Explore as a Storm, Exploit as a Droplet: A Unified Search Technique for the Ansor Optimizer

MICHAEL CANESCHE, UFMG, Brazil
GAURAV VERMA, Stony Brook University, USA
FERNANDO MAGNO QUINTÃO PEREIRA, UFMG, Brazil

Machine-learning models consist of kernels, which are algorithms applying operations on tensors—data indexed by a linear combination of natural numbers. Examples of kernels include convolutions, transpositions, and vectorial products. There exist many ways to implement a kernel. These different implementations form the kernel’s optimization space. Kernel scheduling is the problem of finding the best implementation, given an objective function—typically execution speed. Kernel optimizers such as Ansor, Halide, and AutoTVM solve this problem via search heuristics, which combine two phases: exploration and exploitation. The first step evaluates many different kernel implementations in the search space. The latter tries to improve the best implementations, investigating similar kernels. As an example, Ansor combines kernel generation through sketches for exploration and leverages an evolutionary algorithm to exploit the best sketches. In this work, we demonstrate the potential to reduce Ansor’s search time while enhancing kernel quality by incorporating Droplet Search, an AutoTVM algorithm, into Ansor’s exploration phase. The approach involves limiting the number of samples explored by Ansor, selecting the best, and exploiting it with Droplet Search’s coordinate descent algorithm. Typically, by applying this approach to the first 300 kernels that Ansor generates, we obtain better kernels in less time than if we let Ansor analyze 10,000 kernels. This result has been consistently replicated in 20 well-known deep-learning models (AlexNet, ResNet, VGG, DenseNet, etc) running on three distinct architectures: an AMD Ryzen 7 (x86), an Nvidia A100 tensor core, and an ARM A64FX.

CCS Concepts:
• Software and its engineering → Runtime environments; Compilers.

Additional Key Words and Phrases: Tensor Compiler, Optimization, Kernel Scheduling, Search

ACM Reference Format:
Michael Canesche, Gaurav Verma, and Fernando Magno Quintão Pereira. 2024. Explore as a Storm, Exploit as a Droplet: A Unified Search Technique for the Ansor Optimizer. 1, 1 (February 2024), 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A kernel is an algorithm that applies operations on tensors: chunks of memory indexed by a linear combination of natural numbers. A Tensor Compiler is a compilation infrastructure that generates code for kernels. Since the same kernel can be implemented in many different ways, tensor compilers often face the challenge of solving a problem called kernel scheduling: determining a suitable ordering for the numerous operations a kernel performs on tensors. Kernel scheduling is typically addressed using heuristics because the Kernel Optimization Space—the set of all possible implementations of a kernel—is usually vast.
The Apache TVM tensor compiler employs two distinct optimizing infrastructures for solving kernel scheduling: AutoTVM [Chen et al. 2018] and Ansor [Zheng et al. 2020]. AutoTVM identifies performant parameters for optimization templates. These templates represent the sequence of optimizations that the tensor compiler applies on a kernel, including transformations like loop unrolling, splitting, interchange, and tiling. Many of these optimizations are parameterizable. Examples of parameters include the unrolling factor in loop unrolling or the width of the tiling window in loop tiling. AutoTVM assigns values to these parameters using various search heuristics, such as random sampling, simulated annealing, or coordinate descent. In contrast to AutoTVM, Ansor has the capability to generate entirely new templates. In other words, Ansor is not restricted to a single sequence of optimizations. To fill up these templates with concrete optimization parameters, Ansor employs an evolutionary algorithm.

A Combined Search Infrastructure. Currently, within Apache TVM, AutoTVM and Ansor operate as distinct infrastructures, each exhibiting unique functionality and behavior. While Ansor tends to generate superior kernels, AutoTVM is generally faster in terms of search times. However, our research indicates the feasibility of integrating components from both infrastructures to enhance Ansor’s performance. By doing so, we can develop an improved version of Ansor that produces superior kernels compared to the original tool while also reducing search times.

The core idea presented in this work is as follows: Initially, we allow Ansor to explore the kernel optimization space, leveraging its ability to test different sequences of optimizations during this exploration. Subsequently, following this initial exploration phase, we identify the best kernel discovered by Ansor and employ AutoTVM’s Droplet Search—a line search algorithm—to refine and optimize this kernel further. Droplet Search [Canesche et al. 2023] is an iterative search approach. It seeks the optimal configuration of an optimization template by determining a descent direction along the objective function that models the running time of the kernel. Section 2 explains how this approach is implemented, additionally providing some intuition on why it works.

Summary of Findings. Section 3 assesses the performance of this integrated search technique across three diverse architectures: AMD R7-3700X, Fujitsu ARM A64FX, and Nvidia A100. For reference, consider a baseline scenario in the following discussion where Ansor explores 10,000 different schedules for the same kernel. Each kernel implementation results from a distinct combination of a template and its optimization parameters. By halting Ansor after evaluating 300 kernels, selecting the best one, and subsequently optimizing it with Droplet Search, we often achieved superior kernels for twenty different deep-learning models in significantly less time. Some of these models, such as DenseNet201, encompass over one hundred kernels. As an illustration, applying the combined search approach on MNASNet revealed kernel speedups of 1.59x, 1.02x, and 1.08x on the x86, ARM, and NVIDIA platforms, respectively. Correspondingly, the search time for the combined approach decreased by 1.25x, 1.25x, and 1.18x on these architectures.

Software. The current implementation of the combined search approach is accessible at https://github.com/lac-dcc/bennu. It is possible to incorporate this implementation into Apache TVM with minimum interventions. During the development of this implementation, we enhanced Apache TVM by incorporating functionality to transfer templates from Ansor to AutoTVM, a feature we believe could be a valuable addition to the Apache TVM codebase on its own.

2 IMPLEMENTATION

A kernel is an abstract concept: it can be represented by operations on memory indexed by a linear combination of natural numbers. The actual implementation of a kernel is determined by its schedule. In the context of this paper, a schedule results from the application of code transformations
Explore as a Storm, Exploit as a Droplet: A Unified Search Technique for the Ansor Optimizer

onto the naïve implementation of the abstract kernel. The naïve implementation replaces each linear index with a loop. A schedule is the combination of two notions: a sketch and an annotation of the sketch. Following the terminology introduced in the original Ansor work [Zheng et al. 2020], a sketch is the representation of a kernel that results from the application of a certain sequence of optimizations onto the naïve implementation.

**Example 2.1.** Figure 1 (a) shows an example of an abstract kernel. Figure 1 (b) shows a naïve implementation of the abstract kernel seen in Figure 1 (a). Figure 1 (c) shows two sketches, produced after the application of different code optimizations onto the program in Figure 1 (b).

\[
D_{i, j} = \max(C_{i, j}, 0)
\]

\[
\sum_k A_{i, k} \times B_{k, j}
\]

where \(0 \leq i, j, k \leq 512\)

**Fig. 1.** (a) Abstract view of a kernel. (b) Naïve implementation of the abstract kernel. (c) Two optimization sketches for the naïve kernel. (d) Different annotations for the sketches.

Sketches are parameterized with arguments that determine how each optimization is implemented. These arguments—henceforth called annotations—include for instance, the number of threads assigned to a parallelizable loop, the size of the tiling window, or the unrolling factor of an unrollable loop. Figure 1 (d) shows different parameters of the sketches in Figure 1 (c). The set of every valid configuration of annotations for a given sketch forms the optimization space of that sketch. This space has one dimension for each parameter that is allowed to vary. Figure 2 shows two views of the optimization space of the two sketches in Figure 1 (c).

### 2.1 Space Exploration via Ansor

Ansor solves kernel scheduling in two phases. First, it finds sketches. Then, it finds actual parameters for the sketches via an evolutionary algorithm. Sketching generation happens by the exhaustive application of a small collection of rewriting rules. Each rewriting rule consists of a code transformation, such as tiling, parallelization, or unrolling. For more details of this process, we refer the reader to Section 4.1 of Ansor’s original work [Zheng et al. 2020].

Once sketches are available, the generation of actual schedules happens in two phases: parameters are randomly generated, and then an evolutionary algorithm is used to filter out low quality
schedules, and then combine the good candidates to build new schedules. For more details, we refer
the reader to Section 5 of Zheng et al.’s description [Zheng et al. 2020].

Termination in Ansor. The number of possible schedules is very large; hence, Ansor limits the
amount of schedules with a budget of trials. Each trial consists of the observation of the execution
of an actual schedule. Because a machine learning model contains many kernels (layers), an initial
round of trials is partitioned among these layers. Layers are grouped into a worklist, and receive a
quota of trails in round-robin fashion. After an initial round of optimizations, layers that run for a
very short time are removed from this worklist. This process ensures that layers that run for the
longest time are subject to more extensive optimizations. This approach is what Zheng et al. call
“optimizing with gradient descent”. Because the budget of trials is fixed, Ansor is guaranteed to
terminate.

Limitations of Ansor. The main limitation of Ansor is the fact that it is oblivious to the structure
of the search space. For instance, if by increasing the unrolling factor of a loop from 4 to 6 we
observe a performance improvement, it is likely that if we increase it further to 8, then another
improvement could be also observed. However, if by going to 8, we obtain performance degradation,
then it is also likely that further increases will not bring improvements either. Ansor’s exploitation
approach, via an evolutionary algorithm, is not aware of this notion of neighborhood between
kernels, or of potential convex regions in the optimization space.

2.2 Space Exploitation via Droplet Search
Droplet Search [Canesche et al. 2023] is a kernel scheduling algorithm available in AutoTVM.
AutoTVM differs from Ansor in the sense that it does not create new sketches. Rather, it is restricted
to modifying the parameters of a single sketch—the origin of the optimization space. AutoTVM
provides several independent scheduling approaches: random sampling, grid sampling, genetic
sampling, etc. However, only Droplet Search will be of interest to this presentation. Droplet Search
is a variation of an exploitation algorithm called Coordinate Descent\(^1\). It relies on the premise that

\(^1\)It is not clear who invented Coordinate Descent. Descriptions of the algorithm can be found in classic textbooks [Zangwill
1969]. For a comprehensive overview, we recommend the work of Wright [2015].
the parameters of a sketch can be arranged into a coordinate space. Example 2.2 explains what a coordinate space is.

**Example 2.2.** Let us assume a sketch formed by two optimizations: unrolling and tiling. For the sake of this example, unrolling supports five “unrolling factors”: \( \{1, 2, 3, 4, 5\} \). These are the parameters of the loop unrolling optimization. Tiling is parameterized by the size of the tiling window. Let us assume the following sizes: \( \{1, 2, 4, 8, 16\} \). The optimization space, in this case, is formed by \( 5 \times 5 \) points, such as \( (1, 1) \), which means no optimization, or \( (3, 16) \), which indicates that the loop must be unrolled three times, and then tiled with a window of size 16. These points, e.g., \( (1, 1), (1, 2), (3, 16), \) etc, are the coordinates of the optimization space.

From the notion of coordinates, Droplet Search defines a *neighborhood function*: a function that returns the *neighbors* of a given coordinate. Intuitively, the neighbors of a coordinate are the points that are the closest to it. In Example 2.2, the neighbors of (unrolling = 3, tiling = 8) would be the points \( (2, 8), (4, 8), (3, 4) \) and \( (3, 16) \). From this concept of neighborhood, Droplet Search works iteratively, as follows:

1. At iteration zero, let the best current candidate be the set of parameters that implement no optimization.
2. Let \( (c_1, c_2, \ldots, c_n) \) be the best set of parameters discovered up to iteration \( i \).
   - If there exists \( (c'_1, \ldots, c'_i, \ldots, c'_n) \) yields a faster kernel than \( (c_1, \ldots, c_i, \ldots, c_n) \), then update the current best candidate to use \( c'_i \) instead of \( c_i \).
   - If there is no such \( c'_i \), then the search terminates.

**Limitations of Droplet Search.** Droplet search is a fast search algorithm when compared to *Ansor* or to other approaches available in *AutoTVM* [Canesche et al. 2023]. However, it has two fundamental limitations:

- **Droplet Search is restricted to a single sketch.** In other words, it can modify the annotations of a sketch, but it cannot create new sketches. This is a limitation of any search algorithm used in *AutoTVM*, but it is not a limitation of *Ansor*.
- **Droplet Search is highly dependent on the initial schedule that it receives as the *seed* of the search procedure.** If this initial schedule does not exist in the same convex region as the optimal schedule, then Droplet Search cannot find the optimal schedule.

By combining *Ansor* and Droplet Search, we hope to circumvent the limitations of both search techniques. Section 2.3 explains how these two approaches can be used together.

### 2.3 Combining *Ansor* with Droplet Search

To combine Droplet Search and *Ansor*, we determine two parameters:

- **K**: the budget of trials of *Ansor*.
- **N < K**: a subset of trials.

We then proceed as follows:

1. Run *Ansor* on the target model using only \( N \) trials.
2. Give the best schedule found with \( N \) trials to Droplet Search.
3. Run Droplet Search up to convergence.

Figure 3 provides some intuition on this *modus operandi*. As the figure illustrates, the proposed technique seeks to use *Ansor* to explore the universe of sketches, and then use Droplet Search as the core strategy to explore concrete representations of these sketches.

In Section 3 we demonstrate that by choosing proper values for \( K \) and \( N \) we can outperform *Ansor* in two ways: first, producing faster end-to-end machine learning models; second, reducing...
Ansor uses rewriting rules to produce different sketches. Each sketch describes a sequence of optimizations.

Ansor uses an evolutionary algorithm to annotate the sketches; hence, producing actual schedules.

We choose the best schedule (annotated sketch), considering the running time of the end-to-end model.

We give the best schedule to droplet search, which uses coordinate descent to explore its neighborhood.

Fig. 3. Summary of this proposal.

the search time of Ansor. The modifications needed to add this combination to the current code base of Apache TVM are relatively small. Section 2.5 describes these modifications.

2.4 User Interaction

The proposed extension does not change how Ansor is used. Its original implementation can still be invoked via the same commands without modification. If the user wants to apply Droplet Search on the best model found by Ansor after several trials, they only need to run a second command. This section explains such interactions. After Ansor completes the tuning process, users can integrate the droplet_exploitation function into the solution with a single line of Python code. This functionality allows users to optimize the application of the Ansor solution, offering flexibility and control over the process. Example 2.3 shows the core commands.
Explore as a Storm, Exploit as a Droplet: A Unified Search Technique for the Ansor Optimizer

Example 2.3. In order to optimize an ONNX model, we can use Ansor with the command below. This command will run Ansor with a budget of 10,000 trials:

```
$> python3 benchmarks/models_onnx.py -m ansor -a x86 -t 10000
   -l results/x86_resnet18_10k.json -b models/resnet18.onnx
```

This command has no modification in regards to the original implementation of Ansor. Notice that it will produce a log file called `results/x86_resnet18_10k.json`. This file can be given to Droplet Search, to improve even further the final implementation of the end-to-end model, as follows:

```
$> python3 benchmarks/models_onnx.py -m droplet -a x86 -t 100
   -l results/x86_resnet18_10k.json -b models/resnet18.onnx
```

The above command will run Droplet Search on the best candidate found by Ansor, limiting the number of trials given to Droplet Search by 100.

2.5 Implementation Details

For a better understanding of the implementations, we can separate them into two parts: Ansor infrastructure, and AutoTVM infrastructure. Figure 4 shows a diagram of the modifications made to Ansor.

**Ansor infrastructure.** Droplet Search needs a specific data structure to optimize the best solution found by Ansor. To bridge these two algorithms, three functions were created in the `utils.py` file:

- `get_multilayers` receives the Ansor log file and extracts the layers that will be optimized.
- `get_time` receives the log file from Ansor and returns the total time spent tuning.
- `write_file` writes the results in json format and saves it to the system.

The Space class has essential methods that take Ansor’s input format and apply optimizations to it. In the current structure, it optimizes SplitStep (SP) andPragmaStep (PR) parameters. SP focuses on optimizing tile size with a step power of 2, with the range starting from 2 and going up to \( \min(64, \text{size\_tile}) \). PR focuses on unrolling the loop starting at 64 and going up to 512, with the values varying in a power of 2.
**AutoTVM infrastructure.** One challenge in connecting Ansor and AutoTVM is that a class SearchTask does not have a method called config_space in Ansor. A simple solution to solve this problem was to add a method using a Python parameter into the SearchTask class. This method allows to create a link between Ansor and AutoTVM without having to change the Ansor structure. Thus, Droplet Search reuses what was implemented in AutoTVM and only implements some methods necessary for the Space class to be used.

**Summary of Modifications.** With these modifications, we are adding 3 classes and 15 functions to the current Apache TVM implementation, totaling 307 lines of modified code in the TVM repository (Release 17, Apache TVM v0.15.0). The summary of changes follows below:

1. New files:
   - space.py (created 2 classes, 1 function + 6 functions); LoC: 115
   - droplet_search.py (created 1 class with 3 functions); LoC: 61

2. Modified files:
   - task_scheduler.py (created 1 function); LoC: 48
   - utils.py (created 3 functions); LoC: 77
   - search_task.py (created 1 function); LoC: 6

### 3 EVALUATION

This section evaluates the idea proposed in this paper. In particular, it seeks to demonstrate that by exploiting, via Droplet Search, a reduced set of samples explored by Ansor, it is possible to outperform Ansor itself. We shall refer to this new version of Ansor, which uses Droplet Search, as the Combined Approach. Henceforth, we denote it as DPAnsor, reserving Ansor for the original implementation of that tool. In what follows, we explore three research questions:

**RQ1:** How many samples does DPAnsor need to observe in order to produce kernels that outperform those produced by Ansor with 10,000 trials?

**RQ2:** How many samples can DPAnsor observe and still outperform Ansor in terms of search time when the latter uses 10,000 trials?

**RQ3:** How does the size of models impact the behavior of DPAnsor, in terms of kernel performance and search speed?

Before diving into the research questions, we explain our experimental setup. Notice that a fully containerized version of this methodology has been organized as a docker image, which is publicly available at https://github.com/lac-dcc/bennu/tree/main/docker.

**Hardware.** We evaluated the scheduling approaches on three different architectures, as shown in Figure 5. The hardware consists of a general-purpose desktop architecture (AMD Ryzen 7 [AMD 2019]), a cluster-based machine (ARM A64FX [Ookami 2022]), and two graphics processing units (Nvidia A100 [Nvidia 2020a] and Nvidia RTX3080 [Nvidia 2020b]).

![Fig. 5. The architectures evaluated in this report.](image-url)
Software. The experiments reported in this section use versions of Ansor and AutoTVM (Droplet Search) available at Apache TVM v0.13.0, released in July 2023. The code that implements communication between Ansor and Droplet Search is available at https://github.com/lac-dcc/bennu/tree/main/src.

Benchmarks. This section evaluates kernel scheduling across twenty convolution neural networks. The first column of Figure 6 contains the complete list of these models. All these models are implemented using the ONNX representation. The models used in our study are sourced from the ONNX model zoo available at https://github.com/onnx/models.

Methodology. A machine learning model forms a graph of kernels. Ansor optimizes machine learning models per kernel, assuming kernels can be independently optimized. It starts with a budget of trials, where each trial is a transformation that can be applied to a kernel. Let us call this budget \( K \). Ansor ensures that each kernel receives a fraction of these \( K \) trials. Currently, this initial fraction is \( \min(K/L, 64) \), where \( L \) is the number of layers (kernels) in the model. After an initial round of optimizations, Ansor applies the remaining trials onto kernels that run for the longest time. This approach directs the optimization effort to the kernels that are more likely to contribute to the overall running time of the end-to-end model. In what follows, all the results we report are relative to a baseline version of Ansor equipped with a budget of 10,000 trials. We shall test DPAnsor with either \( K = 1, 10, 25, 50, 100, 200, 300, \) or 1,000 trials. Suppose we choose \( K = 100 \), for instance. In that case, we will run Ansor with a budget of 100 trials, pick the best configuration (which results from the independent optimization of the kernels), and give this configuration to AutoTVM’s Droplet Search. We then let Droplet Search run until it reaches convergence.

Presentation. The three figures that summarize results discussed in this section (Fig. 6, Fig. 7, and Fig. 9) are divided into two parts. The left part (labeled “10k speedup execution time comparison”) compares the quality of the kernels produced by Ansor and DPAnsor. The second part (labeled “10k speedup tuning time comparison”) compares the search time of these two approaches.

3.1 RQ1 – On the Quality of End-to-End Models

In this section, one version of an end-to-end model is better than another for a given architecture if it runs faster in that architecture. The execution time of a model is determined by the schedule of the kernels that constitute it. If we apply Droplet Search to the best model produced by Ansor after it observes 10,000 trials, we will likely improve the model (at least, we should not make it worse). However, this section shows that obtaining a better model via DPAnsor with a much lower budget is possible.

3.1.1 AMD Ryzen 7 (x86-64). The x86 architecture represents a widely adopted instruction set architecture (ISA), serving as the basis for numerous computer processors, including those produced by Intel, AMD, and other manufacturers. Figure 6 presents the results for the x86 architecture. In this setup, we observe most speedups after observing only 25 samples with DPAnsor. Furthermore, by giving the best of 100 trials to DPAnsor, we observe speedups in all 20 models. Speedups improve gradually as more samples are added to DPAnsor, to the point that at 1,000 samples, we see an average speedup (geometric mean) of 34%.

3.1.2 Nvidia A100 (Ampere). CUDA (Compute Unified Device Architecture) is the computer architecture that NVIDIA has developed for its graphic cards (GPUs. Nowadays, GPUs are common platforms to execute machine learning models. Figure 7 summarizes results produced for a particular GPU, the NVIDIA A100. The results are not as stable as those presented in Figure 6. However, the big picture is similar: as more trials are given to DPAnsor, it tends to outperform Ansor. With a
Fig. 6. Comparative Analysis of Optimization Results on x86 Architecture Using AMD Ryzen 7 3700X Processor budget of 300 samples, the former consistently outperforms or ties with the latter, for an average speedup of 12%. If we average absolute running times, then this speedup goes up to 14%. Depending on the model, speedups can be very high. Our approach achieves a 66% improvement in execution time for VGG11 with 30 trials.

Fig. 7. Comparative Analysis of Optimization Results on CUDA (Ampere) Architecture Using Nvidia A100 GPU
3.1.3 **RTX3080 (Ampere).** In addition to being a GPU widely used for gaming, it can be used in machine learning models. Unlike the A100, the RTX3080 has lower bandwidth. This makes it effectively slower in data processing than an A100. Figure 8 summarizes results produced for a particular GPU, the NVIDIA RTX3080. With only a budget of 300 samples, Cuda DPAnsor reaches the same results with 10,000 trials by Ansor.

3.1.4 **ARM A64FX (aarch64).** We evaluate the ARM architecture using the FUJITSU Processor A64FX. Figure 9 summarizes these results. In this case, the results are similar to those observed in the NVIDIA A100 card. Speedups are common with a budget of 300 trials, although not in every model. At 1,000 trials, no regressions are observed, although DPAnsor could not outperform Ansor in the Sufflenet.

3.2 **RQ2 – On the Search Time**

We define a scheduling approach as faster than another if it requires less time to converge to a final, optimized version of an end-to-end model. The search time of Ansor encompasses the time spent applying optimizations to kernels, deriving new optimizations, and running the kernels themselves, with a limit set at 10,000 trials. On the other hand, the search time of DPAnsor involves all the steps of Ansor, constrained to a lower number of trials, along with the time it takes to run Droplet Search until convergence on the kernels that compose a model. Although we have restricted Droplet Search to a maximum of 100 trials, it typically converges well before reaching that limit. This section compares the search time between Ansor and DPAnsor.

3.2.1 **AMD Ryzen 7 (x86-64).** The right side of Figure 6 compares search times in the x86 hardware. For most models, DPAnsor is consistently faster than Ansor for any number of trials up to \( K = 300 \). At 1,000 trials, Ansor becomes consistently faster. Also, Ansor tends to outperform DPAnsor for very large models. This fact happens due to the longer time that Droplet Search takes to converge: the more complex the model, the more room Droplet Search will have to optimize it.
Fig. 9. Comparative Analysis of Optimization Results on aarch64 Architecture Using A64FX Processor

3.2.2 Nvidia A100 (Ampere). The right side of Figure 7 compares search times in the Nvidia setup. Following the behavior observed in the x86 board, Cuda DPAnsor is also consistently faster than Ansor in every setup where $K \leq 300$. At 1,000 samples, the relative benefit that DPAnsor brings onto Ansor disappears. However, even in this scenario, both approaches present almost the same search time.

3.2.3 Nvidia RTX3080 (Ampere). The right side of Figure 8 compares search times in the Nvidia setup. Following the behavior observed in the x86 board, Cuda DPAnsor is also consistently faster than Ansor in every setup where $K \leq 300$. In some cases, at 1,000 samples, there are improvements.

3.2.4 ARM A64FX (aarch64). The right side of Figure 9 summarizes relative speedups in the ARM setting. This setup yields search times similar to those observed in x86 and Cuda. Even the global averages tend to be similar. Limiting the budget of DPAnsor to 300 points, we achieve an average speedup of 1.53x (geometric mean). Again, in some individual cases, speedups are noticeable. In particular, for the small models, DPAnsor is more than twice faster than Ansor, even with a budget of 300 trials.

3.3 RQ3 – On the Impact of Model Size

The behavior of DPAnsor, when compared to Ansor, varies strongly with the size of the model. We summarize this variation with two observations:

1. The larger the model, the less samples DPAnsor needs to observe to outperform Ansor, if Ansor uses a budget of 10,000 samples.

2. The larger the model, the lower the benefit, in terms of search time, of DPAnsor over Ansor.

The rest of this section provides data to support these two conclusions.
Explore as a Storm, Exploit as a Droplet: A Unified Search Technique for the Ansor Optimizer

Figure 10 shows these lines regarding four models and two architectures: AMD’s x86 and NVIDIA’s Ampere. We chose these two architectures because x86 is the scenario where DPAnsor performs better, and Ampere is the scenario where it performs worse.

Figure 10 uses our two smallest and our two largest models. The numbers on the axes show ratios between DPAnsor and Ansor; the latter using a budget of 10,000 trials. Each dot in Figure 10 refers to the number of trials that DPAnsor is allowed to observe before shifting to Droplet Search. The figure labels dots that refer to DPAnsor with one sample (its most restrictive scenario) and dots that refer to 1,000 samples (its least restrictive scenario).

The slopes of the lines in Figure 10 are always negative, meaning that as more samples are given to DPAnsor, the difference between its search time and Ansor’s reduces, but the quality of the kernels that it finds improves. However, the inclination changes with the size of the model. The larger the model, the lower the benefit of DPAnsor over Ansor in terms of search time; but the higher the relative benefit in terms of kernel speed. This result is due to Ansor’s fixed budget of 10,000 trials. In a small model, more trials are distributed to each layer; in a large model, in turn, each layer receives only a handful of trials.

Figure 11 provides further data that corroborates the previous observations. The figure shows how kernel quality and search time vary with the size of models. If we consider AlexNet, for instance, which has only 13 layers, each layer might receive, on average, $10^4/13$ trials. Inevitably, good kernels will emerge from this search. Thus, the benefit of the exploitation phase, which uses Droplet Search, tends to be smaller. On the other hand, if we consider DenseNet201, which has 113 layers, then Ansor allocates, on average, $10^4/113$ trials per layer—less than 100 samples per each kernel of the computational graph. This number is too low to effectively explore the space of possible kernel implementations. In this case, Droplet Search has more opportunity to improve the kernels that Ansor finds. Even using only one trial to find the origin of the optimization space is already enough to have DPAnsor outperforming Ansor in terms of the quality of the model.

3.4 Threats to Validity – On the Statistical Significance of Reported Results

The discussion presented in Sections 3.1 and 3.2 center around the running time of end-to-end models and on the search time of different scheduling techniques. It is possible that the running times presented for kernels and schedulers is similar enough to the point of hindering our conclusions...
meaningless. It is easy to demonstrate that such is not the case for search times. In this case, each scheduling technique runs for a very long time. To give the reader an idea, \( \text{Ansor} \), with a budget of 10,000 trials (or baseline), takes 11,195 seconds to schedule AlexNet, our smallest model. This time drops to 2,839 seconds considering DPAnsor with a budget of 300 trials. In this section, we show that the running times of the kernels produced by \( \text{Ansor} \) and DPAnsor also present statistically significant differences. To this end, we proceed as follows: we choose two end-to-end models (AlexNet and Resnet34), and present their mean running times, plus the standard deviations of such means, and the p-values that result from comparing the original version of \( \text{Ansor} \) with our extended version of it.

**Discussion.** Figure 12 presents an analysis of deviation and p-value for Resnet34 and AlexNet, employing DPAnsor across varied hardware configurations. The networks are characterized by the number of layers, designated as “Num of Layers”. Resnet34 comprises 19 layers, and AlexNet consists of 13 layers. The table includes the average runtime of the best kernels proposed by DPAnsor in seconds, denoted as “Mean Kernel Runtime,” and the average of standard deviation, “Mean Std Dev,” across all layers of the best schedule of a network, in seconds. Additionally, Figure 12 presents the count of layers with a p-value less than 0.01, labeled as “Num of Layers with p-value < 0.01.” We compare the results using “Coefficient of Variation” (CV), presented in percentage (%). It is the...
standard deviation expressed as a percentage of the mean, as a measure of relative variability. For each layer in a network, it is noteworthy that the best-performing kernel is selected.

<table>
<thead>
<tr>
<th>Network</th>
<th>Num of Layers</th>
<th>Hardware</th>
<th>Mean Kernel Runtime (sec)</th>
<th>Mean Std Dev (kernel runtime, sec)</th>
<th>Coefficient of Variation (%)</th>
<th>Num of Layers with p-value &lt; 0.01</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet34</td>
<td>19</td>
<td>AMD R7-3700X</td>
<td>5.45E-04</td>
<td>3.35E-05</td>
<td>9.88E+00</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A64FX</td>
<td>3.22E-04</td>
<td>1.51E-05</td>
<td>4.07E+00</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A100</td>
<td>2.95E-05</td>
<td>1.03E-07</td>
<td>3.48E-01</td>
<td>6</td>
</tr>
<tr>
<td>AlexNet</td>
<td>13</td>
<td>AMD R7-3700X</td>
<td>8.94E-04</td>
<td>5.36E-05</td>
<td>5.24E+00</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A64FX</td>
<td>4.91E-04</td>
<td>1.26E-06</td>
<td>2.79E+01</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A100</td>
<td>4.66E-05</td>
<td>4.46E-08</td>
<td>9.58E-02</td>
<td>8</td>
</tr>
</tbody>
</table>

Fig. 12. Coefficient of Variation and P-Value Test

This analysis provides insights into the networks’ behavior under different hardware settings, emphasizing deviations from the mean and layers with statistically significant DPAnsor. The Standard deviation measures how individual values deviate from the mean. A lower standard deviation indicates proximity to the mean and a higher standard deviation suggests greater dispersion. The p-value serves as a probability measure, assessing the strength of evidence against a null hypothesis. The null hypothesis typically posits no effect or difference. A lower p-value signifies stronger evidence against the null hypothesis, leading to its rejection when \( p \leq \alpha \), where \( \alpha \) is the chosen significance level. Conversely, \( p > \alpha \) results in the failure to reject the null hypothesis, indicating a lack of statistical significance at the selected threshold. In our case, we have selected the significance level, \( \alpha \) as 0.01.

As shown in Figure 12, both the networks under test performed similarly on the three different hardware. We could observe a CV of 10% in the average runtime of the best kernel for Resnet34 and 5% for AlexNet, which DPAnsor suggested for x86. ARM A64FX showed a lower CV with 4% for Resnet34 and less than 1% for AlexNet. The reported CV is the least in Nvidia A100, with less than 1% for both networks. We observed a noteworthy trend where many layers in each network exhibited p-values of less than 0.01. This indicates a statistically significant pattern per layer, suggesting that the observed results are unlikely to have occurred by chance.

4 CONCLUSION

This proposal outlines the integration of an extension into the Ansor kernel scheduler, introducing an exploitation phase to enhance its functionality. The extension incorporates Droplet Search’s coordinate descent algorithm, utilizing the best candidate identified by Ansor as the initial seed for Droplet Search. This integration offers benefits to both Ansor and Droplet Search:

- It enhances Ansor’s capability to exploit kernel schedules. The current implementation of Ansor lacks awareness of the relationships between neighboring kernel schedules, as it lacks a concept of “distance” between schedules. By introducing this extension, Ansor gains the ability to explore and leverage similarity relations within the neighborhood of successful schedules.

- It addresses Droplet Search’s primary limitation, which is its reliance on a well-defined seed (the initial kernel that initiates coordinate descent). Currently, the seed is determined manually, requiring a programmer to provide Droplet Search with an appropriate initial kernel. This proposal automates the seed generation process by utilizing Ansor.

As detailed in Section 2.4, the proposed extension does not impact current Apache TVM users, as the use of Droplet Search with Ansor remains entirely optional. Section 2.5 outlines that the modifications to the existing code base are minimal, involving 307 lines of code across five files. Finally, Section 3 demonstrates that the proposed extension effectively improves Ansor in terms of the quality of the end-to-end models it produces and its search time.
Software. The proposed extension has been submitted to the Apache TVM community as a pull request. Additionally, we have set up a docker container with all the necessary dependencies and scripts to reproduce the experiments discussed in Section 3. The container is available at https://github.com/lac-dcc/bennu.

REFERENCES


