost Model for Tensor Program Tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023.* ACM, 833–845. https://doi.org/10.1145/3575693.3575737

[27] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. 2022. Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization. (2022). https://proceedings.mlsys.org/paper/2022/hash/069059b7ef840f0c74a814ec9237b6ec-Abstract.html

[28] Lianmin Zheng. 2020. Optimizing Operators with Auto-scheduling. https://tvm.apache.org/docs/tutorial/auto_scheduler_matmul_x86.html#sphx-glr-tutorial-auto-scheduler-matmul-x86-py.

[29] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020.* USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

[30] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. 2022. AStitch: Enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022.* ACM, 359–373. https://doi.org/10.1145/3503222.3507723

[31] Hongyu Zhu. 2022. Get Started Tutorial: Generate a Matmul Kernel. https://github.com/microsoft/nnfusion/blob/a87b05c80c834db06aaa890d2ed4e364bc62eba4/artifacts/get_started_tutorial/README_GET_STARTED.md.

[32] Hongyu Zhu. 2022. Roller RTX 3090 config file. https://github.com/microsoft/nnfusion/blob/roller/artifacts/roller/arch/rtx3090.py.

[33] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. 2022. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022.* USENIX Association, 233–248. https://www.usenix.org/conference/osdi22/presentation/zhu