Explore as a Storm, Exploit as a Droplet: A Unified Search Technique for the `MetaSchedule`

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

Recently, we have submitted a "Request for Comments" about augmenting TVM Ansor with a quick exploitation phase based on coordinate descent. The proposed technique was well received by the TVM community, and eventually a new algorithm was approved in Ansor. We have realized that this approach seems to improve other kernel schedule that performs wide exploration. To demonstrate this fact, we have also implemented our exploitation phase onto TVM `MetaSchedule`. The `MetaSchedule` is the most recent kernel scheduler added to TVM, and, in many settings, seems to outperform Ansor. Incidentally, it also greatly benefits from fine-tuning via droplet search, as we demonstrate in this report.

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 `MetaSchedule`. 1, 1 (June 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. As Ansel et al. [2024] explains, many tensor compilers, including `TVM` [Chen et al. 2018], `nvFuser` [Sarofeen et al. 2022] and `NNC` [Zolotukhin 2021], follow a design probably inspired by `Halide` [Ragan-Kelley et al. 2013]. An essential characteristic of this design is the separation between the semantics of the kernel (what the kernel does) from its schedule (when the kernel does it). Since the same kernel semantics can be implemented by many different schedules, 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 three distinct optimizing infrastructures for solving kernel scheduling: `AutoTVM` [Chen et al. 2018], `Ansor` [Zheng et al. 2020], and the `MetaSchedule`. `AutoTVM` finds parameters for *kernel sketches*. The sketch of a kernel represents the sequence of optimizations that the tensor compiler applies on the abstract description of that kernel, such as loop unrolling, splitting, interchange, and tiling. Many of these optimizations are parameterizable.

Authors' addresses: Michael Canesche, UFMG, Brazil, michaelcanesche@dcc.ufmg.br; Gaurav Verma, Stony Brook University, USA, gaurav.verma@stonybrook.edu; Fernando Magno Quintão Pereira, UFMG, Brazil, fernando@dcc.ufmg.br.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Association for Computing Machinery.
XXXX-XXXX/2024/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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. One of these heuristics is of interest to this paper: Droplet Search [Canesche et al. 2024]. Droplet Search 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. In contrast to AutoTVM, the other two autotuners—Ansor and and MetaSchedule —have the capability to generate new sketches. In other words, Ansor and MetaSchedule are not restricted to a single sequence of optimizations. To fill up these templates with concrete optimization parameters, MetaSchedule employs an evolutionary algorithm. Section 2.1 provides further details on how MetaSchedule works, whereas Section 2.2 explains how Droplet Search works.

A Combined Search Infrastructure. In general, MetaSchedule typically generates higher-quality kernels compared to AutoTVM's Droplet Search, as MetaSchedule is not limited to a single search space. Each sketch generated by MetaSchedule leads to the exploration of an entirely new search space. However, when constrained to a single sketch, Droplet Search tends to outperform MetaSchedule's evolutionary algorithm. Drawing on the terminology from recent work by Ding et al. [2023], Droplet Search's coordinate descent approach is "*hardware centric*", while MetaSchedule's genetic algorithm is "*input centric*"; better embodying this quality that Sorensen et al. [2019] calls "*performance portability*". In essence, Droplet Search, by traversing the search space contiguously, is sensitive to cache sizes and cache hierarchy levels. Nevertheless, despite Droplet Search's tendency to identify optimal points within the search space, this algorithm is unable to navigate beyond this space due to its reliance on the initial sketch.

Inspired by these observations, this paper studies the following research question: "*Is it possible to combine* MetaSchedule*'s exploration and Droplet Search's exploitation mechanisms; thus, obtaining the advantages of each approach?*" By doing so, we can develop a version of MetaSchedule that produces superior kernels compared to the original tool while also reducing search times. This paper brings evidence that such a combination is effective. The core idea presented in this work is as follows: Initially, we allow MetaSchedule to explore the kernel optimization space, leveraging its "space travel ability" to test different sequences of optimizations during this exploration. Subsequently, following this initial exploration phase, we identify the most promising kernel space discovered by MetaSchedule and employ AutoTVM's Droplet Search—a line search algorithm—to find a good kernel within this space.

Summary of Findings. This paper describes findings of an eminently empirical nature. The search techniques discussed in Section 2 are not an original contribution of this work: MetaSchedule's exploration algorithm was created by TVM [Chen et al. 2018], and Droplet Search's exploitation approach was designed by Canesche et al. [2024]. Nevertheless, the combination of these two techniques into a practical tool required a number of experimental observations and engineering decisions which Section 3 organizes into six research questions.

The experiments discussed in this paper show that the combined exploration-exploitation methodology outperforms the original implementation of MetaSchedule in terms of kernel quality and search speed. Positive results were observed across four different processors (AMD R7-3700X, Fujitsu ARM A64FX, NVIDIA RTX 3080, and NVIDIA A100), and in 20 popular deep-learning models, including AlexNet, VGG, ResNet, MobileNet, Inception, GoogleNet, and DenseNet. As an illustration, by terminating MetaSchedule after sampling 300 kernels and subsequently optimizing the best candidate with Droplet Search, we consistently outperformed MetaSchedule (running with a budget of 10,000 samples) across all 4×20 architecture-model pairs. For instance, applying the combined search approach to MnasNet yielded kernel speedups of 1.13x, 1.12x, and 1.08x on x86, ARM, and NVIDIA platforms, respectively. Correspondingly, the search time for the combined approach decreased by 1.70x, 1.44x, and 1.39x on these architectures.

2 EXPLORATION VIA META SCHEDULE; EXPLOITATION VIA DROPLET SEARCH

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*. The schedule of a kernel determines in which order the different memory elements are accessed when the kernel runs. Following the terminology introduced in the original Ansor work [Zheng et al. 2020], a schedule is the combination of two notions: a *sketch* and an *annotation* of the sketch. Definition 2.1 enumerates these notions.

Definition 2.1 (The Kernel Search Space). The naïve implementation of a kernel replaces each linear index in the abstract representation of the kernel with a loop. A sketch is a sequence of transformations, such as loop fusion, splitting, or tiling, that can be applied onto the naïve implementation of the kernel. An annotation of the sketch is the set of parameters that control the effect of each optimization in the sketch, such as unrolling factor, length of tiling window, number of threads in parallelization, etc. We call an annotated sketch a "kernel". A kernel is a concrete program: it effectively runs. Each sketch determines a kernel search space, which is the set of every valid way to annotate that sketch.

Example 2.2. 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).

(a)

$$C[i, j] = \sum_k A[i, k] \times B[k, j]$$

$$D[i, j] = \max(C[i, j], 0)$$

$$\text{where } 0 \le i, j, k \le 512$$

(b)

```
for i in range(512):
    for j in range(512):
        for k in range(512):
            C[i, j] += A[i, k] * B[k, j]
for i in range(512):
    for j in range(512):
        D[i, j] = max(C[i, j], 0.0)
```

(c-i)

```
parallel i.0@j.0@i.1@j.1 in range(P0):
    for k.0 in range(P1):
        for i.2 in range(P2):
            unroll k.1 in range(P3):
                unroll i.3 in range(P4):
                    vectorize j.3 in range(P5):
                        C[...] += A[...] * B[...]
for i.4 in range(P6):
    vectorize j.4 in range(P7):
        D[...] = max(C[...], 0.0)
```

(d-i)

```
P0 = 256
P1 = 32
P2 = 16
P3 = 16
P4 = 4
P5 = 16
P6 = 64
P7 = 16
```

(c-ii)

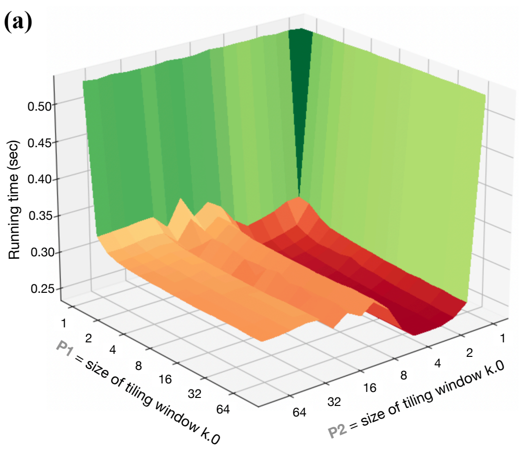
```
parallel i.2 in range(P8):
    for j.2 in range(P9):
        for k.1 in range(PA):
            for i.3 in range(PB):
                vectorize j.3 in range(PC):
                    C[...] += A[...] * B[...]
parallel i.4 in range(PD):
    for j.4 in range(PE):
        D[...] = max(C[...], 0.0)
```

(d-ii)

```
P8 = 16
P9 = 128
PA = 512
PB = 32
PC = 4
PD = 512
PE = 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.

Figure 1 (d) shows different parameters of the sketches in Figure 1 (c). As introduced in Definition 2.1, the set of every valid configuration of annotations for a given sketch forms the *search 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).



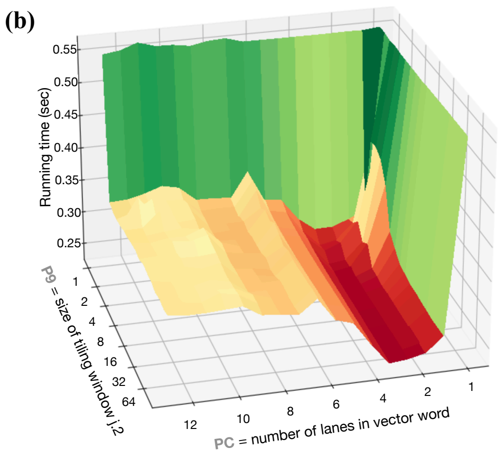


Fig. 2. (a) A three-dimensional view of the optimization space formed by the parameters **P1** and **P2** seen in Figure 1 c-i. (b) A three-dimensional view of the optimization space, this time involving parameters **P9** and **PC**.

2.1 Space Exploration via `Meta Schedule`

Meta Schedule is a unified scheduling approach developed by TVM that combines the strengths of two previous systems. It offers the flexibility to manually generate search spaces, similar to AutoTVM, while also providing the efficiency and automation of space generation as seen in Ansor. Meta Schedule finds the best way to run code on different hardware by intelligently searching the space that Definition 2.1 formalizes. The core search approach adopted by `MetaSchedule` is based on Ansor's algorithm [Zheng et al. 2020]—the evolutionary search. Thus, following Ansor's implementation, search in `MetaSchedule` can be divided into three phases:

Schedule generation: in this phase, new optimization templates (also called "Schedules") are created. As explained in Definition 2.1, each optimization template determines one kernel search space.

Kernel annotation: in this phase, an initial population of annotated kernels is created. Following Definition 2.1, each annotated kernel is a point in the search space determined by the schedule that provides the annotations.

Kernel evolution: in this phase, candidate kernels are sorted according to an estimation of their performance (via `MetaSchedule`'s cost model). The best candidates are sampled (executed and timed), and this information is used to improve the cost model.

Schedule Generation. The generation of a new optimization template happens via the application of a small collection of rewriting rules[1]. Each rewriting rule provokes a code transformation, such as tiling, parallelization, or unrolling. The schedule generation rules are hardware-dependent: some rules, like those that bind loop iterations to threads, are only well-defined for GPUs, for instance.

[1]These are the rules that were implemented in https://github.com/apache/tvm/blob/main/src/meta_schedule/schedule_rule on 06/12/2024

Kernel Annotation. Schedule are not executable programs: they have annotations which must be replaced with actual values. These values are the parameters of optimizations, as Definition 2.1 explains. Thus, as a second step of the iterative exploration approach of `MetaSchedule`, an initial population of annotated templates is produced via the application of "initialization rules". Each initialization rule is parameterized by a probability distribution, which associates concrete values with the probability that they can be chosen.

Evolution of the Annotated Schedule. To explore different kernel spaces, `MetaSchedule` keeps a population of promising *candidate kernels*. This population is updated in an iterative process. At each iteration, the current population of candidates evolves through the application of mutation strategies. To select the next set of candidates, `MetaSchedule` does not run every kernel in the current population. Instead, it uses a cost model to select candidates that are likely to run efficiently. These promising candidates are executed, and the result of these samples is used to recalibrate the cost model; hence, improving the estimates of the next candidates. Periodically, the algorithm reports statistical information regarding the search progress, including maximum and minimum scores, population size, and the success rate of mutations. Upon completion of the specified number of iterations, the best-performing kernels are recorded as the winning candidates.

Termination in `MetaSchedule`. The number of possible schedules is very large; hence, The Meta Schedule limits the amount of sampled kernels with a *budget of trials*. Each trial consists of the observation of the execution of an actual schedule, which happens at the end of the evolutionary phase. Because a machine learning model contains many kernels, an initial round of trials is partitioned among these layers. Layers are grouped into a worklist, and receive a quota of trials 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. [2020] call "optimizing with gradient descent". Because the budget of trials is fixed, `MetaSchedule` is guaranteed to terminate.

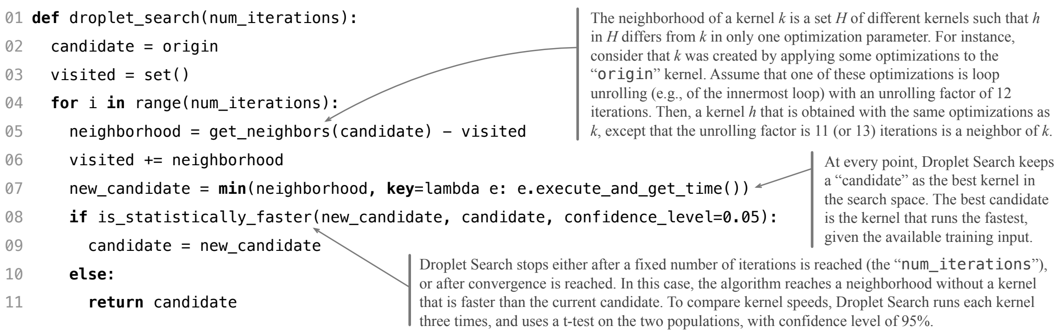
Limitations of`MetaSchedule`. The main limitation of `MetaSchedule` is the fact that it is oblivious to the structure of the search space. For instance, if by increasing the unrolling factor of a loop from 4 to 6 we observe a performance improvement, it is likely that if we increase it further to 8, 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 future improvements. `MetaSchedule`'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. 2024] 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*[2]. It relies on the premise that the parameters of a sketch can be arranged into a coordinate space. Figure 3 contains an annotated version of the algorithm.

Example 2.3. 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

[2]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].



```
def droplet_search(num_iterations):
  candidate = origin
  visited = set()
  for i in range(num_iterations):
    neighborhood = get_neighbors(candidate) - visited
    visited += neighborhood
    new_candidate = min(neighborhood, key=lambda e: e.execute_and_get_time())
    if is_statistically_faster(new_candidate, candidate, confidence_level=0.05):
      candidate = new_candidate
    else:
      return candidate
```

The neighborhood of a kernel k is a set H of different kernels such that h in H differs from k in only one optimization parameter. For instance, consider that k was created by applying some optimizations to the "`origin`" kernel. Assume that one of these optimizations is loop unrolling (e.g., of the innermost loop) with an unrolling factor of 12 iterations. Then, a kernel h that is obtained with the same optimizations as k, except that the unrolling factor is 11 (or 13) iterations is a neighbor of k.

At every point, Droplet Search keeps a "candidate" as the best kernel in the search space. The best candidate is the kernel that runs the fastest, given the available training input.

Droplet Search stops either after a fixed number of iterations is reached (the "`num_iterations`"), or after convergence is reached. In this case, the algorithm reaches a neighborhood without a kernel that is faster than the current candidate. To compare kernel speeds, Droplet Search runs each kernel three times, and uses a t-test on the two populations, with confidence level of 95%.

Fig. 3. The Droplet Search kernel scheduling algorithm. This pseudo-code is a simplified version of the original presentation of the algorithm, taken from [Canesche et al. 2024]. We have removed speculation and parallelism from this version, as these features are immaterial for the presentation of our ideas.

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 5×5 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., $(1, 1)$, $(3, 16)$, 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 2.3, the neighbors of $(\text{unrolling} = 3, \text{tiling} = 8)$ 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.
   (a) If there exists $c'_i, 1 \leq i \leq n$, such that $(c_1, \ldots, c'_i, \ldots, c_n)$ 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.
   (b) 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. 2024]. 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 `Meta Schedule` with Droplet Search

To combine Droplet Search and `MetaSchedule`, we determine two parameters:

- K: the budget of trials of `MetaSchedule`.

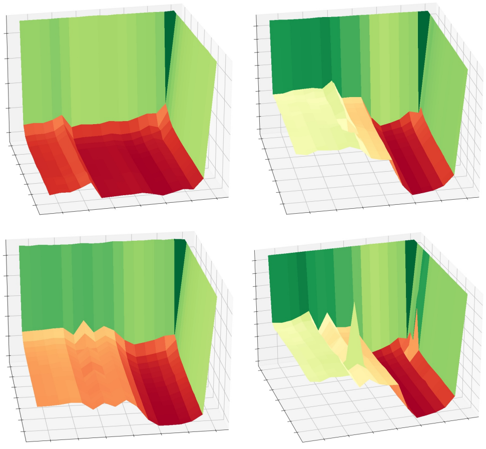
- $N < K$: a subset of trials.

We then proceed as follows:

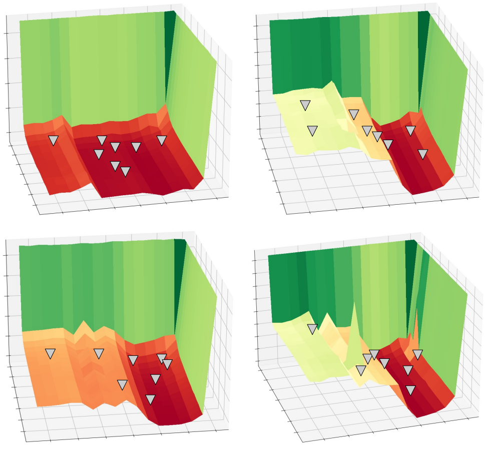
(1) Run `MetaSchedule`on the target model using only N trials.
(2) Give the best schedule found with N trials to Droplet Search.
(3) Run Droplet Search up to convergence.

Figure 4 provides some intuition on this *modus operandi*. As the figure illustrates, the proposed technique seeks to use `MetaSchedule` to explore the universe of sketches, and then use Droplet Search as the core strategy to explore concrete representations of these sketches.

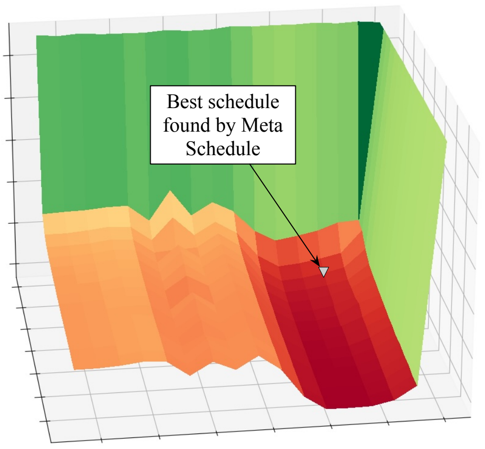
(**a**) Meta Schedule uses rewriting rules to produce different sketches. Each sketch determines a new *kernel search space* via the optimizations that it encodes.



(**b**) Meta Schedule annotates the sketches via initialization and evolution rules. It runs some of the *annotated templates*, to collect good kernel candidates.



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



(**d**) We give the best schedule to Droplet Search, which uses coordinate descent to explore its neighborhood.

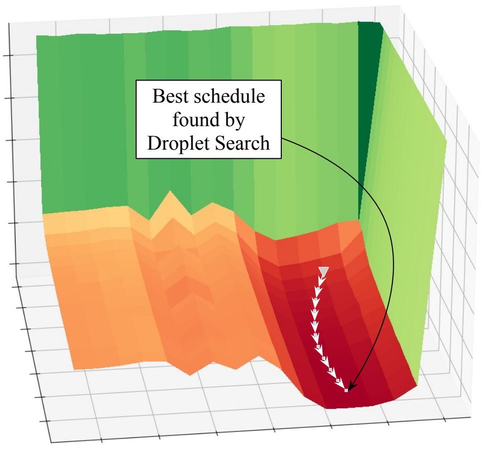


Fig. 4. Coarse exploration of different kernel search spaces with `MetaSchedule`, and careful exploitation of the best candidate with Droplet Search.

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 the search time of Ansor. The modifications needed to add this combination to the current code base of Apache TVM are relatively small. Section 2.4 describes these modifications.

2.4 Implementation Details

In order to combine MetaSchedule's current implementation with Droplet Search, we have added 3 classes and 15 functions to the current Apache TVM code base. These modifications add up to 498 lines of code in the TVM repository (Release 17, Apache TVM v0.17.0). The summary of changes follows below:

(1) New files:
  - space.py (created 1 class, 11 functions); LoC: 240
  - droplet.py (created 1 class with 9 functions); LoC: 124
  - post_opt.py (created 1 function); LoC: 38
  - utils.py (created 4 functions); LoC: 96

The proposed extension does not change how MetaSchedule is used. Its original implementation can still be invoked via the same commands without modification. If users want 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 via a tutorial-like example:

Example 2.4. 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 dpms -a x86 -t 10000 \
   -l results/x86_resnet18_10k -b models/resnet18.onnx
```

This command has no modification in regards to the original implementation of MetaSchedule. Notice that it will produce a log file inside of results/x86_resnet18. 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 dpms -a x86 -t 100 \
   -l results/x86_resnet18_10k -b models/resnet18.onnx
```

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

3 EVALUATION

This section evaluates the new version of the MetaScheduler augmented with droplet search. In particular, it seeks to demonstrate that by exploiting, via Droplet Search, a reduced set of samples explored by MetaSchedule, it is possible to outperform MetaSchedule itself. We shall refer to this new version of MetaSchedule, which uses Droplet Search, as the *Combined Approach.* Henceforth, we denote it as DPMS, reserving MetaSchedule for the original implementation of that tool. In what follows, we explore six research questions:

RQ1: How many samples does DPMS need to observe in order to produce kernels that outperform those produced by MetaSchedule with 10,000 trials?
RQ2: How many samples can DPMS observe and still outperform MetaSchedule in terms of search time when the latter uses 10,000 trials?
RQ3: How does the size of models impact the behavior of DPMS, in terms of kernel performance and search speed?
RQ4: How does the average number of samples that Droplet Search gauges per layer varies with the initial budget allocated to DPMS 's exploration phase?

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://remove/due/to/blind/review.

Hardware and Software. We evaluated the scheduling approaches on four 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]). The experiments reported in this section use versions of MetaSchedule and AutoTVM (Droplet Search) available at Apache TVM v0.16.0, released in April 2024.

Device	Architecture	ISA	Clock Max (GHz)	Memory			
				RAM (GB)	Cache L1 (MiB)	Cache L2 (MiB)	Cache L3 (MiB)
AMD R7-3700X	x86-64	x86-64 (x86)	4.4	64	0.512	4	32
ARM A64FX	aarch64	ARMv8.2 + SVE	2.2	32	6	32	-
Nvidia A100	Ampere	PTX	1.4	40	-	-	-
Nvidia RTX3080	Ampere	PTX	1.7	12	-	-	-

Fig. 5. The architectures evaluated in this report.

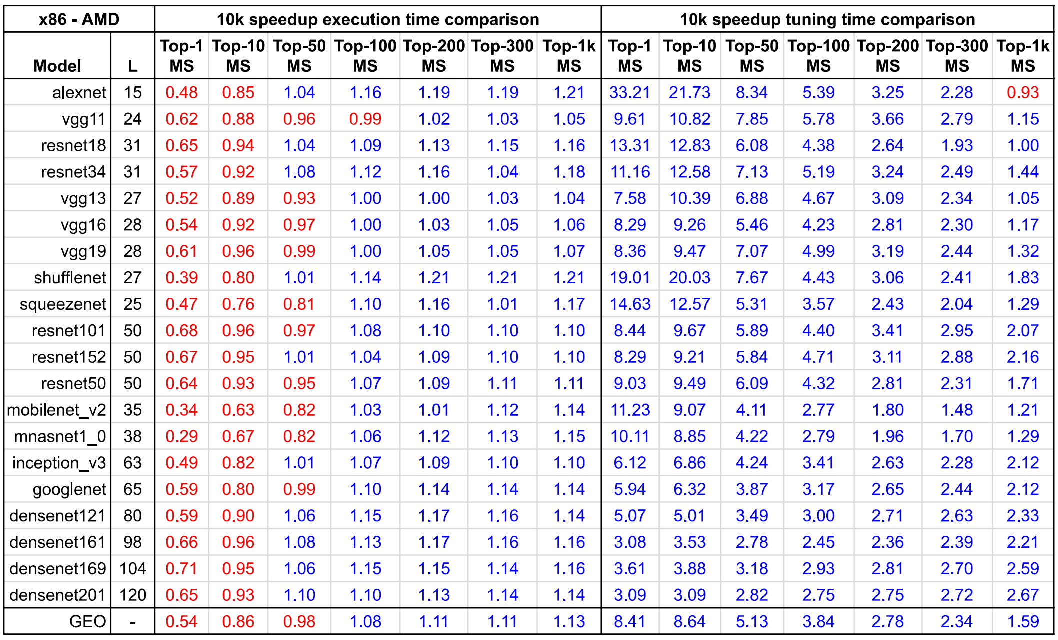
Benchmarks. This section evaluates kernel scheduling across twenty neural networks. The first column of Figure 6 contains the complete list of these models. All these models are implemented using the ONNX representation. The models used in our study are sourced from the ONNX model zoo available at https://github.com/onnx/models.

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 DPMS with either $K = 1, 10, 50, 100, 200, 300,$ 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.

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 DPMS with a much lower budget is possible. The four figures that summarize results discussed in this section and in Section 3.2 (Fig. 6, Fig. 7, Fig. 8, and Fig. 9) are divided into two parts. The left part (labeled "10k speedup execution time comparison") compares the running time of the kernels produced by Ansor and DPMS. The second part (labeled "10k speedup tuning time comparison") compares the search time of these two approaches.

Discussion: 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 recording only 50 samples with DPMS. Furthermore, by giving the best of 100 trials to DPMS, we observe speedups in all 20 models. Speedups improve gradually as more samples are added to DPMS, to the point that with 1,000 samples, we see an average speedup (geometric mean) of 13%.



x86 - AMD		10k speedup execution time comparison							10k speedup tuning time comparison						
Model	L	Top-1 MS	Top-10 MS	Top-50 MS	Top-100 MS	Top-200 MS	Top-300 MS	Top-1k MS	Top-1 MS	Top-10 MS	Top-50 MS	Top-100 MS	Top-200 MS	Top-300 MS	Top-1k MS
alexnet	15	0.48	0.85	1.04	1.16	1.19	1.19	1.21	33.21	21.73	8.34	5.39	3.25	2.28	0.93
vgg11	24	0.62	0.88	0.96	0.99	1.02	1.03	1.05	9.61	10.82	7.85	5.78	3.66	2.79	1.15
resnet18	31	0.65	0.94	1.04	1.09	1.13	1.15	1.16	13.31	12.83	6.08	4.38	2.64	1.93	1.00
resnet34	31	0.57	0.92	1.08	1.12	1.16	1.04	1.18	11.16	12.58	7.13	5.19	3.24	2.49	1.44
vgg13	27	0.52	0.89	0.93	1.00	1.00	1.03	1.04	7.58	10.39	6.88	4.67	3.09	2.34	1.05
vgg16	28	0.54	0.92	0.97	1.00	1.03	1.05	1.06	8.29	9.26	5.46	4.23	2.81	2.30	1.17
vgg19	28	0.61	0.96	0.99	1.00	1.05	1.05	1.07	8.36	9.47	7.07	4.99	3.19	2.44	1.32
shufflenet	27	0.39	0.80	1.01	1.14	1.21	1.21	1.21	19.01	20.03	7.67	4.43	3.06	2.41	1.83
squeezenet	25	0.47	0.76	0.81	1.10	1.16	1.01	1.17	14.63	12.57	5.31	3.57	2.43	2.04	1.29
resnet101	50	0.68	0.96	0.97	1.08	1.10	1.10	1.10	8.44	9.67	5.89	4.40	3.41	2.95	2.07
resnet152	50	0.67	0.95	1.01	1.04	1.09	1.10	1.10	8.29	9.21	5.84	4.71	3.11	2.88	2.16
resnet50	50	0.64	0.93	0.95	1.07	1.09	1.11	1.11	9.03	9.49	6.09	4.32	2.81	2.31	1.71
mobilenet_v2	35	0.34	0.63	0.82	1.03	1.01	1.12	1.14	11.23	9.07	4.11	2.77	1.80	1.48	1.21
mnasnet1_0	38	0.29	0.67	0.82	1.06	1.12	1.13	1.15	10.11	8.85	4.22	2.79	1.96	1.70	1.29
inception_v3	63	0.49	0.82	1.01	1.07	1.09	1.10	1.10	6.12	6.86	4.24	3.41	2.63	2.28	2.12
googlenet	65	0.59	0.80	0.99	1.10	1.14	1.14	1.14	5.94	6.32	3.87	3.17	2.65	2.44	2.12
densenet121	80	0.59	0.90	1.06	1.15	1.17	1.16	1.14	5.07	5.01	3.49	3.00	2.71	2.63	2.33
densenet161	98	0.66	0.96	1.08	1.13	1.17	1.16	1.16	3.08	3.53	2.78	2.45	2.36	2.39	2.21
densenet169	104	0.71	0.95	1.06	1.15	1.15	1.14	1.16	3.61	3.88	3.18	2.93	2.81	2.70	2.59
densenet201	120	0.65	0.93	1.10	1.10	1.13	1.14	1.14	3.09	3.09	2.82	2.75	2.75	2.72	2.67
GEO	-	0.54	0.86	0.98	1.08	1.11	1.11	1.13	8.41	8.64	5.13	3.84	2.78	2.34	1.59

Note: L denotes the number of the optimized layers.

Fig. 6. Comparative Analysis of Optimization Results on the x86 Architecture Using an AMD Ryzen 7 3700X Processor. Numbers show Meta Schedule/DPMS ratios. Thus, results higher than 1.0 (in blue) denote improvements of DPMS (this paper) over MetaSchedule.

Discussion: 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 the results produced for a particular GPU, the NVIDIA A100. The results are not as consistent as those presented in Figure 6. However, the big picture is similar: as more trials are given to DPMS, it tends to outperform MetaSchedule. Speedups are common with a budget of 300 trials, although not in every model. At 1,000 trials, no statistically significant regressions are observed Depending on the model, speedups can be consequential. Our approach achieves a 16% improvement in execution time for VGG11 with 300 trials.

Discussion: 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 DPMS reaches the same results with 10,000 trials by MetaSchedule.

Cuda - A100		10k speedup execution time comparison							10k speedup tuning time comparison						
Model	L	Top-1 MS	Top-10 MS	Top-50 MS	Top-100 MS	Top-200 MS	Top-300 MS	Top-1k MS	Top-1 MS	Top-10 MS	Top-50 MS	Top-100 MS	Top-200 MS	Top-300 MS	Top-1k MS
alexnet	14	0.29	0.48	0.65	0.84	0.85	0.85	1.06	19.73	13.79	6.86	4.48	2.81	2.06	1.09
vgg11	18	0.13	0.66	0.85	0.97	0.99	1.16	1.08	25.06	16.81	7.28	4.73	2.93	2.08	0.90
resnet18	20	0.27	0.35	0.43	0.74	0.81	0.86	1.00	19.68	14.78	6.60	4.03	2.27	1.62	0.67
resnet34	20	0.40	0.31	0.52	0.71	0.88	0.90	1.04	18.40	15.37	9.19	6.55	4.29	3.25	1.36
vgg13	20	0.25	0.74	0.96	1.01	1.08	1.04	1.18	23.05	16.25	6.65	4.29	2.47	1.79	0.81
vgg16	20	0.24	0.72	0.88	1.00	1.06	1.14	1.22	21.41	15.14	6.77	4.33	2.53	1.85	0.91
vgg19	20	0.30	0.61	0.89	0.98	1.05	1.25	1.29	21.79	15.62	6.32	4.17	2.43	1.80	1.00
shufflenet	29	0.37	0.64	0.82	0.91	0.96	0.98	0.99	7.54	7.24	6.09	4.11	2.65	2.00	1.33
squeezenet	28	0.26	0.44	0.72	0.82	0.90	0.97	1.00	11.83	10.64	6.04	3.95	2.35	1.79	1.05
resnet101	29	0.28	0.50	0.77	0.94	0.99	1.03	1.08	8.45	7.10	4.39	3.05	2.08	1.63	1.27
resnet152	29	0.27	0.64	0.83	0.94	1.05	1.09	1.17	8.13	7.45	4.37	3.05	2.06	1.79	1.22
resnet50	29	0.22	0.55	0.69	0.84	0.92	0.99	1.07	8.74	7.41	4.19	2.83	1.75	1.30	0.94
mobilenet_v2	34	0.26	0.43	0.70	0.85	0.95	0.97	1.00	6.33	5.89	3.83	2.69	1.77	1.45	1.09
mnasnet1_0	36	0.23	0.42	0.67	0.82	0.96	0.98	1.00	6.42	6.89	3.84	2.39	1.62	1.37	1.19
inception_v3	70	0.30	0.48	0.65	0.76	0.95	0.98	1.01	3.92	3.61	2.90	2.34	1.72	1.45	1.16
googlenet	73	0.25	0.59	0.69	0.85	0.94	0.97	1.01	7.93	5.51	2.87	1.95	1.31	1.07	0.89
densenet121	139	0.50	0.76	0.88	1.00	1.02	1.02	1.03	3.70	3.91	3.68	3.42	3.32	3.25	3.03
densenet161	179	0.50	0.82	0.94	1.04	1.05	1.05	1.06	2.65	2.84	2.84	2.84	2.75	2.76	2.72
densenet169	187	0.49	0.75	0.91	1.02	1.03	1.04	1.04	2.36	2.66	2.61	2.73	2.69	2.68	2.60
densenet201	219	0.47	0.75	0.87	1.00	1.03	1.03	1.03	1.60	1.74	1.91	1.95	1.92	1.93	1.88
GEO	-	0.30	0.56	0.75	0.90	0.97	1.01	1.06	8.59	7.42	4.58	3.33	2.30	1.87	1.24

Fig. 7. Comparative Analysis of Optimization Results on the CUDA (Ampere) Architecture Using an NVIDIA A100 GPU. Numbers show Meta Schedule/DPMS ratios. Thus, results higher than 1.0 denote improvements of DPMS (this paper) over MetaSchedule.

Discussion: 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 AMD Ryzen 7 (x86-64). With a budget of 300 samples, DPMSconsistently outperforms or ties with the MetaSchedule, for a geomean speedup of 9%.

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 MetaSchedule 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 DPMS involves all the steps of MetaSchedule, 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 MetaSchedule and DPMS.

Discussion: AMD Ryzen 7 (x86-64). The right side of Figure 6 compares search times in the x86 hardware. For most models, DPMS is consistently faster than MetaSchedule for any number of trials up to $K = 300$. At 1,000 trials, MetaSchedule becomes consistently faster. Also, MetaSchedule tends to outperform DPMS 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.

Discussion: 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 DPMS is also consistently

Cuda - RTX3080		10k speedup execution time comparison							10k speedup tuning time comparison						
Model	L	Top-1 MS	Top-10 MS	Top-50 MS	Top-100 MS	Top-200 MS	Top-300 MS	Top-1k MS	Top-1 MS	Top-10 MS	Top-50 MS	Top-100 MS	Top-200 MS	Top-300 MS	Top-1k MS
alexnet	14	0.36	0.74	0.80	0.85	0.96	0.98	1.03	43.11	20.55	7.82	4.77	2.82	2.13	1.02
vgg11	18	0.35	0.82	0.88	0.97	1.01	1.07	1.11	34.95	23.71	8.38	5.17	2.95	2.15	0.89
resnet18	20	0.39	0.60	0.71	0.83	0.92	0.93	1.01	26.35	20.19	6.45	3.76	2.09	1.52	0.83
resnet34	20	0.28	0.45	0.65	0.80	0.88	0.98	1.06	31.69	20.04	6.59	3.78	2.27	1.67	0.84
vgg13	20	0.25	0.57	0.86	0.97	1.03	1.07	1.09	28.73	19.57	6.24	3.98	2.34	1.73	0.83
vgg16	20	0.32	0.62	0.89	0.99	1.00	1.00	1.11	28.95	20.22	7.42	4.40	2.50	1.76	0.82
vgg19	20	0.33	0.80	0.95	0.97	1.02	1.09	1.10	33.31	17.13	6.94	4.17	2.45	1.77	0.91
shufflenet	29	0.37	0.55	0.78	0.92	0.98	0.99	1.00	23.20	21.02	8.49	4.92	3.16	2.61	2.01
squeezenet	28	0.32	0.56	0.80	0.87	0.95	0.99	1.00	18.08	19.43	7.08	4.18	2.33	1.78	1.09
resnet101	29	0.37	0.63	0.81	0.91	0.97	0.99	1.02	14.86	12.38	5.62	3.39	2.19	1.77	1.23
resnet152	29	0.40	0.65	0.82	0.91	1.03	1.03	1.03	13.83	13.87	5.47	3.30	2.03	1.72	1.24
resnet50	29	0.37	0.64	0.78	0.91	0.96	1.00	1.03	13.93	11.58	5.16	3.02	1.72	1.29	0.97
mobilenet_v2	34	0.32	0.53	0.68	0.84	0.97	0.98	1.00	11.82	12.17	4.91	2.88	1.72	1.38	1.18
mnasnet1_0	36	0.33	0.51	0.68	0.87	0.96	0.99	1.00	14.59	12.58	4.95	2.73	1.71	1.39	1.22
inception_v3	70	0.39	0.64	0.78	0.93	1.00	1.00	1.01	7.90	9.99	5.46	3.81	2.92	2.70	2.55
googlenet	73	0.33	0.65	0.83	0.95	0.99	1.00	1.00	10.75	9.53	3.37	2.07	1.55	1.46	1.39
densenet121	139	0.57	0.76	0.85	0.98	1.01	1.01	1.01	7.29	5.77	2.57	1.82	1.59	1.48	1.36
densenet161	179	0.57	0.79	0.92	1.01	1.01	1.01	1.01	4.95	5.82	6.39	6.32	6.11	5.95	5.98
densenet169	187	0.56	0.76	0.91	1.01	1.01	1.01	1.01	4.43	5.90	6.12	6.37	6.51	6.29	6.07
densenet201	219	0.54	0.80	0.95	1.01	1.01	1.01	1.01	4.08	5.46	5.80	6.14	6.01	6.00	5.97
GEO	-	0.38	0.64	0.81	0.92	0.98	1.01	1.03	15.14	12.93	5.85	3.84	2.57	2.10	1.46

Fig. 8. Comparative Analysis of Optimization Results on the CUDA (Ampere) Architecture Using an NVIDIA RTX3080 GPU. Numbers show Meta Schedule/DPMS ratios. Thus, results higher than 1.0 denote improvements of DPMS (this paper) over MetaSchedule.

faster than MetaSchedule in every setup where $K \leq 300$. At 1,000 samples, the relative benefit that DPMS brings onto MetaSchedule disappears. However, even in this scenario, both approaches present almost the same search time.

Discussion: 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 DPMS is also consistently faster than MetaSchedule in every setup where $K \leq 300$. In some cases, at 1000 samples, there are improvements.

Discussion: 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 DPMS to 300 points, we achieve an average speedup of 1.94x (geometric mean). Again, in some individual cases, speedups are noticeable. In particular, for the small models, DPMS is more than twice faster than MetaSchedule, even with a budget of 300 trials.

3.3 RQ3 – On the Impact of Model Size

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

(1) The larger the model, the less samples DPMS needs to observe to outperform MetaSchedule, if MetaSchedule uses a budget of 10,000 samples.
(2) The larger the model, the lower the benefit, in terms of search time, of DPMS over MetaSchedule.

The rest of this section provides data to support these two conclusions.

ARM - A64FX		10k speedup execution time comparison							10k speedup tuning time comparison						
Model	L	Top-1 MS	Top-10 MS	Top-50 MS	Top-100 MS	Top-200 MS	Top-300 MS	Top-1k MS	Top-1 MS	Top-10 MS	Top-50 MS	Top-100 MS	Top-200 MS	Top-300 MS	Top-1k MS
alexnet	14	0.31	0.58	0.78	0.92	1.03	1.02	1.12	18.54	14.02	6.82	4.49	2.92	2.18	0.95
vgg11	18	0.27	0.55	0.68	0.76	0.82	0.87	1.00	7.59	8.67	5.94	4.39	3.15	2.46	1.19
resnet18	20	0.43	0.57	0.74	0.85	0.99	1.05	1.20	11.15	10.54	5.85	4.23	2.77	2.10	1.18
resnet34	20	0.38	0.65	0.71	0.89	1.07	1.19	1.27	11.12	9.69	6.12	4.34	2.78	2.14	1.21
vgg13	20	0.29	0.53	0.79	0.84	0.92	0.95	1.11	3.88	6.34	4.45	3.74	2.46	2.01	1.05
vgg16	20	0.31	0.59	0.69	0.75	0.89	0.92	1.01	5.20	6.91	4.70	3.84	2.85	2.25	1.32
vgg19	20	0.21	0.54	0.69	0.85	0.93	0.98	1.04	6.12	6.30	4.74	3.84	2.58	2.14	1.30
shufflenet	29	0.46	0.81	0.97	1.07	1.16	1.21	1.24	11.72	10.25	5.95	4.07	2.73	2.17	1.48
squeezenet	28	0.26	0.54	0.75	1.01	1.11	1.15	1.19	11.36	10.08	4.80	3.02	1.95	1.62	1.27
resnet101	29	0.39	0.68	0.90	1.01	1.14	1.16	1.22	5.95	6.19	4.30	3.38	2.49	2.29	1.72
resnet152	29	0.40	0.76	0.79	0.89	1.06	1.15	1.20	6.16	6.30	4.43	3.46	2.52	2.20	1.68
resnet50	29	0.41	0.65	0.70	0.84	1.02	1.12	1.16	6.17	6.20	4.42	3.43	2.32	1.91	1.61
mobilenet_v2	34	0.34	0.62	0.77	1.01	1.12	1.13	1.15	6.94	6.32	3.77	2.49	1.73	1.47	1.33
mnasnet1_0	36	0.39	0.62	0.74	0.94	1.07	1.12	1.13	6.45	5.68	3.47	2.40	1.65	1.44	1.32
inception_v3	70	0.40	0.74	0.88	0.98	1.08	1.10	1.06	4.20	4.14	2.96	2.48	2.06	1.91	1.70
googlenet	73	0.36	0.61	0.81	0.98	1.06	1.08	1.12	4.10	3.88	2.84	2.29	1.93	1.72	1.62
densenet121	139	0.52	0.75	0.90	1.08	1.14	1.14	1.10	3.19	3.14	2.52	2.17	1.99	1.92	1.77
densenet161	179	0.55	0.80	1.01	1.15	1.18	1.19	1.19	2.18	2.22	1.89	1.79	1.72	1.67	1.60
densenet169	187	0.44	0.86	1.03	1.11	1.15	1.16	1.13	2.36	2.37	2.13	2.00	1.88	1.82	1.84
densenet201	219	0.46	0.87	1.06	1.19	1.21	1.20	1.19	1.99	2.05	1.93	1.78	1.74	1.75	1.71
GEO	-	0.37	0.66	0.81	0.95	1.05	1.09	1.14	5.76	5.77	3.92	3.04	2.26	1.94	1.42

Fig. 9. Comparative Analysis of Optimization Results on an Aarch64 Architecture Using an A64FX Processor. Numbers show Meta Schedule/DPMS ratios. Thus, results higher than 1.0 denote improvements of DPMS (this paper) over MetaSchedule.

Discussion: The Search vs Quality Slope. The kernel optimization technique impacts two core numbers: the search time and the speed of the final model. We can use these two quantities—search time (S) and model performance (P)—to define an $S \times P$ line characterizing the behavior of the optimization technique. 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 DPMS performs better, and Ampere is the scenario where it performs worse.

Figure 10 uses our two smallest and two largest models. The numbers on the axes show ratios between DPMS and MetaSchedule; the latter using a budget of 10,000 trials. Each dot in Figure 10 refers to the number of trials that DPMS is allowed to observe before shifting to Droplet Search. The figure labels dots that refer to DPMS 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 DPMS, the difference between its search time and MetaSchedule'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 DPMS over MetaSchedule in terms of search time; but the higher the relative benefit in terms of kernel speed. This result is due to MetaSchedule'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.

3.4 RQ4: The Average Number of Droplet Search Samples

Droplet Search has a convergence criterion, which is discussed in Section 3.3 of its description [Canesche et al. 2024]. In this case, search stops once there is no statistically significant

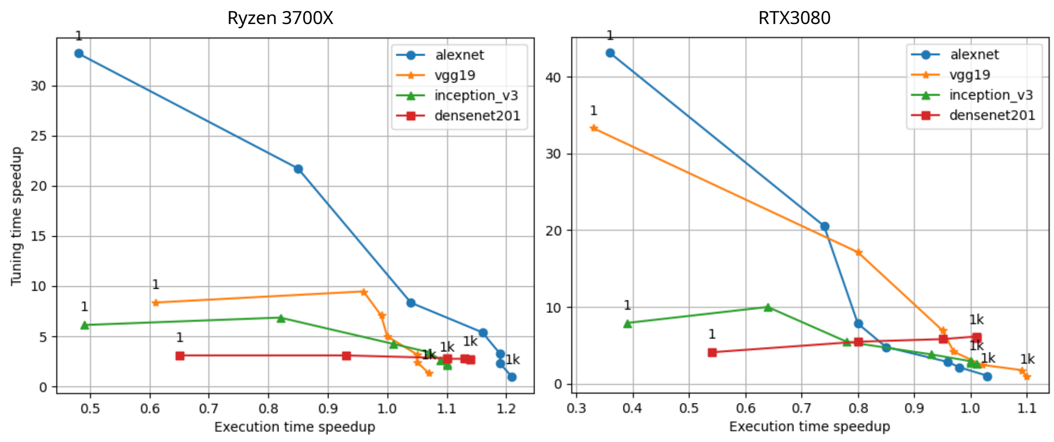


Fig. 10. The search vs quality line that characterizes four models optimized in the AMD (Left) and in the NVIDIA (Right) setting: the larger the model, the lower the slope. Numbers on the X and Y axes are speedup/slowdown relative to MetaSchedule with a budget of 10,000 trials. Numbers for the left chart are available in Figure 6, and numbers for the right chart are available in Figure 8. The labels on the dots (1 and 1k) refer to the number of trials given to DPMS.

difference between the current kernel and the kernels within its neighborhood. Thus, the more optimized is the seed of the coordinate descent algorithm, the less iterations it is—intuitively—expected to take until convergence. This section investigates if this hypothesis is true.

Discussion. We count the number of iterations of Droplet Search on the different architectures that we have, considering different budges for DPMS. Figure 11 shows this number for each model evaluated on the x86 setting. The figure reports total number of trials and average number of trials per layer. The general tendency is that the larger the budget of trials allocated to DPMS, the faster Droplet Search converges.

Figure 12 summarizes, for each architecture, the numbers earlier seen in Figure 11. Figure 12 reports averages per model (considering the 20 available models), and averages per layer. In the latter case, we divide the total sum of trials observed for all the models by the sum of the number of layers present in every model. In every case, the same tendency is evidence: more trials sampled during exploration imply in less trials sampled during exploitation. This result is intuitive: as previously mentioned, Droplet Search tends to reach stability the closer to a local optimum it starts.

4 CONCLUSION

This report has described our experience adding an exploitation methodology on top of the kernel scheduling strategy already in place in the MetaSchedule optimizer. The new approach uses MetaSchedule to explore different spaces of kernel schedules, chooses the best candidate found thus far, and uses Droplet Search to further exploit it. This methodology benefits MetaSchedule and Droplet Search:

- It enhances MetaSchedule's capability to exploit "hardware boundaries". The previous implementation of MetaSchedule was not aware of the relationships between neighboring kernel schedules, as it lacked a concept of "distance" between the implementatios of kernels. The new exploration phase improves MetaSchedule's ability to better adjust optimization parameters to hardware constraints, such as cache sizes and vector widths.

x86-64		Sum of trials							Average trials per layer						
Model	L	Top-1 MS	Top-10 MS	Top-50 MS	Top-100 MS	Top-200 MS	Top-300 MS	Top-1k MS	Top-1 MS	Top-10 MS	Top-50 MS	Top-100 MS	Top-200 MS	Top-300 MS	Top-1k MS
alexnet	13	187	201	205	199	169	162	171	14	15	16	15	13	12	13
vgg11	17	232	210	185	172	158	158	160	14	12	11	10	9	9	9
resnet18	19	428	437	441	351	316	328	318	23	23	23	18	17	17	17
resnet34	19	427	419	359	387	428	350	351	22	22	19	20	23	18	18
vgg13	19	245	274	245	213	190	177	187	13	14	13	11	10	9	10
vgg16	19	263	228	230	193	218	185	186	14	12	12	10	11	10	10
vgg19	19	216	240	215	233	207	189	204	11	13	11	12	11	10	11
shufflenet	20	295	297	256	287	271	275	272	15	15	13	14	14	14	14
squeezenet	23	489	557	390	421	381	370	364	21	24	17	18	17	16	16
resnet101	28	818	860	759	694	674	684	655	29	31	27	25	24	24	23
resnet152	28	841	726	784	699	687	704	720	30	26	28	25	25	25	26
resnet50	28	925	839	784	757	643	657	644	33	30	28	27	23	23	23
mobilenet_v2	33	524	532	606	603	486	539	524	16	16	18	18	15	16	16
mnasnet1_0	36	706	678	696	706	653	678	685	20	19	19	20	18	19	19
inception_v3	56	1410	1393	1344	1239	1272	1263	1270	25	25	24	22	23	23	23
googlenet	62	1430	1384	1350	1333	1335	1311	1310	23	22	22	22	22	21	21
densenet121	73	2070	1823	1768	1648	1756	1696	1736	28	25	24	23	24	23	24
densenet161	93	2530	2381	2359	2269	2282	2271	2280	27	26	25	24	25	24	25
densenet169	97	2665	2513	2419	2435	2387	2485	2484	27	26	25	25	25	26	26
densenet201	113	3110	3004	2846	2955	3063	2890	2946	28	27	25	26	27	26	26
AVG	-	991	950	912	890	879	869	873	22	21	20	19	19	18	18

Fig. 11. Number of trials sampled by Droplet Search until reaching convergence, considering the AMD x86-64 architecture. The average number of trials per layer divides the total number of trials per the number of layers in each deep learning model.

	Sum of trials							Average trials per layer						
Arch	Top-1 MS	Top-10 MS	Top-50 MS	Top-100 MS	Top-200 MS	Top-300 MS	Top-1k MS	Top-1 MS	Top-10 MS	Top-50 MS	Top-100 MS	Top-200 MS	Top-300 MS	Top-1k MS
x86-64	19811	18996	18241	17794	17576	17372	17467	24	23	22	22	22	21	21
ARM A64FX	19610	17785	15768	14450	13506	13683	13072	24	22	19	18	17	17	16
Cuda RTX3080	17803	13982	11886	10965	10779	10751	10819	22	17	15	13	13	13	13
Cuda A100	24831	20278	17147	14655	13731	14110	13832	30	25	21	18	17	17	17

Fig. 12. Average number of trials per model (considering 20 models) for different architectures. The averages per layer are the quotient of the total number of trials for all the models divided by the total number of layers in all the models.

- It addresses Droplet Search's two limitations: its reliance on a well-defined *seed* (the initial kernel that initiates coordinate descent), and its inability to explore different kernel spaces. Previously, the seed and the kernel space were determined manually, requiring a programmer to provide Droplet Search with an initial annotated sketch. The proposed methodology automates the seed generation process by utilizing MetaSchedule.

As detailed in Section 2.4, the proposed extension does not impact current Apache TVM users, because the use of Droplet Search with MetaSchedule is optional. As Section 3 demonstrates, the proposed extension improves MetaSchedule in terms of kernel quality and search time.

REFERENCES

AMD. 2019. AMD Ryzen^{TM} 7 3700X. https://www.amd.com/en/product/8446. [Online; accessed 18-Jan-2024].

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In ASPLOS. ACM, New York, USA, 623–630.

Michael Canesche, Vanderson M. Rosario, Edson Borin, and Fernando Magno Quintão Pereira. 2024. The Droplet Search Algorithm for Kernel Scheduling. , 25 pages. https://doi.org/10.1145/3650109 Just Accepted.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In OSDI. USENIX, Berkeley, USA, 578–594.

Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. 2023. Hidet: Task-Mapping Programming Paradigm for Deep Learning Tensor Programs. In ASPLOS (Vancouver, BC, Canada). Association for Computing Machinery, New York, NY, USA, 370–384. https://doi.org/10.1145/3575693.3575702

Nvidia. 2020a. Nvidia A100 Tensor Core GPU. https://www.nvidia.com/en-in/data-center/a100/. [Online; accessed 18-Jan-2024].

Nvidia. 2020b. Nvidia RTX3080 Tensor Core GPU. https://www.nvidia.com/en-in/geforce/graphics-cards/30-series/. [Online; accessed 18-Jan-2024].

Ookami. 2022. ACCESS Research Provider, A64FX Cluster. https://www.stonybrook.edu/ookami/. [Online; accessed 18-Jan-2024].

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. SIGPLAN Not. 48, 6 (jun 2013), 519–530. https://doi.org/10.1145/2499370.2462176

Christian Sarofeen, Piotr Bialecki, Jie Jiang, Kevin Stephano, Masaki Kozuki, Neal Vaidya, and Stas Bekman. 2022. Introducing nvFuser, a deep learning compiler for PyTorch. https://pytorch.org/blog/introducing-nvfuser-a-deep-learning-compiler-for-pytorch/. [Online; accessed 15-Mar-2024].

Tyler Sorensen, Sreepathi Pai, and Alastair F. Donaldson. 2019. One Size Doesn't Fit All: Quantifying Performance Portability of Graph Applications on GPUs. In IISWC. IEEE, New York, US, 155–166. https://doi.org/10.1109/IISWC47752.2019.9042139

Stephen J. Wright. 2015. Coordinate Descent Algorithms. Math. Program. 151, 1 (jun 2015), 3–34. https://doi.org/10.1007/s10107-015-0892-3

W. Zangwill. 1969. Nonlinear Programming, A Unified Approach (1st ed.). Prentice Hall, USA.

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 OSDI. USENIX Association, USA, Article 49, 17 pages.

Mikhail Zolotukhin. 2021. NNC walkthrough: how PyTorch ops get fused. https://dev-discuss.pytorch.org/t/nnc-walkthrough-how-pytorch-ops-get-fused/125. [Online; accessed 15-Mar-2024].