The Droplet Search Algorithm for Kernel Scheduling

MICHAEL CANESCHE, UFMG, Brazil
VANDERSON M. ROSARIO, Cadence Design Systems, USA
EDSON BORIN, Unicamp, Brazil
FERNANDO MAGNO QUINTÃO PEREIRA, UFMG, Brazil

Kernel scheduling is the problem of finding the most efficient implementation for a computational kernel. Identifying this implementation involves experimenting with the parameters of compiler optimizations, such as the size of tiling windows and unrolling factors. This paper shows that it is possible to organize these parameters as points in a coordinate space. The function that maps these points to the running time of kernels, in general, will not determine a convex surface. However, this paper provides empirical evidence that the origin of this surface—an unoptimized kernel—and its global optimum—the fastest kernel—reside on a convex region. We call this hypothesis the "droplet expectation". Consequently, a search method based on the coordinate descent algorithm tends to find the optimal kernel configuration quickly if the hypothesis holds. This approach—called Droplet Search—has been available in Apache TVM since April of 2023. Experimental results with six large deep learning models on various computing devices (ARM, Intel, AMD, and NVIDIA) indicate that Droplet Search is not only as effective as other AutoTVM search techniques but also two to ten times faster. Moreover, models generated by Droplet Search are competitive with those produced by TVM's AutoScheduler (Ansor), despite the latter using four to five times more code transformations than AutoTVM.

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, Vanderson M. Rosario, Edson Borin, and Fernando Magno Quintão Pereira. 2024. The Droplet Search Algorithm for Kernel Scheduling. *ACM Trans. Arch. Code Optim.* 1, 1, Article 1 (January 2024), 28 pages. https://doi.org/10.1145/3650109

1 INTRODUCTION

In the context of this paper, a *kernel* is a function that reads and write data indexed by a linear combination of natural numbers. Kernels are typically implemented as nests of affine loops. Examples of kernels include matrix multiplication, transposition, and convolutions. Following Jin et al. [2022], we say that a deep learning model is a function implemented as alternating layers of kernels and non-linear functions (sigmoid, ReLU, *etc.*). Examples of deep-learning models include neural networks, such as BERT [Devlin et al. 2019], ResNet-18 [He et al. 2015], VGG-16 [Sengupta et al. 2018], MobileNet [Howard et al. 2017], and MXNet [Chen et al. 2015].

Authors' addresses: Michael Canesche, UFMG, Brazil, michael.canesche@dcc.ufmg.br; Vanderson M. Rosario, Cadence Design Systems, USA, vrosario@cadence.com; Edson Borin, Unicamp, Brazil, edson@ic.unicamp.br; Fernando Magno Quintão Pereira, UFMG, Brazil, fernando@dcc.ufmg.br.

XXXX-XXXX/2024/1-ART1 $15.00
https://doi.org/10.1145/3650109

The Space of Kernel Schedulings. A kernel is an abstract concept that supports different concrete implementations [Jin et al. 2022]. Each implementation, in this paper, is called a *kernel schedule*. The set of every schedule of a kernel is called its *search space*. The choice of implementation impacts the performance of the kernel. Finding the best schedule for a kernel is an optimization problem whose objective function is running time: the faster a kernel runs, the better that schedule is. The problem of finding exact analytical solutions to kernel scheduling is open, even when fixing the computer architecture [Tollenaere et al. 2023]. Hence, typical techniques are stochastic, take time to converge and provide no guarantees of optimality [Lebedev and Belecky 2021].

Coordinates and Neighborhoods. Kernel schedules differ due to the application of code transformations, such as tiling, unrolling, and thread blocking. If we fix the sequence of transformations that define the search space, then each schedule is uniquely determined by the transformation parameters: unrolling factor and tiling window per loop, number of threads per block, *etc.* These parameters admit a total order: if $m < n, n, m \in \mathbb{N}$, then an unrolling factor of n is larger than an unrolling factor of m. This ordering determines *coordinates* on the space of kernel schedules; hence, yielding a notion of a neighborhood. The *neighborhood function* relates kernel configurations produced by transformation vectors that differ by a minimal difference on one parameter.

The Key Observation and the Implied Hypothesis: In a *convex optimization space*, every local minimum is a global minimum. The space of kernel schedules is usually not convex, as Example 3.5, in Section 2, will show. However, we believe that the following expectation applies to the vast majority of machine learning models: *It is possible to project the set of all kernel schedules onto a system of coordinates, such that the region between the origin of this space and the fastest kernel configuration form a convex hypersurface with respect to the running time of the kernels.* Hence, the optimum configuration can be reached from the origin by deriving the running time function along a continuous neighborhood of kernel configurations. We call this observation the *Droplet Expectation*, and formalize it in Section 3 (see Definition 3.6 on Page 8).

Based on this expectation, this paper describes a scheduling technique called *Droplet Search*, which is currently part of Apache TVM[1]. This paper evaluates Droplet Search on six architectures (two x86 CPUs, two ARM CPUs, and two NVIDIA GPUs); and on six models (BERT, ResNet-18, VGG-16, MobileNet, MXNet, and Inception-v3). Section 4 shows that Droplet Search runs up to ten times faster than the other four search algorithms in AutoTVM [Chen et al. 2018] and the search algorithm in TVM's Ansor [Zheng et al. 2020]. The kernels produced via Droplet Search tend to outperform those produced by the other search approaches in AutoTVM, and approximate those produced by Ansor, even though the latter might use up to four times more transformation parameters. These results, summarized in Figure 11 (Page 15), come from the following contributions:

- **Intuition:** the droplet expectation is not a theorem: in Section 4.4, we show that it is possible to disprove it using analytical cost models involving discontinuous functions. However, the hypothesis is expected to hold in cost models described by contiguous functions involving only positive domain and coefficients: a property that Renganarayana and Rajopadhye [2008] call "*Positivity*". These models are rather common: Table II in Renganarayana and Rajopadhye's manuscript lists eight of them from previous work. More recent models [Olivry et al. 2021, 2020] share similar properties, as Section 4.4 discusses.
- **Simplicity:** the implementation of droplet's search in AutoTVM 0.14.0 (the pseudo code in Figure 4, Page 6) consists of 127 lines of commented Python code. For comparison, the implementation of TVM XGB's search [Chen et al. 2018] uses 971 lines in three files (`xgboost_tuner.py`, `xgboost_cost_model.py` and `sa_model_optimizer.py`).

[1]Release v0.13.0 (https://github.com/apache/tvm/pull/14683).

Adaptability: Droplet Search works in any setting where AutoTVM does, including settings like a Cortex A7, where Ansor cannot be used to optimize MobileNet [Howard et al. 2017].

Efficiency: Droplet Search converges more than twice as fast as the different algorithms available in AutoTVM (random sampling, grid search, genetic search, and XGBoost) and usually is more than four times faster than TVM's AutoScheduler.

Effectiveness: in almost every experiment we ran, Droplet Search delivered kernels that either match or outperform those produced by the other schedulers in AutoTVM. Compared to Ansor, in a universe of 30 experiments, Droplet Search found faster kernels in eleven settings, and lost in twelve. However, Ansor uses up to four times more transformations.

2 THE SEARCH SPACE

A kernel admits an *abstract view*, formed by an iteration space, a data space, and a computation constrained into these zones[2]. As mentioned in Section 1, this view can be implemented in multiple ways as long as the data dependencies encoded in the kernel's computation are respected. Each implementation differs in how the iteration space is traversed. This scheduling determines how the kernel's computation updates the data space. Example 2.1 clarifies these notions.

Example 2.1. Figure 1 shows an abstract view of the matrix multiplication kernel. The computations performed by the kernel can be indexed by triples (i, j, k), which form its iteration space. The bounds R_A, C_B, and C_A that delimit this space abide by two constraints. The first, $C_a = R_B$, is mandatory for correctness; the second—R_A is even—we use for the sake of the example. Figure 1 also shows five implementations of the abstract kernel. These implementations produce the same matrix `C`; however, the order in which the computations occur—*the kernel schedule*—might vary.

(a)

Data space:

$A \subseteq (0 \dots R_A) \times (0 \dots C_A)$

$B \subseteq (0 \dots R_B) \times (0 \dots C_B)$

$C \subseteq (0 \dots R_C) \times (0 \dots C_C)$

Iteration space:

$i \in (0 \dots R_A)$

$j \in (0 \dots C_B)$

$k \in (0 \dots C_A)$

Computation:

$C_{ij} = C_{ij} + A_{ik} \times B_{kj}$

Constraints:

$C_A = R_B, R_A \% 2 = 0$

(b)

```
void mm(f64* A, f64* B, f64* C,
        int RA, int CA, int CB) {
  for (int i0 = 0; i0 < RA; i0++) {
    for (int j0 = 0; j0 < CB; j0++) {
      for (int k0 = 0; k0 < CA; k0++) {
        int ij = i0 * CB + j0;
        int ik = i0 * CA + k0;
        int kj = k0 * CB + j0;
        C[ij] += A[ik] * B[kj];
} } } }
```

(c)

```
void mm_jki(f64* A, f64* B, f64* C,
            int RA, int CA, int CB) {
  for (int j0 = 0; j0 < CB; j0++) {
    for (int k0 = 0; k0 < CA; k0++) {
      for (int i0 = 0; i0 < RA; i0++) {
        int ij = i0 * CB + j0;
        int ik = i0 * CA + k0;
        int kj = k0 * CB + j0;
        C[ij] += A[ik] * B[kj];
} } } }
```

(d)

```
void mm_jki_tile_16(f64* A, f64* B, f64* C,
                    int RA, int CA, int CB) {
  for (int j0 = 0; j0 < CB; j0 += 16) {
    for (int k0 = 0; k0 < CA; k0 += 32) {
      for (int i0 = 0; i0 < RA; i0 += 16) {
        int jmax = j0 + 16 > CB ? CB : j0 + 16;
        for (int j1 = j0; j1 < jmax; ++j1) {
          int kmax = k0 + 32 > CA ? CA : k0 + 32;
          for (int k1 = k0; k1 < kmax; ++k1) {
            int imax = i0 + 16 > RA ? RA : i0 + 16;
            for (int i1 = i0; i1 < imax; ++i1) {
              int ij = i1 * CB + j1;
              int ik = i1 * CA + k1;
              int kj = k1 * CB + j1;
              C[ij] += A[ik] * B[kj];
} } } } } } }
```

(e)

```
void mm_jki_tile_8(f64* A, f64* B, f64* C,
                   int RA, int CA, int CB) {
  for (int j0 = 0; j0 < CB; j0 += 8) {
    for (int k0 = 0; k0 < CA; k0 += 32) {
      for (int i0 = 0; i0 < RA; i0 += 16) {
        int jmax = j0 + 8 > CB ? CB : j0 + 8;
        for (int j1 = j0; j1 < jmax; ++j1) {
          int kmax = k0 + 32 > CA ? CA : k0 + 32;
          for (int k1 = k0; k1 < kmax; ++k1) {
            int imax = i0 + 16 > RA ? RA : i0 + 16;
            for (int i1 = i0; i1 < imax; ++i1) {
              int ij = i1 * CB + j1;
              int ik = i1 * CA + k1;
              int kj = k1 * CB + j1;
              C[ij] += A[ik] * B[kj];
} } } } } } }
```

(f)

```
void mm_jki_roll(f64* A, f64* B, f64* C,
                 int RA, int CA, int CB) {
  for (int j0 = 0; j0 < CB; j0++) {
    for (int k0 = 0; k0 < CA; k0++) {
      for (int i0 = 0; i0 < RA; i0 += 2) {
        int ij = i0 * CB + j0;
        int ik = i0 * CA + k0;
        int kj = k0 * CB + j0;
        C[ij] += A[ik] * B[kj];
        ij = (i0+1) * CB + j0;
        ik = (i0+1) * CA + k0;
        kj = k0 * CB + j0;
        C[ij] += A[ik] * B[kj];
} } } }
```

Fig. 1. (a) The abstract representation of the matrix multiplication kernel. (b-c) Canonical schedules of the kernel. (d-f) Schedules that result from applying loop tiling and loop unrolling with different parameters onto the canonical schedule in (c).

[2] Notions of iteration and data space are standard in the compiler literature. Such concepts appeared independently in the works of Feautrier [1991] and Wolf and Lam [1991], eventually leading to the concept known today as the *Polyhedral Model*.

Transformation Vectors. As hinted in Example 2.1, in the context of this paper, the implementation of kernels differ concerning code transformations. These transformations are guided by *parameters*. Example 2.2 shows parameters for two well-known transformations: tiling and unrolling.

Example 2.2. Figure 1 (e) shows the kernel that comes from Figure 1 (c) after the application of three instances of tiling: a transformation that partitions the iteration space into smaller regions. In this example, tiling happens along the three axes of the iteration space. The dimensions of the tiling window are 8, 32, and 16 points. Each one of these sizes is an optimization parameter. Figure 1 (f) shows the kernel produced after an application of unrolling onto the innermost loop of the kernel in Figure 1 (c). The unrolling factor, *i.e.*, the transformation parameter, is 2.

The parameters of code transformations can be organized into *transformation vectors*. A transformation vector is a tuple whose elements represent these parameters. The order in which these parameters appear in the vector determines the order in which transformations are applied to programs. Thus, if a given code transformation has multiple parameters, then these parameters exist sequentially in the transformation vector. Example 2.3 shows how the parameters seen in Example 2.2 can be represented as transformation vectors.

Example 2.3. Consider the ordered application of the two loop-related compiler optimizations mentioned in Example 2.2 onto the abstract kernel in Figure 1 (a): tiling along dimensions A, B, and C, and unrolling on dimensions A, B, and C. The vectors representing any application of these code transformations have the format: $\langle \texttt{tile}_A : t_A, \texttt{tile}_B : t_B, \texttt{tile}_C : t_C, \texttt{roll}_A : r_A, \texttt{roll}_B : r_B, \texttt{roll}_C : r_C \rangle$. Figure 2 shows the vectors that produce the kernels in Figure 1.

Fig-1c $\langle \text{tile}_A\text{: } \mathit{0}, \text{tile}_B\text{: } \mathit{0}, \text{tile}_C\text{: } \mathit{0}, \text{roll}_A\text{: } \mathit{0}, \text{roll}_B\text{: } \mathit{0}, \text{roll}_C\text{: } \mathit{0} \rangle$

Fig-1d $\langle \text{tile}_A\text{: } \mathit{16}, \text{tile}_B\text{: } \mathit{16}, \text{tile}_C\text{: } \mathit{32}, \text{roll}_A\text{: } \mathit{0}, \text{roll}_B\text{: } \mathit{0}, \text{roll}_C\text{: } \mathit{0} \rangle$

Fig-1e $\langle \text{tile}_A\text{: } \mathit{16}, \text{tile}_B\text{: } \mathit{8}, \text{tile}_C\text{: } \mathit{32}, \text{roll}_A\text{: } \mathit{0}, \text{roll}_B\text{: } \mathit{0}, \text{roll}_C\text{: } \mathit{0} \rangle$

Fig-1f $\langle \text{tile}_A\text{: } \mathit{0}, \text{tile}_B\text{: } \mathit{0}, \text{tile}_C\text{: } \mathit{0}, \text{roll}_A\text{: } \mathit{2}, \text{roll}_B\text{: } \mathit{0}, \text{roll}_C\text{: } \mathit{0} \rangle$

Fig. 2. The transformation vectors that produce kernels in Figure 1.

The iteration space of an abstract kernel can be traversed in many ways. If we ascribe one loop per dimension of the iteration space, any ordering of these loops that respects data dependencies is a valid traversal. The implementation of any such ordering, subject to a null transformation vector (the vector representing the absence of transformations), is called a *canonical schedule*. A kernel may have more than one canonical schedule, as Example 2.4 illustrates. Canonical schedules lead to *Kernel Transformation Spaces*, which Definition 2.5 formalizes.

Example 2.4. Figures 1 (b) and (c) show canonical schedules for the abstract kernel in Figure 1 (a). Any permutations of the loops in Lines 03-05 is a valid canonical schedule. Canonical schedules, by definition, are not subject to code transformations: they represent the application of a null transformation vector. As a consequence, they do not show the effects of typical compiler optimizations, such as loop invariant code motion. Thus, the invariant code in Line 06 of Figure 1 (b) remains inside the innermost loop.

Definition 2.5 (Kernel Transformation Space). Let $\langle \mathcal{P}_1 : p_1, \mathcal{P}_2 : p_2, \ldots, \mathcal{P}_n : p_n \rangle$ be a transformation vector, such that each parameter p_i comes from a range of parameters $\mathcal{P}_i$. The Cartesian product $\mathcal{P}_1 \times \mathcal{P}_2 \times \ldots \times \mathcal{P}_n$ plus a canonical kernel $\mathcal{K}$ form a kernel transformation space. Each point of this space represents a configuration of $\mathcal{K}$ transformed by an instance of that Cartesian product. $\mathcal{K}$ is called the *origin* of this space and the set of transformation parameters is called the *basis* of this space. If the basis contains n parameters, the space is called n-dimensional.

Example 2.6. The vectors in Example 2.3, plus the canonical kernel in Figure 1 (c), form a six-dimensional transformation space. The origin of this space is the kernel in Figure 1 (c), and its basis is formed by three parameters of loop tiling, and three parameters of loop unrolling.

Not every sequence of code transformations is valid. For instance, unrolling a loop by a factor larger than that loop's trip count is not meaningful. Thus, Definition 2.5 implicitly assumes that the transformation space is produced by *valid* transformation vectors. Furthermore, the set of kernels determined by Definition 2.5 is non-exhaustive, even under bounded parameters. Non-exhaustiveness follows from two simplifications that we enumerate below, which are assumed in the construction of the transformation space. These simplifications are also adopted in AutoTVM [Chen et al. 2018], and have been incorporated into further work inspired by it [Zhang et al. 2021]:

(1) Code transformations are applied to the canonical kernel always in a fixed order. In other words, it is possible that by varying the order in which transformations are applied, the space could be extended with new concrete kernels.
(2) Every point in the transformation space comes directly from the canonical kernel via the application of one transformation vector. Notice that successive applications of optimizations could, in principle, produce kernels outside the transformation space.

The performance of the kernels that form the transformation space might vary. These variations depend on the scheduling, on the target architecture and on the dimensions of the iteration and data spaces. The problem of finding the best implementation of a kernel, given these constraints, is known as the *Kernel Scheduling Problem*, a notion extensively discussed in previous work (see Section 5 for references). To keep this paper self-contained, Definition 2.7 restates this concept. Search strategies for Kernel Scheduling are key to optimize deep learning models. Thus, Definition 2.7 is a well-researched problem. Example 2.8 revisits some of these techniques.

Definition 2.7 (Kernel Scheduling Problem). Given a computing device, a transformation space with origin $\mathcal{K}$ and basis $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$, plus valid inputs for $\mathcal{K}$, kernel scheduling asks for the fastest implementation of $\mathcal{K}$ in the transformation space.

Example 2.8. Figure 3 provides a pictographic metaphor for different search techniques for the kernel scheduling problem implemented in AutoTVM, and contrasts them with the Droplet Search method that this paper introduces. Section 3 shall describe the droplet algorithm. In regards to the other methodologies, the search proceeds as follows:

Grid: explores a bounded region of the transformation space. Regular ranges of transformation parameters determine this region.

Random: points of the transformation space are sampled randomly. Sampling usually follows a uniform distribution on predefined bounds placed onto the parameters.

Annealing: Simulated annealing [Kirkpatrick et al. 1988] is a refinement of random search, where sampling alternates between regions that are close and distant from current best points. Points within a neighborhood are sampled, and, from time to time, the center of this neighborhood changes. The probability of such large jumps happening decreases with time.

3 DROPLET SEARCH

The Droplet Search is a greedy heuristic to explore the kernel transformation space. The proposed technique is a variation of the *Coordinate Descent* Algorithm[3], with extensions proposed by Richtárik and Takác [2012] to enable synchronous parallelism. Figure 4 provides an overview of the search

[3]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].

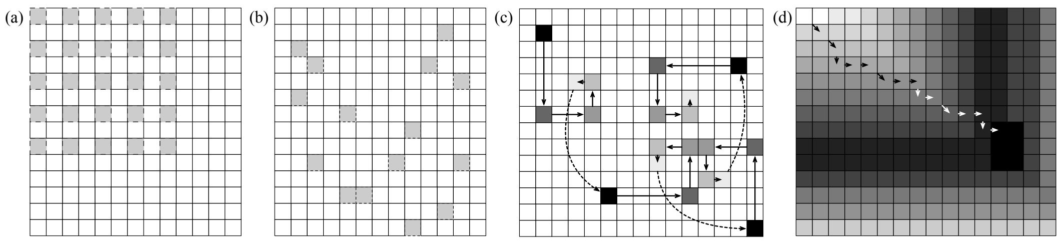


Fig. 3. Pictographic metaphors for different search algorithms. (a,b) Grid search and Random sampling: every configuration within the gray area—and only those—will be evaluated. (c) Simulated Annealing: solid lines represent moves that go "down-hill", *i.e.*, towards faster kernel configurations; dashed edges represent moves that go "up-hill", *i.e.*, which accept to explore slower configurations to escape from local minima. (d) Droplet Search: the darker the color of a configuration, the faster the running time of that configuration.

algorithm evaluated in this paper, and Figure 3 (d) provides a graphical metaphor to illustrate how the algorithm works: at each step—up to a maximum fixed number of iterations—search chooses the most profitable kernel configuration that is considered a "neighbor" of the current best configuration. The rest of this section discusses each part of this algorithm shown in Figure 4.

```
def search(num_iterations):
  candidate = origin                                                  # The origin of the coordinate descent is the
  visited = set()                                                     # canonical kernel from Def. 2.5
  for i in range(num_iterations):
    neighborhood = get_neighbors(candidate).difference(visited)       # The neighborhood function from Section 3.1
    if len(neighborhood) < num_threads():                             # grabs one kernel to evaluate per thread
      neighborhood.union(speculate()).difference(visited)             # The `visited` set avoids double evaluation
    visited.union(neighborhood)
    ps = map(lambda n: Process(target=n.run).start(), neighborhood)   # Synchronous parallelism through speculation
    map(lambda p: p.join(), ps)                                       # is the subject of Section 3.4
    new_candidate = min(neighborhood, key=lambda e: e.running_time)
    if is_statistically_faster(new_candidate, candidate, 0.05):       # Stop criteria are discussed in  Section 3.3
      candidate = new_candidate                                       # We use a confidence level of 0.05 to
    else:                                                             # distinguish the running time of samples.
      return candidate
```

Fig. 4. Droplet Search, a Coordinate Descent variation proposed in this paper to solve the Scheduling Problem from Definition 2.7.

The successful application of Coordinate Descent depends on a suitable "neighborhood function", subject of Section 3.1. Search tends to converge to a global optimal, based on the intuition that Section 3.2 provides. Convergence is based on criteria discussed in Section 3.3. This closes with discussions of two optimizations that speed search up: parallelization and speculation (Sec. 3.4).

3.1 The Neighborhood Function

Any code transformation used in this paper is parameterized by one positive integer. Example 2.2 (Page 4) describes some of these parameters. This assumption—transformations defined by one positive integer—forces a total ordering over different instances of a code transformation. The *domain* of a code transformation is the set of all the values of its parameters. This paper works only with numeric domains; however, a domain does not need to be contiguous, as Example 3.1 shows.

Example 3.1. The list below shows different compiler optimizations and the ordering between their instances. The example takes liberties for the sake of the illustration; *i.e.*, vector lengths are architecture-dependent, and not every size might be available:

Peeling, with parameter `peel` $= n, n \in [0, 1, 2, \ldots,$ `maxp`$]$ being the number of iterations to peel. The set of values $\{0, 1, \ldots,$ `maxp`$\}$ is the domain of the transformation.

Unrolling, with parameter `roll` $= n, n \in [0, 1, 2, \ldots,$ `maxr`$]$ being the unrolling factor.

Thread blocking, (in graphics-processing units) with parameter `thread` $= n, n \in [0, 1, 2, \ldots,$ `maxt`$]$ being the number of GPU threads per block.

Tiling, with parameter `tile` $= n, n \in [N/20, N/15, N/12, N/10, N/6, N/4, N/3, N/2]$, where N is the size of the iteration space. We assume that N is a multiple of 60. Notice that this example chooses perfect divisors of the iteration space, but different ranges are possible.

Vectorization, with parameter `vect` $= n, n \in [0, 2, 4, 8, 16]$ being the vector length.

In any of the above examples, we say that an instance of a transformation is less than another instance based on the ordering between their parameters. For example, if we consider vectorization (`vect`), then `vect` $= m <$ `vect` $= n$ if, and only if, $m < n$.

The order between parameters leads to a notion of neighborhood, formalized as follows:

Definition 3.2 (Neighborhood). Consider two transformation vectors of an n-dimensional space: $v_1 = \langle \mathcal{P}_1 : p_1, \ldots, \mathcal{P}_{i-1} : p_{i-1}, \mathcal{P}_i : x, \mathcal{P}_{i+1} : p_{i+1}, \ldots, \mathcal{P}_n : p_n \rangle$; and $v_2 = \langle \mathcal{P}_1 : p_1, \ldots, \mathcal{P}_{i-1} : p_{i-1}, \mathcal{P}_i : y, \mathcal{P}_{i+1} : p_{i+1}, \ldots, \mathcal{P}_n : p_n \rangle$, such that each p is a transformation parameter within range $\mathcal{P}$. Vectors v_1 and v_2 are neighbors along dimension i, $1 \leq i \leq n$, if they differ only on dimension i and if:

(1) $x < y$ (without loss of generality);
(2) $\forall z \in \mathcal{P}_i, z \neq x$, if $z < y$, then $z < x$.
(3) $\forall z \in \mathcal{P}_i, z \neq y$, if $z > x$, then $z > y$.

If two vectors are neighbors along one dimension, then they are called *neighbors*. The set of every neighbor of a given transformation vector is called the *neighborhood* of that vector.

Each dimension of a transformation vector v is considered separately when determining the neighbors of v. There exist at most two neighbors per transformation parameter; therefore, the number of neighbors of v grows linearly with the number of its dimensions. This observation is essential for scalability: if the neighborhood of a vector considered variations in two or more dimensions, then the size of the neighborhood would be exponential on the number of dimensions.

Example 3.3. Consider the following transformation vector: $v = \langle \mathtt{tile}_A : 0, \mathtt{tile}_B : 8, \mathtt{tile}_C : 16, \mathtt{roll}_A : 4, \mathtt{roll}_B : 0, \mathtt{roll}_C : 0 \rangle$, which applies onto the canonical kernel in Figure 1 (c). If we let $\mathtt{roll}_A = [2, 4, 8, 16, 24]$, then v has two neighbors along dimension $\mathtt{roll}_A$. These neighbors are $v_1 = \langle \ldots, \mathtt{roll}_A : 2, \ldots \rangle$; and $v_2 = \langle \ldots, \mathtt{roll}_A : 8, \ldots \rangle$.

3.2 Convexity

The goal of this paper is to find implementations for abstract kernels that minimize their running times. To this purpose, we define the running time function RT as follows:

Definition 3.4 (The Running Time Function). Given: (i) a canonical Kernel $\mathcal{K}$; (ii) a transformation vector v with parameters $\mathcal{P}$; (iii) a computer architecture A; and (iv) input data I for the canonical kernel; we define the running time function $RT_{A,I,\mathcal{K}}(v) : \mathcal{P} \mapsto \mathbb{R}$ as a function that maps the implementation of $\mathcal{K}$ optimized by v to the time it takes to process input I on target A.

In practice, RT is obtained by running the kernel configurations. The notion of neighborhood makes the transformation space a *coordinate space*: it is possible to define the distance between two transformation vectors. The running time function RT divides this coordinate space into two halves: if $RT_{A,I,\mathcal{K}}(v) = t$, then we have a region formed by $t_{lower} < t$, and a region formed by $t_{higher} \geq t$. Thus, $RT_{A,I,\mathcal{K}}$ determines a *hypersurface* onto the transformation space. This hypersurface is convex if its local minima are equal to its global minimum. We say that vector v is a *minima* of $RT_{A,I,\mathcal{K}}$ if $RT_{A,I,\mathcal{K}}(v) < RT_{A,I,\mathcal{K}}(v')$, for any v' within the neighborhood of v. A global minimum is the smallest local minimum in a set. Example 3.5 illustrates these concepts.

Example 3.5. Figure 5 shows hypersurfaces produced by two running time functions. Each function is parameterized by a different architecture and different inputs (the dimensions of the iteration space). The functions are parameterized by the same canonical kernel—the configuration in Figure 1(b). Two parameters form the transformation space: $\texttt{tile}_A \in [0, 20, 40, \ldots, 120]$, and $\texttt{tile}_B \in [0, 20, 40, \ldots, 120]$. The figure shows, for each running time function, its optimal configuration. The figure also shows at least one local minimum that differs from the optimum. These hypersurfaces are not convex because they contain local minima that are not globally optimal.

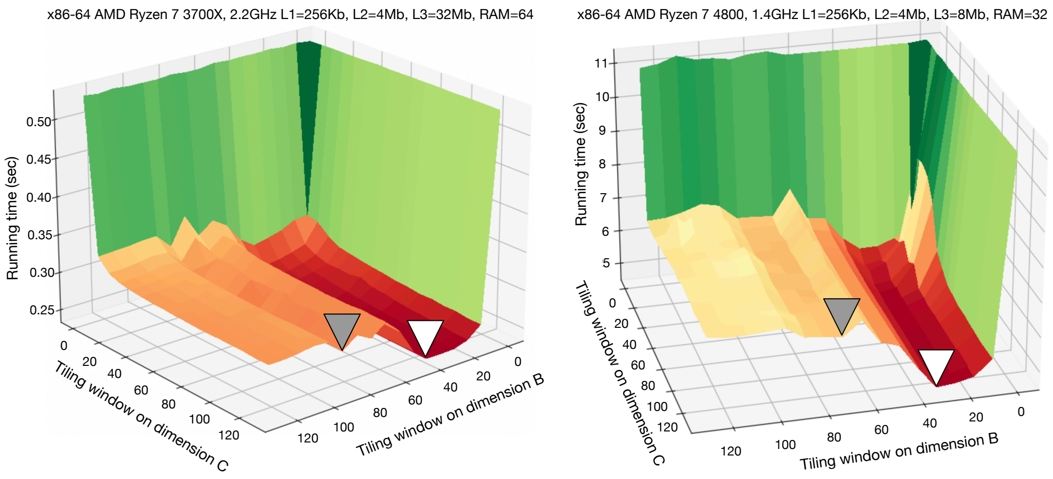


Fig. 5. Performance variation for the kernel in Figure 1 (b), considering the following parameters of the iteration space: (Left) $A = 1000, B = 800, C = 700$; and (Right) $A = 3500, B = 1400, C = 2400$. White pins show optimal tiling configurations, and gray pins show points of local minima that are not globally optimal.

The hypersurfaces in Figure 5 are not convex. However, they have the following property: the origin and the global optimum belong to the same convex region. This property remains true, at least for the two settings in Figure 5, if we add more dimensions to the transformation space considered in Example 3.5, such as an extra tiling window, or unrolling of the innermost loop, or interchange of any pair of loops. Definition 3.6 states this observation as our working hypothesis.

Definition 3.6 (The Droplet Expectation). Let v_{opt} be the optimum kernel of the running time function $RT_{A,I,\mathcal{K}}$. We expect v_{opt} and $\mathcal{K}$, the unoptimized kernel, to belong into a a convex contiguous subset of the hypersurface determined by $RT_{A,I,\mathcal{K}}$. Hence, we expect the existence of a chain of neighbor vectors $v_0, v_1, v_2, \ldots, v_{n-1}, v_n$, where $\mathcal{K} = v_0$ and $v_n = v_{opt}$, with the following properties:

Contiguous chain: $v_i \in neighborhood(v_{i-1}), 0 < i \leq n$; and
Descending chain: $RT(v_i) < RT(v_{i-1}), 0 < i \leq n$.

A continuous hypersurface can be partitioned into *convex regions*: maximal convex sets formed by the transitive closure of the neighborhood function. Whenever the Droplet Expectation is confirmed, the origin and global optimum belong to the same convex region. Thus, there exists a "*downhill path*" from origin to optimum that can be found by Coordinate Descent (assuming an idealized running time function without statistical variations). If the Droplet Expectation fails, then the origin and global optimal belong into distinct convex regions. Figure 6 illustrates these ideas.

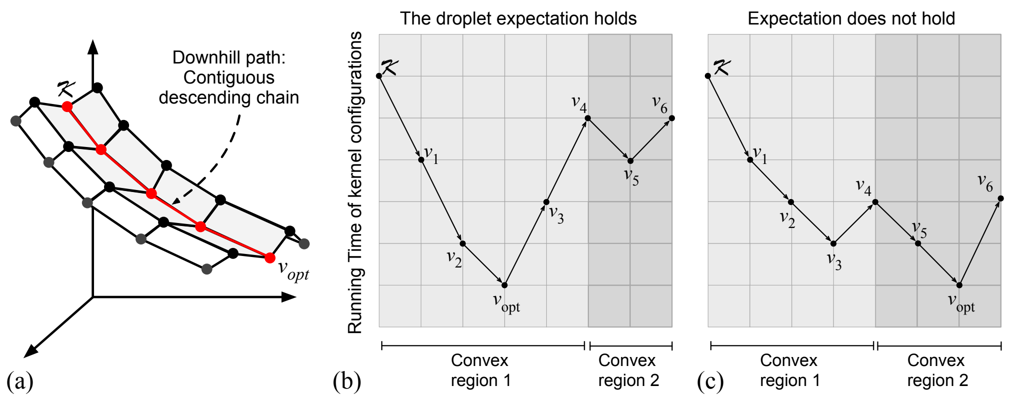


Fig. 6. (a) Downhill path from origin to global optimum on a three-dimensional hypersurface. (b) Expectation holds: origin and global optimum belong to the same convex region. (c) Expectation fails: origin and global optimum belong to different convex regions. Configuration v_3 is a *local minimum*, but not a global optimum.

Intuition. The hypothesis stated in Definition 3.6 is not a certainty: it is possible to disprove it with analytical models involving discontinuous functions, as Section 4.4 shows. However, Section 4 demonstrates that the hypothesis holds on a variety of architectures and models. Indeed, the effect of many compiler optimizations can be described by polynomial equations involving only positive coefficients ranging on positive domains. The second derivative of such functions, if they exist, will be always positive. This condition is sufficient to ensure convexity [Bertsekas 2009, Pro.1.1.10]. In the words of Renganarayana and Rajopadhye [2008], "*The use of polynomial functions with this property leads to convex optimization problems which can be solved for real solutions in polynomial time*". Renganarayana and Rajopadhye call this property *Positivity*. Notice that the ordering of the various optimization levels along each dimension of the search space is primarily responsible for making the droplet expectation hold. For instance, going back to Example 3.5, the expansion of the tiling window might be beneficial for performance until the number of elements in this window overgrows the size of the L1 cache. After this point, any further increase will cause cache misses.

3.3 Stop Criterion

The existence of the convex path expected in Definition 3.6 does not ensure that such a path can be discovered via a coordinate descent search procedure. Running time has a stochastic nature: This nature implies that the actual evaluation of $RT_{A,I,\mathcal{K}}$ on a kernel produced by a transformation vector v is prone to variations. Thus, coordinate descent might reach suboptimal configurations that are apparently optimal due to measurement fluctuations on $RT_{A,I,\mathcal{K}}(v)$.

To increase the reliability of coordinate descent, multiple evaluations of each point of the transformation space are in order. However, each further evaluation contributes to increasing the total time necessary to solve scheduling. The implementation of Droplet Search that we analyze in Section 4 performs three evaluations of each point visited in the process of solving scheduling. Three evaluations are the default sampling procedure adopted by AutoTVM. We use Student's Test (following Levine [1969]'s implementation) with significance level $\alpha = 0.05$ to compare the two populations of three samples each[4]. In other words, the search stops if we reach a transformation vector v with no neighbor that yields a faster kernel configuration with a confidence level of 95%.

3.4 Synchronous Parallelism and Speculation

The algorithm in Figure 4 keeps a current *candidate* transformation vector, which is initialized with the origin of the optimization space in Line 02 and is updated in Line 13 whenever a faster kernel configuration is found. To find a faster configuration, every point in the neighborhood of the current candidate is considered. The evaluation of these points happens in parallel, as Lines 09 and 10 of Figure 4 shows. However, we have observed that it is often difficult to fill up every available thread with unvisited kernel configurations. Example 3.7 explains how this difficulty emerges.

Example 3.7. Figure 7 shows how parallel evaluation of kernel configurations happens. Points in the neighborhood of the current candidate are evaluated in synchronous batches. Figure 7 represents these points as gray boxes. Usually, there will be a non-empty intersection of configurations between the neighborhood of the current candidate and the next candidate. Intersecting points will be stored in the `visited` set seen in Figure 4. In this example, intersecting points are marked with gray numbers in Figure 7 (b) and (c). Overlaps reduce parallelism: only five points are evaluated concurrently; thus, resources will be underutilized in processors with more than five cores.

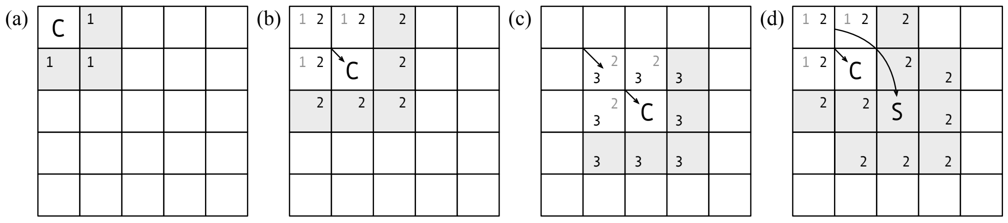


Fig. 7. (a-c) Progress of Droplet Search without speculation. Each number shows the order in which points are evaluated. Gray boxes denote points currently evaluated. Gray numbers show points in the neighborhood of the current candidate that was already evaluated. (d) Differential speculation: a subset of the extended neighborhood of the current candidate is evaluated to maximize thread occupancy.

Differential Speculation. We resort to *speculation* to maximize thread occupancy. Speculation, in the context of this paper, is the evaluation of kernel configurations in the *extended neighborhood* of the current candidate. The extended neighborhood is formed by the neighbors of neighbors. The algorithm in Figure 4 determines these extra points—the *Speculative Set*–in Line 07. This set is built via differential speculation: the history of previous coordinates of the best candidates determines a speculative set as follows: Let v_{n-1} be the candidate at iteration $n - 1$ of coordinate descent, and let v_n be the candidate at iteration n. By Definition 3.2, v_{n-1} and v_n differ in only one dimension, *e.g.*: $v_{n-1} = \langle \ldots, \mathcal{P} : x_{n-1}, \ldots \rangle$ and $v_n = \langle \ldots, \mathcal{P} : x_n, \ldots \rangle$. We choose a new transformation vector $v_s = \langle \ldots, \mathcal{P} : x_n + s, \ldots \rangle$, where $x_n + s$ is the value that follows x_n within the parameter $\mathcal{P}$. By the nature of Coordinate Descent, it is likely that v_s is not in the visited set. Nevertheless, if the centroid of the speculative set is in the `visited` set, then Line 07 of Figure 4 still ensures that

[4]Previous work has observed that speedups due to compiler optimizations may not follow a normal distribution [Álvares et al. 2021]. Student's Test is parametric; hence, not recommended for non-Gaussian distributions. However, non-parametric tests also have shortcomings. In particular, they tend to require more samples. As an example, the minimum recommended number of samples for Wilcoxon's [Wilcoxon 1992] non-parametric test would be five, under a confidence level of 95%.

multiple evaluations will not happen. Given v_s, we let the speculative set be its neighborhood. Line 07 of Figure 4 adds this set to the batch of configurations waiting to be evaluated.

Example 3.8. Figure 7 (d) provides some idea of how speculation works. We let S be the centroid of a speculative set. This kernel configuration, S, is chosen by extending the overall direction of the coordinate descent path speculatively: an action represented by the longer arrow in Figure 7 (d). In this example, speculation increases the number of active threads from five to ten.

Even though we restrict speculation to the extended neighborhood of the current candidate, this technique can still cause the line search of coordinate descent to leave a convex region. However, Section 4.5 shows that such event is not happening in practice. Had we adopted larger speculation steps, *i.e.*, outside the extended neighborhood of the current candidate, then the risk of leaving the convex region would be higher.

4 EVALUATION

This section compares Droplet Search with similar techniques employed in the implementation of Apache TVM. To this effect, this section investigates the following research questions:

RQ1: Can we demonstrate the Droplet Expectation in different architectures, at least for transformation vectors involving a small number of dimensions?
RQ2: How effective is Droplet Search on end-to-end models compared to other search techniques on different computer architectures?
RQ3: What is the average number of samples that Droplet Search takes to converge to optimal results, compared to other search techniques?
RQ4: Does the Droplet Search converge to a global optimum when applied to an industrial-quality analytical cost model?
RQ5: How does the number of threads used in Droplet Search influence the convergence time of the algorithm and the quality of the model that it finds, with and without speculation?
RQ6: How do kernels tuned via AutoTVM and Ansor compare to hand-written implementations of kernels in TensorFlow?
RQ7: How does the behavior of Droplet Search vary, in terms of quality and speed, on each individual kernel that constitutes a machine-learning model?
RQ8: What is the impact of the confidence level on the convergence rate of Droplet Search in terms of search speed and quality of the final model?

Hardware. This section evaluates scheduling approaches on six computer architectures, which Figure 8 enumerates. This mix contains two general-purpose desktop architectures (AMD and Intel); two embedded system-on-chips (ARM) and two graphics processing units (NVIDIA).

				Memory			
Device	**Arch**	**ISA**	**Clock Max (GHz)**	**Ram (GB)**	**Cache L1 (KiB)**	**Cache L2 (MiB)**	**Cache L3 (KiB)**
AMD R7-3700X	x86-64	x86	4.4	64	256	4	32
Intel i7-3770	x86-64	x86	3.9	16	32	2.56	8
Cortex-A7	ARMv7-32	arm	2.0	4	32	2	-
Cortex-A72	ARMv8-A	arm	1.5	4	80	4	-
RTX 3080	Ampere	ptx	1.7	12	-	-	-
GTX 1650	Turing	ptx	1.6	4	-	-	-

Fig. 8. The architectures evaluated in this paper.

Software. This section uses `Apache TVM v0.11.1`, released on March of 2023. We implemented Droplet Search as part of AutoTVM. AutoTVM also provides four other search techniques: grid, random, genetic (GA) and simulated annealing (XGB). XGB is described by Chen et al. [2018]. This technique uses a cost model to guide simulated annealing. The constants that define this cost model are learned during a training phase. We also compare our implementation of Droplet Search with AutoScheduler (Ansor) [Zheng et al. 2020], also available in `Apache TVM v0.11.1`.

Benchmarks. This section evaluates kernel scheduling on five convolutional neural networks and on one encoder stack of transformers (BERT). Figure 9 enumerates the models. The schedulers in AutoTVM, including Droplet Search, use the same optimization parameters. We have not chosen these parameters: they are pre-defined—per model—in the distribution of AutoTVM. Optimization parameters differ depending on the computer architecture. In the CPUs, these parameters refer to two optimizations: *tiling* and *unrolling*. In the GPUs, they concern two more: *thread blocking*: the ability to partition CUDA threads into blocks, and *shared memory tiling*: the ability to bring data from global to shared memory. All the parameters are *divisors* of boundaries of the iteration space. For instance, the size of a tile window must be a divisor of the size of the iteration space along the tiled dimension. This fact removes discontinuities from the search space (code that the compiler inserts to handle boundary conditions), as Section 4.4 will explain.

	Auto-TVM									**Auto-Schedule**					
Arch	**x86**			**ARM**			**Cuda**			**x86**		**ARM**		**Cuda**	
Network	**S**	**P**	**K**	**S**	**P**	**K**	**S**	**P**	**K**	**P**	**K**	**P**	**K**	**P**	**K**
Resnet-18	10K	5	12	0.5M	12	28	0.3T	9	16	27	24	43	24	39	24
vgg-16	11.3K	4	9	0.9M	12	27	2.3T	9	18	26	18	43	18	37	18
mobilenet	16.7K	5	18	0.6M	13	38	0.2T	8	19	26	22	40	22	37	22
mxnet	21.4K	5	20	-	-	-	0.4T	9	24	26	25	-	-	39	25
inception_v3	76.1K	5	43	1.0M	12	92	0.5T	9	50	26	56	41	56	37	56
bert	12.5K	4	7	-	-	-	2.7T	5	6	30	9	-	-	32	9

Fig. 9. The machine-learning models evaluated in this paper. "S" is the number of possible configurations formed by a given number "P" of transformation parameters. The largest neighborhood explored by Droplet Search at any time contains $2 \times P + 1$ points. "K" is the number of kernels that form each model. The implementation of the models vary according to the architecture. TVM's implementation of MXNet does not run on the ARM boards.

Ansor uses more optimizations than AutoTVM. Whereas AutoTVM is restricted to a single transformation vector (in TVM's parlance, a *kernel template*), Ansor creates several of them. The extra templates effectively give Ansor access to inter-kernel optimizations, such as loop fusion and fission: it can merge operators (*i.e.*, kernels) in the computational graph, for instance. Figure 9 shows the number of parameters in the largest template that Ansor explores for each network. The optimizers used in AutoTVM—Droplet Search included—are restricted to intra-kernel optimizations. Nevertheless, we hope to demonstrate that even accessing a smaller pool of optimizations, Droplet Search can be competitive with Ansor, which resorts to more extensive code transformations.

4.1 RQ1 – The Droplet Expectation

Definition 3.6 specifies a behavior likely to characterize typical implementations of linear kernels. This expectation is not a guarantee; thus, while we anticipate to find a path from origin to optimum along which performance improves gradually, such a path might not exist. In this case, Droplet Search will be stuck on a local minimum. Nevertheless, this section provides some empirical evidence that the Droplet Expectation holds. Section 4.2 provides further evidence: the best kernel configurations found by coordinate descent are similar to the best configurations found via more extensive techniques, even when speculation is enabled.

Methodology. We have analyzed the behavior of two kernels: matrix multiplication and convolution on six architectures. In this experiment, we use two-dimensional spaces for the sake of visualization. On CPUs (ARM or AMD), the transformation space is formed by the tiling of the two innermost loops of each kernel. Tile sizes, in both dimensions, are $0, 8, 16, 24, \ldots, 128$; hence, we have a 17×17 transformation space. On GPUs, the transformation space is formed by the number of threads in two-dimensional thread blocks. The possible number of threads are $1, 2, 4, \ldots, 32$, also forming a 17×17 transformation space. Convolution uses a $1{,}024 \times 1{,}024$ matrix with a 3×3 filter. Multiplication uses the following matrix sizes: $1{,}000 \times 700$, 700×800, and $1{,}000 \times 800$.

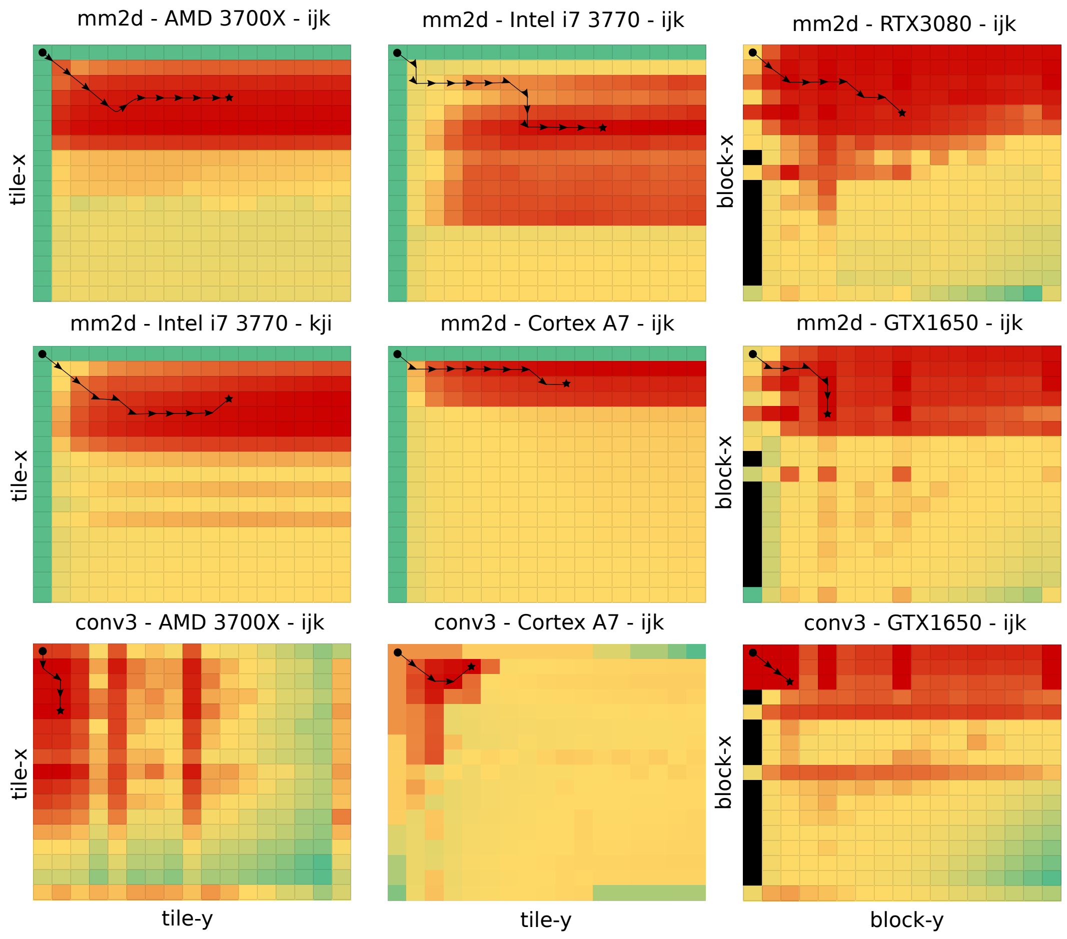


Fig. 10. Visual representation of nine different 17×17 transformation spaces, showing the path traversed by coordinate descent from origin to global optimum. Matrix multiplication is mm2d, and convolution is conv3. Performance improves from green to red. Black cells denote invalid configurations. We show, on top of each grid, the nesting order of loops, where the canonical configuration starts with the nested sequence `ijk`.

Discussion. The Droplet Expectation holds in every one of these 2×6 scenarios. Figure 10 shows representations for nine of these spaces, highlighting the path taken by coordinate descent. Some configurations involving thread blocking on the GPU are not valid; thus, they are not evaluated. In every scenario, the optimal kernel configuration is close to the origin, as the paths in Figure 10 emphasize. The Droplet Expectation holds if we increase the number of dimensions in the transformation space, or the number of kernels in the model. Section 4.2 discusses this experiment.

4.2 RQ2 – End-to-end Effectiveness

The goal of any kernel scheduling technique is to find efficient implementations of computational graphs involving kernels. This section investigates how Droplet Search fares in such task, compared with other scheduling techniques, when optimizing well-known machine learning models.

Methodology. We evaluate six end-to-end models on six architectures, with a hard limit of 10,000 evaluations per search technique. This—arbitrary—limit prevents experiments from running for too long. The Droplet Search will stop after evaluating 10,000 kernel configurations; however, it tends to stabilize earlier, as discussed in Section 4.8. The other search techniques do not have a notion of premature convergence; hence, they evaluate 10,000 configurations. Notice that Figure 9 shows that every transformation space contains more than 10,000 configurations. All the search algorithms evaluate kernel configurations in parallel. The parallelism in Droplet Search follows the ideas from Section 3.4. We use a confidence interval of 95% when comparing a candidate kernel configuration with kernels within its neighborhood. Section 4.3 discusses the impact of this last choice.

Discussion. Figure 11 compares the effectiveness of different search techniques. Considering only the search techniques in AutoTVM, Droplet Search generally yields the best kernels or ties for the best (usually with XGB). Its search time, however, is faster. On the AMD CPU, for instance, Droplet Search optimizes VGG-16 in 10% of the time the other scheduling approaches require. Ansor tends to produce faster kernels than the techniques implemented in AutoTVM, including Droplet Search. Ansor explores more optimization parameters: it has access to a large number of kernel templates, whereas AutoTVM uses only one. Nevertheless, due to excessive memory consumption, Ansor (and also AutoTVM's XGB) could not be used to schedule our largest models in the ARM boards. We have observed that Droplet Search does not perform well on the GPUs. In this case, the size of the search space (from 200M to 2.7T configurations, as seen in Figure 9) forces coordinate descent to converge due to the limit of iterations in every model, except ResNet-18. Incidentally, on ResNet-18 Droplet Search produces faster kernels than Ansor in any GPU.

4.3 RQ3 – Stop Criteria

As Section 3.3 explains, our current implementation of Droplet Search stops once it reaches a candidate kernel configuration faster than all its neighbors. Statistical significance between runtime differences is determined via Student's T-Test applied over two populations consisting of three samples each, with a confidence level of 95%. Yet, measurements might fluctuate, and premature termination is possible. If we tighten the confidence level, termination might happen too early. If we lose it, convergence can take too long, and coordinate descent might visit configurations that are not statistically significantly faster. This section investigates how Droplet Search fares once we vary the confidence level for comparing kernel configurations.

Methodology. We evaluate the five deep-learning models listed in Figure 9 on the Intel i7 using five different levels of confidence: 99%, 95%, 90%, 75%, and no test. In the latter case, we use the absolute arithmetic average of three samples to determine which kernel is faster.

Hardware	Model (ms)	Model (ms)	Search (min)	Model (ms)	Search (min)	Model (ms)	Search (min)	Model (ms)	Search (min)	Model (ms)	Search (min)	Model (ms)	Search (min)
resnet-18	**No opt**	**Droplet**		**gridsearch**		**Random**		**GA**		**XGB**		**Ansor**	
AMD3700X	24.48	15.15	25.4	15.31	191.1	15.66	194.1	15.80	194.6	15.38	198.1	15.63	217.0
Inteli7-3770	71.40	45.74	22.7	45.79	184.9	45.75	196.6	45.82	196.1	45.79	210.7	45.74	213.9
Cortex-A7	867.02	640.55	31.0	672.12	300.4	650.62	385.6	641.41	348.4	-	-	-	-
Cortex-A72	276.41	102.39	188.3	125.36	327.7	115.09	435.1	113.93	382.3	105.68	422.8	111.22	1.1k
RTX3080	0.76	0.53	130.6	1.75	199.5	0.63	195.2	0.53	219.9	0.53	349.9	0.70	189.0
GTX1650	3.93	2.35	88.7	5.72	204.6	2.41	192.2	2.13	191.5	2.03	450.7	2.53	202.7
vgg-16	**No opt**	**Droplet**		**gridsearch**		**Random**		**GA**		**XGB**		**Ansor**	
AMD3700X	243.28	127.03	16.0	128.96	189.9	128.24	199.8	127.55	195.8	129.59	204.2	122.82	212.7
Inteli7-3770	661.00	399.49	26.5	400.05	286.3	399.60	317.6	399.99	306.9	401.26	333.5	398.50	342.6
Cortex-A7	-	-	-	-	-	-	-	-	-	-	-	-	-
Cortex-A72	1.57k	0.94k	375.1	1.05k	651.5	1.51k	752.7	1.68k	769.8	1.07k	659.8	0.97k	1.2k
RTX3080	2.42	1.92	121.9	3.63	219.3	2.10	198.1	1.95	211.7	1.94	489.2	2.93	208.6
GTX1650	12.00	10.29	117.8	18.94	381.3	10.49	274.9	10.48	238.4	10.28	693.3	15.89	228.2
mobilenet	**No opt**	**Droplet**		**gridsearch**		**Random**		**GA**		**XGB**		**Ansor**	
AMD3700X	15.32	6.06	38.7	6.70	194.8	6.64	222.8	6.48	206.0	6.40	212.2	6.04	203.8
Inteli7-3770	28.56	17.10	38.9	17.50	354.8	17.18	420.0	17.40	412.3	17.42	427.7	15.05	171.0
Cortex-A7	342.58	293.68	42.1	296.22	266.6	295.05	368.4	293.21	347.0	-	-	-	-
Cortex-A72	240.45	84.76	144.7	241.74	414.1	208.81	532.9	253.56	490.3	261.23	1.3k	55.56	791.4
RTX3080	0.44	0.41	127.0	1.39	200.6	0.49	159.1	0.43	201.0	0.42	284.4	0.27	169.5
GTX1650	1.98	1.42	103.4	9.03	186.6	1.60	159.6	1.50	209.7	1.40	386.0	1.24	173.0
mxnet	**No opt**	**Droplet**		**gridsearch**		**Random**		**GA**		**XGB**		**Ansor**	
AMD3700X	58.37	33.18	46.2	34.55	193.7	33.42	208.2	33.17	204.7	33.18	205.0	34.30	214.3
Inteli7-3770	163.91	99.07	42.5	99.98	186.9	100.47	219.4	100.66	207.5	100.40	214.3	100.44	187.1
Cortex-A7	-	-	-	-	-	-	-	-	-	-	-	-	-
Cortex-A72	-	-	-	-	-	-	-	-	-	-	-	-	-
RTX3080	2.03	1.57	128.9	8.02	184.4	1.83	179.9	1.62	207.9	1.57	337.0	1.42	184.6
GTX1650	10.76	6.79	137.8	60.00	185.8	7.30	183.8	6.97	206.9	6.75	188.4	6.15	188.4
inception_v3	**No opt**	**Droplet**		**gridsearch**		**Random**		**GA**		**XGB**		**Ansor**	
AMD3700X	85.29	56.95	110.7	57.47	200.5	55.03	207.4	57.99	193.5	55.35	208.7	52.46	212.1
Inteli7-3770	247.59	153.89	110.5	194.44	196.4	157.57	217.2	164.65	194.5	155.50	222.4	150.20	188.4
Cortex-A7	3.42k	2.43k	123.8	2.73k	310.3	2.63k	403.6	2.58k	382.1	-	-	-	-
Cortex-A72	928.34	408.25	260.7	503.07	286.0	439.94	409.5	422.54	376.5	408.26	441.4	430.87	868.9
RTX3080	4.22	4.12	175.8	40.73	189.7	4.43	201.9	3.85	224.1	3.59	364.1	2.72	177.2
GTX1650	26.84	13.62	165.4	307.99	171.9	17.77	202.1	15.32	212.0	15.45	488.6	11.59	183.4
bert	**No opt**	**Droplet**		**gridsearch**		**Random**		**GA**		**XGB**		**Ansor**	
AMD3700X	205.64	119.37	31.5	120.37	156.3	121.32	205.8	121.70	164.6	121.83	177.1	85.79	60.5
Inteli7-3770	388.74	223.35	97.1	229.73	177.6	233.78	241.6	227.11	225.9	231.96	213.3	212.79	108.5
Cortex-A7	-	-	-	-	-	-	-	-	-	-	-	-	-
Cortex-A72	-	-	-	-	-	-	-	-	-	-	-	-	-
RTX3080	12.89	5.26	119.1	20.14	151.4	5.79	129.7	5.50	158.2	5.28	207.2	3.36	172.3
GTX1650	30.76	19.02	128.0	44.52	155.9	19.84	142.3	21.80	182.6	19.38	269.5	14.08	246.9

Fig. 11. Comparison of kernel scheduling techniques. "Model (ms)" is the running time of the best schedule for a given model. "Search (min)" is the time to find that configuration. Light-gray boxes denote statistically similar results (with a confidence level of 95%). Black boxes denote statistically significant best kernel times. Gray boxes with white fonts denote statistically significant best search times. Double borders denote the best schedule produced by an algorithm from AutoTVM only (*i.e.*, Ansor is not considered).

Discussion. We could observe almost no variation in the quality of the best kernel configuration depending on the confidence level. This result seems to indicate, at least for the five models running on the Intel i7 CPU, that there exists a number of "acceptable best" kernels with very similar

Intel i7-3770	Droplet									
	p = 0.01		p = 0.05		p = 0.10		p = 0.25		no p-value	
Benchmark	Time (ms)	Tuner (min)	Time (ms)	Tuner (min)	Time (ms)	Tuner (min)	Time (ms)	Tuner (min)	Time (ms)	Tuner (min)
resnet-18	45.74	42.1	45.75	42.3	45.90	42.8	45.96	43.1	45.93	44.3
vgg-16	401.39	47.6	401.37	48.4	401.18	49.1	401.67	49.7	401.68	49.7
mobilenet	16.80	70.1	17.10	70.2	17.20	72.1	17.25	74.4	17.35	77.6
mxnet	99.07	92.9	99.07	93.8	100.72	95.6	101.13	96.9	100.59	97.6
inception_v3	153.45	172.0	153.89	173.8	153.52	176.6	153.58	176.9	154.31	181.3

Fig. 12. Effect of confidence level (the stop criterion of Section 3.3) on the quality of the best kernel configuration and on the running time of Droplet Search.

dynamic behavior. However, the search time increases—albeit slightly—once we move from high confidence levels towards low confidence levels. This growth in search time happens because more kernel configurations tend to be visited by the search procedure.

4.4 RQ4 – Analytical Models

Analytical cost models are systems of equations that predict the cost of a program (CPU cycles, I/O operations, cache misses, *etc.*) given a model of the hardware. Recently, different research groups have designed analytical cost models to estimate the performance of machine-learning kernels [Olivry et al. 2021, 2020; Sumitani et al. 2023; Tollenaere et al. 2023; Zhang et al. 2021]. This section investigates if the droplet expectation also holds in some of these models.

Methodology. This section evaluates cost models taken from three different sources. The first models were proposed by Olivry et al. [2021]. They estimate a lower bound for the amount of data movement between slow and fast memories. The second model is part of the Xtensa Neural Network Compiler (XNNC), from Cadence Tensilica, and was made available to us through the Cadence Academic Network[5]. This model estimates the number of execution cycles that a program compiled by XNNC takes to execute on Cadence DSPs. Finally, we analyze the eight analytical models listed in Table II of Renganarayana and Rajopadhye [2008]'s work.

4.4.1 *Olivry et al.'s Cost Models.*

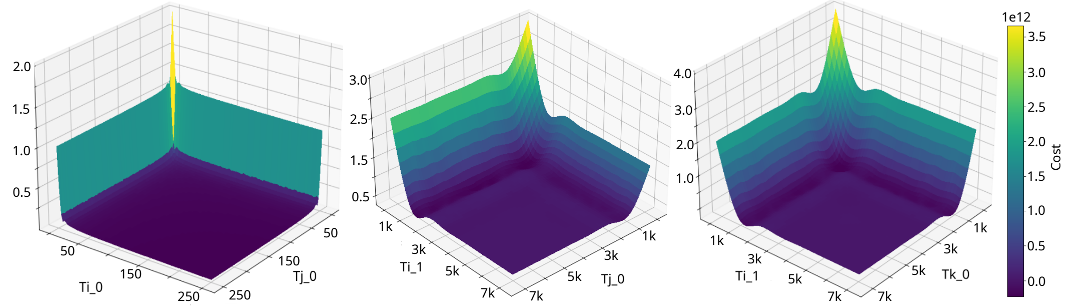
We evaluate the Olivry et al.'s model on the tiled version of matrix-matrix multiplication in the authors' original work (Listing 1 [Olivry et al. 2021]). This program is the default example in Olivry et al.'s online tool[6]. We evaluate it in a system with two caches, with 64KB and 256KB—the dimensions of the first two cache levels used in the Intel i7.

Discussion. Figure 13 shows hypersurfaces produced by the cost models generated via Olivry et al.'s online tool. Each figure relates the size of a bidimensional tiling window with the I/O cost in terms of memory transfers. The lower the cost, the faster the program is expected to run. Figure 13 (a) assumes one level of cache (with 64KB). The other two figures assume two levels (64K and 256KB). Figures 13 (b) and (c) are similar to the surfaces seen in Figure 5, which explore the same program, albeit on an actual machine. In the three figures, the droplet expectation (Definition 3.6) holds. This result is not a coincidence: Olivry et al.'s cost model involve only positive quantities (domain and coefficients); thus, form convex surfaces.

4.4.2 *Tensilica's Cost Models.*

We evaluate the Tensilica model on an implementation of the tiled ReLU kernel on two digital signal processors, called P1 and P6. We chose these two processors because they are used as tests in the Tensilica tool. The DSPs do not have a cache; however, they

[5]https://www.cadence.com/ko_KR/home/company/cadence-academic-network/university-program.html
[6]Available at https://iocomplexity.corse.inria.fr/ioub on June 2023.



(a) Single cache L1.
Cost = Ni*Nj*Nk*(1/Tj_0 + 1/Ti_0 + 2/Nk)

(b) L1 and L2, varying Ti_1 and Tj_0.
Cost = Ni*Nj*Nk*(1/Tj_1 + 1/Ti_1 + 2/Nk) + Ni*Nj*Nk*(2/Tk_0 + 1/Tj_0 + 1/Ti_1)

(c) L1 and L2, varying Ti_1 and Tk_0.

Fig. 13. Amount of memory transfers produced by Olivry et al. [2021]'s cost model applied onto Listing 1 in Olivry et al.'s work: a tiled version of the kernel in Figure 1-b in this paper. We vary tiling dimensions `Ti_0`, `Ti_1`, and `Tk_0`. The input matrices have sides `Ni` = `Nj` = `Nk` = 10K.

contain local memory and system memory. Hence, the compiler must implement direct memory access (DMA) transfers from the system to local memory, and tiling reduces DMA operations.

Discussion. In contrast to Olivry et al.'s cost model, the equations produced by XNNC take into consideration the fact that the tiling window might not be a perfect divisor of the loop's iteration space. If tiling does not perfectly divide the iteration space, then the XNNC compiler generates epilogue code to fetch data outside the tiled loop. Consequently, the performance model contains conditionals. For instance, Figure 14 (a) was produced by a (simplified) equation like `cost` = C_1 + (`if` $W\%63$ `then` C_2 `else` 0) + $C_3 \times (H/64)$. The coefficients C_i represent costs of particular instructions; W and H are tile sizes. Due to these conditionals, the cost models are represented by discontinuous functions. In this case, the droplet expectation does not hold. However, if we restrict valid neighborhoods to only perfect divisors of the iteration space, then the Droplet Expectation holds. For instance, starting with tiling windows with size 16 or greater, coordinate descent achieves the optimal configuration in any of the surfaces seen in Figure 14 after three iterations. Notice that this restriction is unnecessary if some loss is acceptable. When applied to large deep-learning networks—under the same XNNC analytical model—Droplet Search stays very close to the global optimum, although sampling less than 1% of the space covered by an exhaustive grid search.

4.4.3 Renganarayana and Rajopadhye's Cost Models. Renganarayana and Rajopadhye [2008] show that models used to solve the "*Tile Size Selection* (TSS) Problem" are represented by equations whose coefficients and domains are all positive quantities. To support their observation, they list equations taken from eight different analytical models from previous work. These equations all represent instances of the bidimensional tile-size selection problem. They use variables that range on the following quantities:

- **C** : the size of the cache in the target computer architecture;
- **L** : the length of the cache line;
- **h** : the height of the rectangular tiling window;
- **w** : the width of the rectangular tiling window; and
- **n** : the side of an $\mathbf{n} \times \mathbf{n}$ array.

In this section, we fix **C**, **L**, and **n**, and plot the hypersurface formed by **h** and **w**, within a contiguous range of values. This approach simulates Apache TVM's grid search algorithm.

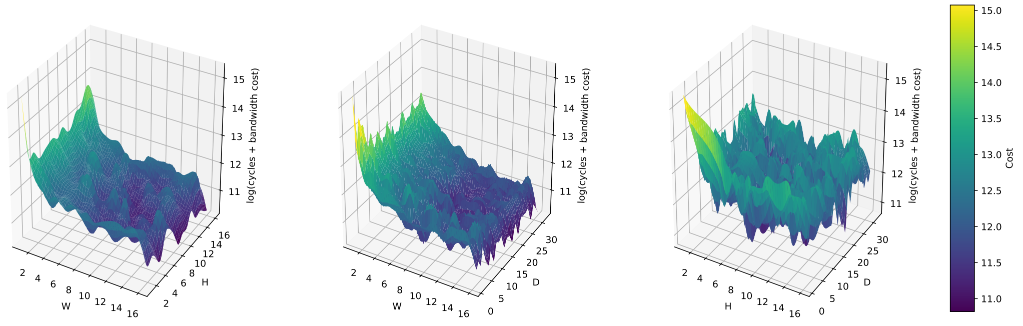


Fig. 14. Part of the tiling search space generated for a ReLU layer by the XNNC's analytical cost model for the P1 digital signal processor. The higher the cost (yellower), the slower the kernel is expected to behave, when deployed onto the actual hardware. To emphasize the non-convex regions, we show only the first 16 sizes of possible tiling windows.

Discussion. Figure 15 shows hypersurfaces for the different equations analyzed by Renganarayana and Rajopadhye. To ease visualization, we remove the **h** and **w** axes (all ranging on the interval $[1, \ldots, 100]$). The Droplet Expectation holds in all these analytical models. In fact, all of these equations use exclusively positive coefficients; hence, yielding convex surfaces.

4.5 RQ5 – Parallelism and Speculation

As explained in Section 3.4, the implementation of Droplet Search evaluated in this paper uses synchronous parallelism and speculation to speed convergence up. This section evaluates the effects of these techniques on search time and on the quality of kernel configurations.

Methodology. We evaluate Droplet Search on the five different end-to-end models on the AMD 3700X CPU. This CPU runs 16 threads (two threads per core, with eight cores). This section experiments with three versions of Droplet Search: its full implementation, an implementation that features parallelism but no speculation, and an implementation that runs on a single thread. We report p-values comparing three executions of the fastest models found with each approach.

Discussion. Figure 16 compares the different implementations of droplet search. Its single-threaded implementation takes 664 seconds to converge (sum of tuning time over five networks). The parallel version, with 16 threads, converges in 358 seconds. With speculation plus parallelism, this time goes down to 291 seconds. Parallelism is a known limitation of coordinate descent. Our implementation suffers from the shortcomings mentioned by Zheng et al. [2000]. The synchronous nature of coordinate descent limits concurrency: the best candidate is chosen after every point of a neighborhood is evaluated. Thus, progress only happens once the slowest point runs. Wang et al. [2016] have shown that it is possible to improve parallelism if more candidate points co-exist. We believe that early preemption of slow points could also speed our implementation up: once the best candidate is found in a neighborhood, the other threads can be aborted. We leave such approaches—multiple candidates and early preemption—open for future work. Speculation improves the running time of our implementation of Droplet Search by a small margin. The parallel version of Droplet Search, with speculation, is 27% faster than the non-speculative implementation (geomean over speedups). This gain comes mostly from faster convergence.

Figure 16 (Right) shows that the extended neighborhood explored via speculation has no effect on the speed of the kernel configurations found via Droplet Search. The search does not find always

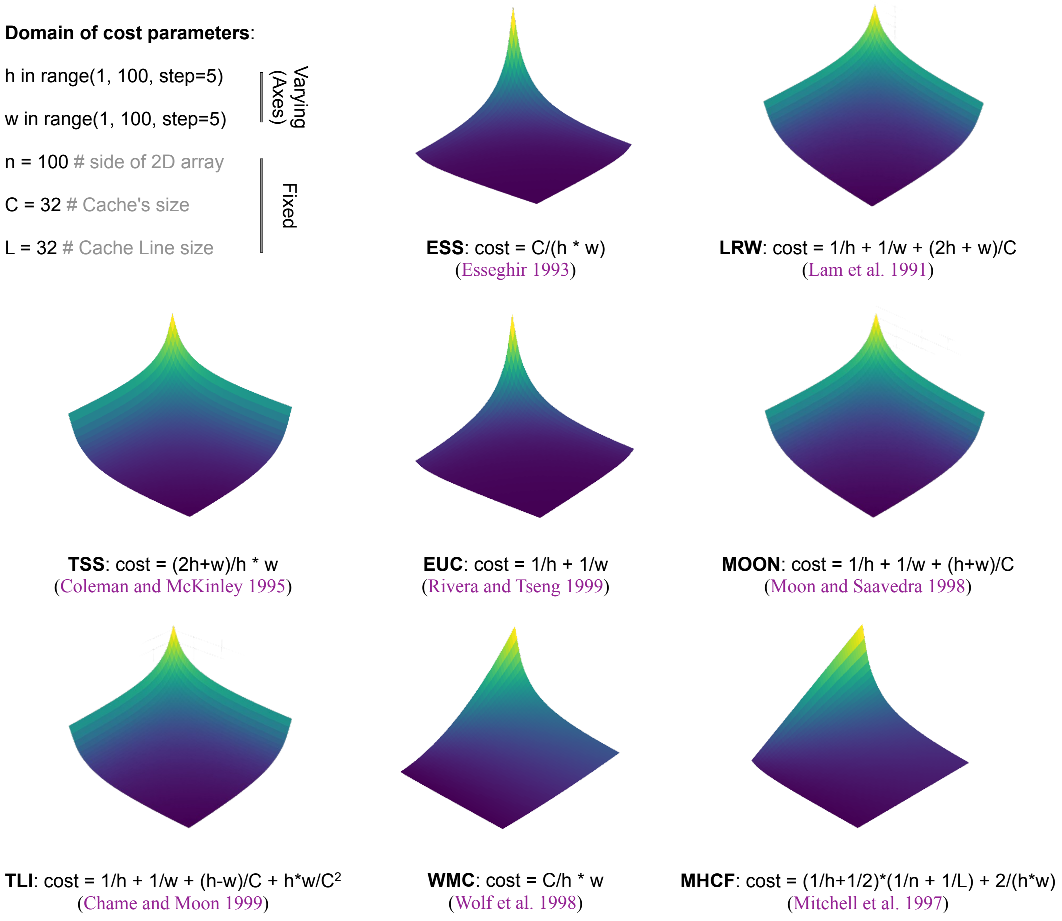


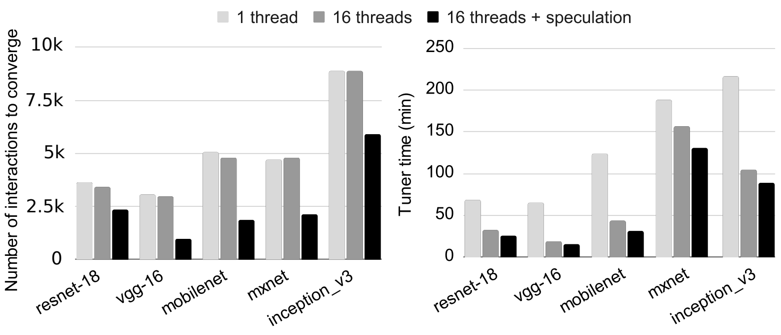
Fig. 15. Hypersurfaces formed by the models in Table II of Renganarayana and Rajopadhye [2008]'s work.

the same final configuration for every layer of every model; however, this phenomenon is due just to statistical variations in the running time of similar kernels. As the figure shows, the p-values reported by a T-Test on the speed of the different kernels is well above 0.05. Thus, the extended neighborhood is not changing the behavior of our implementation of Droplet Search.

4.6 RQ6 – Comparison with TensorFlow

The goal of this section is to bring some perspective about our results to readers that are not familiar with the Apache TVM ecosystem. To this end, we shall present a comparison between three different approaches to develop end-to-end models: AutoTVM, Ansor and TensorFlow [Abadi et al. 2016]. The latter is a Python-based library to write machine learning models. In contrast to AutoTVM or Ansor, TensorFlow does not do, by itself, any form of scheduling: the programmer must feed it with an optimized implementation of a machine learning model.

Methodology. This section compares the different kernel implementation approaches using a benchmark collection formed by six kernels: matrix multiplication, 2D convolution, depthwise separable convolution, pooling, matrix reduction and 2D ReLU. These kernels have been taken



Bench	p-value 1 thread vs speculation	p-value 16 threads vs speculation
resnet-18	0.8352	0.8853
vgg-16	0.8623	0.7325
mobilenet	0.7023	0.7835
mxnet	0.5231	0.8568
inception_v3	0.1425	0.7864

Fig. 16. (Left) Number of iterations until convergence of variations of droplet search. (Middle) Running time of different implementations of Droplet Search. (Right) P-values produced by a T-Test on populations of three executions of the best model found by each technique. The closer to 1.0 is the p-value, the more statistically similar are the two populations.

from the artifact made publicly available by Zhu et al. [2022], and are meant to run on graphics processing units. We evaluate them into our RTX 3080 GPU and on an A6000[7]. Incidentally, we have not been able to apply Zhu et al.'s tool onto these very kernels[8].

Discussion. Figure 17 shows the comparison between different kernel implementation systems. Notice that Figure 17 compares different kernel implementations. Ansor and AutoTVM (Droplet Search and XGB) receive, as input the same code: a kernel implemented in Python with libraries from Apache TVM. TensorFlow receives a different implementation: kernels also written in Python, but with libraries from the TensorFlow package. As an example, the API to invoke the ReLU (rectified linear unit) kernel is `tf.nn.relu(a_tf)` in TensorFlow, and `topi.nn.relu(a_tvm)` in Apache TVM. In short, Figure 17 is comparing two different Python libraries.

In every experiment, Apache TVM has produced faster (or equally faster) kernels than TensorFlow, be it through AutoTVM or Ansor. However, without scheduling, TensorFlow outperforms Apache TVM in two kernels: convolution and depthwise convolution. We have not observed a statistical difference between implementations of the reduction and the ReLU kernels, regardless of the library or the scheduling approach adopted to optimize them. In this regard, we observe that neither Ansor nor AutoTVM implement search for pooling, reduction, and ReLU. The implementation of these kernels, as provided by Zhu et al., does not come with a template of optimization parameters. The other kernels, in contrast, come with templates that enable thread blocking and tiling with shared memory. Loop unrolling is not enabled by these parameters. Nevertheless, Figure 17 confirms some of the results earlier observed by Zhu et al.: Kernels produced by Apache TVM tend to outperform similar kernels produced via TensorFlow. However, contrary to Zhu et al., we have observed smaller differences.

[7]The A6000 GPU is only used in this section. Access to this hardware was kindly provided by the *Discovery Lab* (https://discovery.ic.unicamp.br/).

[8]Roller, the tool implemented by Zhu et al. relies on RT Cores to speedup the execution of kernels. This tool uses functions that are exclusive of Cuda v10.2 and TensorFlow v1.15.2. However, the RTX 3080—the only GPU that we have with RT cores—is only compatible with Cuda v12.0 and TensorFlow v2.13.0. We could not downgrade the version of Cuda. Furthermore, a direct change of APIs, to upgrade the versions of Cuda and TensorFlow in Roller was not enough to let us reuse the tool: the updated version of the tool compiles successfully; however, it does not produce kernels. We faced similar issues when trying to reuse another artifact that targets RT cores: Heron [Bi et al. 2023]. Heron was implemented with the Cuda v11.0 API. We have also updated it to use Cuda v12.0. The updated version compiles and produces kernels; however, these kernels crash during execution.

Microbenchs	Model (ms)	Model (ms)	Search (min)	Model (ms)	Search (min)	Model (ms)	Search (min)	Model (ms)
RTX3080	**Original**	**Droplet**		**Autotvm (XGB)**		**Ansor**		**TensorFlow**
matmul	0.01798	0.00577	2.50	0.00577	54.10	0.01266	37.96	0.24704
conv2d	414.85434	2.09061	24.38	2.18308	31.95	1.81956	72.39	43.88901
depthwise	63.78266	0.44082	44.46	0.44013	56.74	0.46296	64.67	25.03834
pooling	0.59630	0.59545	0.04	0.59525	0.04	0.96103	71.90	60.46896
reduce	0.31750	0.31340	0.02	0.31340	0.02	0.31243	2.69	9.07090
relu	0.00017	0.00017	0.02	0.00017	0.02	0.17265	0.18	4.73173
A6000	**Original**	**Droplet**		**Autotvm (XGB)**		**Ansor**		**TensorFlow**
matmul	0.01780	0.00626	2.61	0.00632	24.60	0.01166	39.20	0.38927
conv2d	367.64008	1.93585	60.13	1.79232	69.72	1.69408	69.72	20.15370
depthwise	52.25241	0.53334	44.05	0.53213	81.28	0.70036	67.01	14.51671
pooling	0.68824	0.68812	0.04	0.68820	0.05	1.13004	81.29	10.23547
reduce	0.38104	0.38100	0.02	0.38105	0.03	0.38019	2.97	6.10128
relu	0.00020	0.00020	0.02	0.00020	0.02	0.17265	0.38	3.76800

Fig. 17. Comparison between Apache TVM and TensorFlow on the six kernels made available by Zhu et al. [2022], on two graphics processing units. This figure uses the same notation as Figures 11 and 18: dark boxes indicate the fastest models, and gray boxes indicate the fastest search times. Light gray boxes indicate results that are statistically similar. TensorFlow does not implement search.

4.7 RQ7 – Intra-Kernel Behavior

The models explored in Section 4.2 consist of multiple kernels: each layer is implemented as an independent kernel. Droplet Search and all the other search techniques available in AutoTVM are intra-kernel. Thus, kernels are optimized independently from each other. This section analyzes the effects of Droplet Search on individual kernels and compares these effects with results obtained by other scheduling techniques. By showing that Droplet Search finds similar configurations as exhaustive techniques, we provide further evidence that the droplet expectation is common.

Methodology. We analyze each kernel of ResNet-18 and VGG-16 in separate, reporting search time and kernel running time. Each kernel is extracted from its encompassing model via TVM's code generator. ResNet-18 and VGG-16 are our smallest networks, in number of kernels. We restrict this study to only two models because the individual analysis of each kernel is time consuming (meaning human time, not machine time). Nevertheless, we believe that these results could be extrapolated to the other models, which sport similar implementations.

Discussion. Figure 18 shows how the search techniques fare on each layer of ResNet-18 and VGG-16. We do not show results for Ansor, because, as Figure 9 shows, Ansor's implementation recognizes more layers on each model. Droplet Search never yields worse configurations than the other search techniques. Furthermore, its search is faster, converging with fewer samples. In this case, column "iter" in Figure 18 provides the number of kernel configurations evaluated by each search technique. In contrast to Droplet Search, the other approaches used in AutoTVM do not have a notion of convergence. The search stops once a determined number of kernel configurations is visited. However, sampling is not equally divided among the kernels: AutoTVM draws more samples for kernels that run for more time. As a example, the different search procedures of AutoTVM sample Layer Five of ResNet-18 1,024 times, as Figure 18 shows. Droplet Search, in contrast, stops after 120 iterations. The samples that Droplet Search evaluates depends more on the dimensions of the layer, such as the number of channels, width and height of the filter used. The four largest

resnet-18	Droplet			Gridsearch			Random			GA			XGB		
Layer	Time (ms)	Tuner (s)	iter	Time (ms)	Tuner (s)	iter	Time (ms)	Tuner (s)	iter	Time (ms)	Tuner (s)	iter	Time (ms)	Tuner (s)	iter
0	0.83	72.6	128	0.83	362.5	308	0.83	379.7	308	0.83	383.5	308	0.83	386.7	308
1	0.81	125.5	96	0.82	1079.2	980	0.81	1091.0	980	0.82	1098.4	980	0.82	1119.2	980
2	0.09	131.2	209	0.09	1115.8	980	0.09	1127.3	980	0.09	1128.4	980	0.09	1153.3	980
3	0.42	223.8	198	0.42	1045.9	896	0.42	1059.9	896	0.43	1052.8	896	0.42	1076.2	896
4	0.05	123.5	163	0.05	1019.4	896	0.05	1035.6	896	0.05	1033.7	896	0.05	1055.5	896
5	0.80	111.4	120	0.80	1130.1	1024	0.80	1137.3	1024	0.80	1149.0	1024	0.80	1168.1	1024
6	0.42	129.7	272	0.42	1020.4	864	0.43	1029.0	864	0.42	1026.5	864	0.42	1050.2	864
7	0.05	131.9	181	0.05	985.8	864	0.05	999.9	864	0.05	1000.4	864	0.05	1020.2	864
8	0.81	113.2	254	0.81	1086.9	972	0.81	1105.0	972	0.81	1114.9	972	0.82	1123.8	972
9	0.44	107.8	331	0.44	866.0	720	0.44	884.5	720	0.44	881.9	720	0.44	898.4	720
10	0.05	139.0	134	0.05	829.1	720	0.05	846.6	720	0.05	847.4	720	0.05	865.2	720
11	0.81	112.9	260	0.82	921.8	800	0.82	948.1	800	0.82	960.4	800	0.81	967.5	800
SUM	5.585	1522.5	2346	5.588	11463.0	10024	5.591	11643.9	10024	5.587	11677.1	10024	5.589	11884.3	10024
AVG	0.465	126.9	2346	0.466	955.3	835	0.466	970.3	835	0.466	973.1	835	0.466	990.4	835
vgg-16	Droplet			Gridsearch			Random			GA			XGB		
0	0.60	71.1	57	1.21	377.8	308	0.60	380.2	308	0.60	382.1	308	0.60	384.3	308
1	13.13	114.5	112	15.74	1095.0	1078	13.14	1116.0	1078	13.13	1110.6	1078	13.16	1148.5	1078
2	6.30	108.0	107	7.63	1163.3	1232	6.29	1212.4	1232	6.30	1201.5	1232	6.30	1242.3	1232
3	12.74	110.7	107	15.14	1380.9	1408	12.74	1439.1	1408	12.80	1413.9	1408	12.91	1473.3	1408
4	6.34	112.0	121	7.98	1347.4	1440	6.36	1457.8	1440	6.34	1392.0	1440	6.36	1478.8	1440
5	12.73	116.9	121	17.84	1611.9	1620	12.80	1731.4	1620	12.79	1682.0	1620	12.79	1762.7	1620
6	6.26	93.9	104	8.52	1419.0	1440	6.27	1533.4	1440	6.27	1482.2	1440	6.28	1565.2	1440
7	12.77	122.4	124	16.14	1742.3	1600	12.84	1830.0	1600	12.79	1791.5	1600	12.76	1874.6	1600
8	3.24	108.4	107	4.77	1258.7	1200	3.24	1286.4	1200	3.24	1291.3	1200	3.24	1325.2	1200
SUM	74.13	957.9	960	94.97	11396.2	11326	74.28	11986.8	11326	74.27	11747.1	11326	74.40	12254.9	11326
AVG	8.24	106.4	107	10.55	1266.2	1258	8.25	1331.9	1258.4	8.25	1305.2	1258	8.27	1361.7	1258

Fig. 18. The search techniques of AutoTVM applied on individual layers of deep-learning models on the AMD R7-3700X. Evaluations average three samples. Light gray boxes mark running times that are statistically similar with confidence level of 95%. Black boxes denote statistically significant running times. Gray boxes mark statistically significant search times. The sum of search times approximates the results in Figure 11. The sum of kernel times, *i.e.*, "Time (ms)", is strictly less than the running times reported in Figure 11.

layers in Figure 18 are, in this order: the 6^{th}, the 8^{th}, the 9^{th} and the 11^{th}. Incidentally, these layers account for the largest number of samples observed by Droplet Search.

4.8 RQ8 – Convergence Rate

The convergence rate of a search mechanism used to solve the kernel scheduling problem measures how fast that technique closes on its final solution. As mentioned in Section 4.7, the different techniques that AutoTVM uses to solve kernel scheduling iterate until a fixed number of configurations are evaluated. We have observed that it is often possible to stop iterations before, once a sufficiently good configuration is reached. That is the approach adopted in Droplet Search, as Section 3.3 explains. In what follows, we investigate how the quality of the final solution to scheduling improves as the number of evaluations progresses.

Methodology. We set the maximum number of iterations of AutoTVM's grid, random, genetic and XGB search to 10,000. This number, 10K, includes the evaluations of kernel configurations in neighborhoods or speculative sets that Droplet Search uses. Ansor shall use the same limit of evaluations. We then inspect the speed of the best kernel configuration that each one of these search techniques find RestNet-18 throughout the search. We show results for ResNet-18 only; however, we have evaluated the convergence rate for the other models. Results tend to be similar.

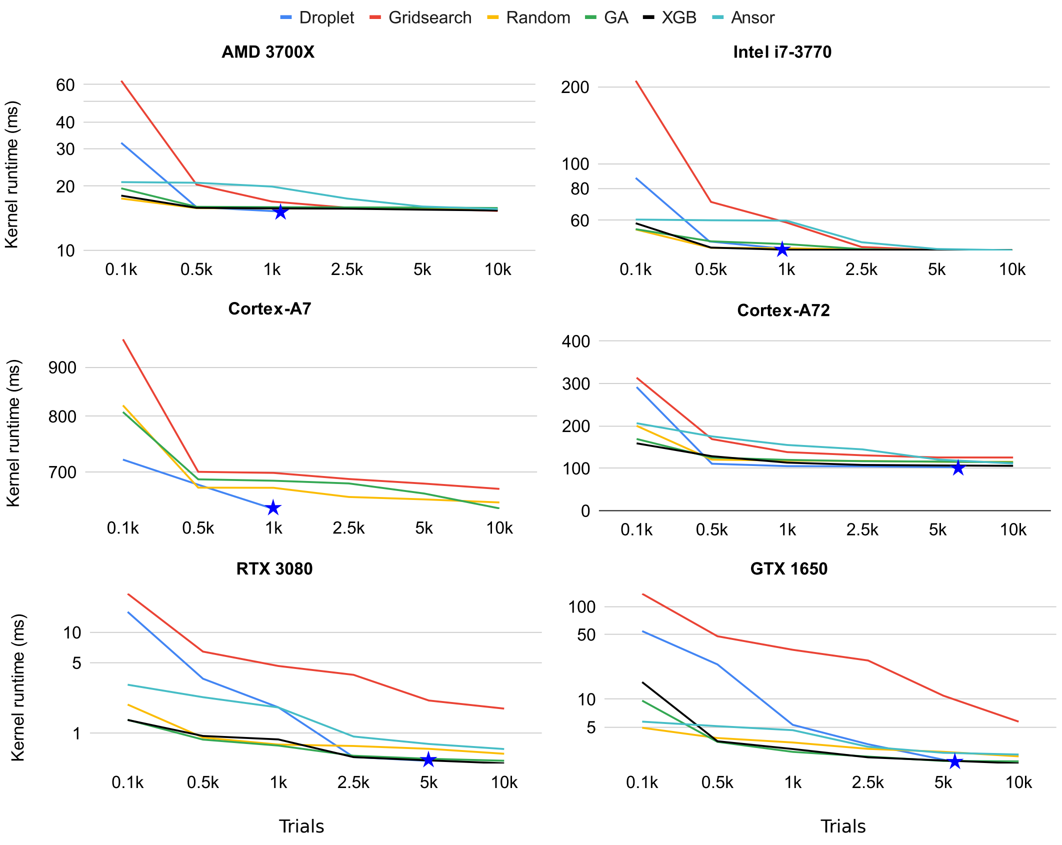


Fig. 19. Quality of the solution for kernel scheduling versus the number of evaluations (trials) used to find that solution for ResNet-18. The blue star marks the convergence of the Droplet Search.

Discussion. Figure 19 shows the results of this experiment. Droplet Search usually converges to the best solution before 10,000 evaluations. The fastest convergence was observed on the Intel CPU: coordinate descent stabilized after 1,278 configurations were evaluated. Notice that convergence happened before 2K evaluations in every CPU. The slowest convergence happened on the GTX GPU: 7,125 evaluations. Similarly, on the RTX, 6,502 configurations were evaluated. Similar results were also observed on the other four models: Droplet Search stabilizes before 10K iterations; typically before 3K iterations on the CPUs, and before 8K iterations on the GPUs.

5 RELATED WORK

This paper aims to find the best concrete implementation for a given program. We recognize two main approaches to this theme, which we shall call *autotuning* (*e.g.*, compiler autotuning) and *scheduling* (*e.g.*, program autotuning, following Tollenaere et al. [2023] taxonomy). The latter is the problem from Definition 2.7. These problems differ in two essential ways:

Training: In autotuning, the compiler is trained offline on many programs—its training set—before being applied to an unknown program. Thus, the compiler uses information acquired from general programs before optimizing a specific program. In scheduling, there is no pre-training phase: the compiler does not try to generalize the behavior of a universe of programs to predict the behavior of an individual program.

Sampling: In autotuning, the compiler typically evaluates the target program once (although there are exceptions, like in the work of Cavazos et al. [2007]), using, as a guide, the behavior learned from observations made on the training set. In scheduling, the compiler is allowed to run the target program multiple times. Information acquired from these executions will guide the search for good optimizations.

Much of the current techniques employed in autotuning originate in the work of Cavazos and his collaborators [Agakov et al. 2006; Ashouri et al. 2018, 2016; Cavazos et al. 2006; Cavazos and O'Boyle 2006; Moss et al. 1997; Simon et al. 2013; Thomson et al. 2010]. The growing availability of predictive models and benchmarks to train these models has made autotuning works common in the recent literature [Brauckmann et al. 2020; Cummins et al. 2021a,b; Da Silva et al. 2021; Silva et al. 2021]. Figure 20 compares six autotuning-based techniques with six scheduling-based approaches. Whereas autotuning typically concerns the optimization of general programs, scheduling is mainly seen in the optimization of deep-learning models composed of kernels. Autotuning has been used, for instance, to find good sequences of optimizations for `clang` [da Silva et al. 2021; Silva et al. 2021]; or to fine-tune the Hotspot JIT compiler [Cavazos and O'Boyle 2006].

Work	Training	Sampling	Space	Guide	Target	Platform
Rapidly Selecting Good Compiler Optimizations Using Performance Counters	Y	± 25	list	Performance Counters	Programs	General
Exploring the Space of Optimization Sequences for Code-Size Reduction: Insights and Tools	Y	1	list	Program features	Programs	General
Method-specific dynamic compilation using logistic regression	Y	1	vec[n], n is 4, 9, or 20	Program features	Functions	JIT
Hybrid Optimizations: Which Optimization Algorithm to Use?	Y	1	vec[1]	Program features	Function	JIT
Mapping Computations in Heterogeneous Multicore Systems with Statistical Regression on Program Inputs	Y	1	vec[8]	Input features	Functions	Parallel
Compiler-Based Graph Representations for Deep Learning Models of Code	Y	1	vec[1]	Program features	Programs	Parallel
Ansor : Generating High-Performance Tensor Programs for Deep Learning	N	[1..]	param list	Program features	Kernels	ML
Automatic Generation of High-Performance Convolution Kernels on ARM CPUs for Deep Learning	N	[1..]	param list	Grid search	Convolutions	ML
TVM: An automated end-to-end optimizing compiler for deep learning	N	[1..]	params	Program features	Kernels	ML
Autotuning Convolutions Is Easier Than You Think	N	[1..]	params	I/O complexity model	Convolutions	ML
I/O Lower Bounds for Auto-tuning of Convolutions in CNNs	N	[1..]	params	I/O complexity model	Kernels	ML
This paper	N	[1..]	params	Coordinate descent	Kernels	ML

Fig. 20. Qualitative comparison between different previous work. See Section 5 for the meaning of columns. We classify the first six works as autotuning and the last six as scheduling problems.

The transformation Space. Scheduling techniques usually fix the sequence of optimizations that form the search space, and vary their parameters [Chen et al. 2018; Tollenaere et al. 2023; Zhang et al. 2021]. However, there are scheduling approaches that accept different transformation vectors [Essadki et al. 2023; Meng et al. 2022; Phothilimthana et al. 2021; Zheng et al. 2020]. Within the TVM community, these different transformation vectors are called *templates*. For instance, a common technique—adopted in TVM's Ansor—is to assume that each interchange of loops forms a different template. Figure 20 distinguishes fixed vectors as "params" and templated vectors as "param list". Autotuning usually fix the parameters of each optimization; however, they have more freedom to compose sequences of optimizations. For instance, Cavazos et al. [2007] form sequences of 500 optimizations drawn from a universe of 121 possible compilation flags. This approach is also adopted by Silva et al. [2021]: they produce lists of up to 100 elements drawn from approximately 80 compilation flags. Figure 20 uses the notation "list" to denote this way to build the search space. In Figure 20, we denote fixed-length vectors as "vec[n]".

The Search Guide. Techniques used in autotuning or scheduling differ in how the search for good transformation sequences is performed. In this paper, we use coordinate descent. Our approach does not depend on static characteristics of the program; only on its dynamic behavior. Several search techniques use program features (static characteristics) to steer the search. These techniques are usually data-agnostic. An exception is the work of Da Silva et al. [2021], who use the runtime values of inputs to choose program configurations. Da Silva et al. capitalize on the convexity of the search space; however, in their case, tuning is guided by linear regression, not coordinate descent. Recent scheduling techniques have used analytical models to prune the search space [Kaufman et al. 2021; Mogers et al. 2022; Tollenaere et al. 2023; Zhang et al. 2021]. Mogers et al. [2022] have shown, in the context of the Lift compiler [Steuwer et al. 2016], that pruning can be very effective, as "*only 1 out of 49,000 candidates [generated by random search to optimize convolution] satisfies the constraints [hence is valid]*". Pruning is orthogonal to the search techniques that Section 5 evaluates. For instance, an interesting continuation of the ideas in this paper would be to use Tollenaere et al.'s cost model to remove from the active neighborhood kernel configurations that are unlikely to improve on the best candidate seen by coordinate descent.

6 CONCLUSION

This paper has introduced a new kernel scheduling technique—droplet search—and has demonstrated its effectiveness by optimizing the code of six different deep learning models on six different hardware architectures. Droplet Search relies on the following observation: the optimization space formed by code transformation parameters usually determines a convex region that includes the origin of this space (no optimization) and the best kernel configuration that this space contains. Experiments show that Droplet Search tends to find kernels as efficient as any of the other search techniques available in AutoTVM; however, it does so faster. Droplet Search also compares well with TVM's Ansor, despite using a much smaller pool of optimizations.

Droplet Search is currently available in Apache TVM. This implementation still offers room for improvements. In particular, its convergence rate on large search spaces can be slow. Furthermore, this implementation could benefit more from parallelism, because it keeps only one best candidate at any time. We conjecture that coordinate descent could be modified to use multiple line searches; hence, offering more opportunities for parallelization. Finally, the current implementation of Droplet Search explores the parameters of only four TVM optimizations: tiling, unrolling, thread blocking and shared memory tiling. Adding more optimizations to this list would be a welcome improvement to that implementation. All these ideas are directions that we hope to see explored in the future.

ACKNOWLEDGMENT

This project was sponsored by Cadence Design Systems, and the authors express their gratitude to Eric Stotzer for facilitating Cadence's financial support. Additionally, the authors acknowledge the support of CNPq (grants 314645/2020-9 and 406377/2018-9), FAPEMIG (grant PPM-00333-18), FAPESP (grant 2013/08293-7), and CAPES (Edital PrInt and grant 88887.668980/2022-00). Finally, the authors extend their appreciation to the ACM TACO reviewers for their valuable suggestions, which significantly contributed to the enhancement of this work.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In OSDI (Savannah, GA, USA). USENIX Association, New York, USA, 265–283.

F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. 2006. Using Machine Learning to Focus Iterative Optimization. In CGO. IEEE Computer Society, Washington, DC, USA, 295–305. https://doi.org/10.1109/CGO.2006.37

Andrei Rimsa Álvares, José Nelson Amaral, and Fernando Magno Quintão Pereira. 2021. Instruction visibility in SPEC CPU2017. J. Comput. Lang. 66 (2021), 101062. https://doi.org/10.1016/j.cola.2021.101062

Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning Using Machine Learning. Comput. Surv. 51, 5 (2018), 96:1–96:42. https://doi.org/10.1145/3197978

Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. 2016. COBAYN: Compiler Autotuning Framework Using Bayesian Networks. TACO 13, 2 (2016), 21:1–21:25. https://doi.org/10.1145/2928270

D. Bertsekas. 2009. Convex Optimization Theory. Athena Scientific, Nashua, NH, USA. https://books.google.com.br/books?id=0H1iQwAACAAJ

Jun Bi, Qi Guo, Xiaqing Li, Yongwei Zhao, Yuanbo Wen, Yuxuan Guo, Enshuai Zhou, Xing Hu, Zidong Du, Ling Li, Huaping Chen, and Tianshi Chen. 2023. Heron: Automatically Constrained High-Performance Library Generation for Deep Learning Accelerators. In ASPLOS (Vancouver, BC, Canada). Association for Computing Machinery, New York, NY, USA, 314–328. https://doi.org/10.1145/3582016.3582061

Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-Based Graph Representations for Deep Learning Models of Code. In CC. Association for Computing Machinery, New York, NY, USA, 201–211. https://doi.org/10.1145/3377555.3377894

John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. 2007. Rapidly Selecting Good Compiler Optimizations Using Performance Counters. In CGO. IEEE Computer Society, USA, 185–197. https://doi.org/10.1109/CGO.2007.32

John Cavazos, J. Eliot B. Moss, and Michael F. P. O'Boyle. 2006. Hybrid Optimizations: Which Optimization Algorithm to Use?. In Proceedings of the 15th International Conference on Compiler Construction (Vienna, Austria) (CC'06). Springer-Verlag, Berlin, Heidelberg, 124–138. https://doi.org/10.1007/11688839_12

John Cavazos and Michael F. P. O'Boyle. 2006. Method-Specific Dynamic Compilation Using Logistic Regression. In OOPSLA (Portland, Oregon, USA). Association for Computing Machinery, New York, NY, USA, 229–240. https://doi.org/10.1145/1167473.1167492

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. CoRR abs/1512.01274 (2015), 6 pages. arXiv:1512.01274 http://arxiv.org/abs/1512.01274

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In OSDI (Carlsbad, CA, USA) (OSDI'18). USENIX Association, USA, 579–594.

Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. 2021a. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In ICML, Vol. 139. PMLR, Baltimore, Maryland, USA, 2244–2253.

Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. 2021b. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. CoRR abs/2109.08267 (2021), 12 pages. arXiv:2109.08267

Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quintão Pereira. 2021. AnghaBench: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction. In CGO. IEEE, Los Alamitos, CA, USA, 378–390. https://doi.org/10.1109/CGO51591.2021.9370322

Junio Cezar Ribeiro Da Silva, Lorena Leão, Vinicius Petrucci, Abdoulaye Gamatié, and Fernando Magno Quintão Pereira. 2021. Mapping Computations in Heterogeneous Multicore Systems with Statistical Regression on Program Inputs. ACM Trans. Embed. Comput. Syst. 20, 6, Article 112 (oct 2021), 35 pages. https://doi.org/10.1145/3478288

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In NAACL-HLT, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, New York, US, 4171–4186. https://doi.org/10.18653/v1/n19-1423

Mohamed Essadki, Bertrand Michel, Bruno Maugars, Oleksandr Zinenko, Nicolas Vasilache, and Albert Cohen. 2023. Code Generation for In-Place Stencils. In CGO. Association for Computing Machinery, New York, NY, USA, 2–13.

https://doi.org/10.1145/3579990.3580006

Paul Feautrier. 1991. Dataflow analysis of array and scalar references. Int. J. Parallel Program. 20, 1 (1991), 23–53. https://doi.org/10.1007/BF01407931

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. CoRR abs/1512.03385 (2015), 12 pages. arXiv:1512.03385 http://arxiv.org/abs/1512.03385

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. CoRR abs/1704.04861 (2017), 9 pages. arXiv:1704.04861 http://arxiv.org/abs/1704.04861

Charles Jin, Phitchaya Mangpo Phothilimthana, and Sudip Roy. 2022. Neural Architecture Search Using Property Guided Synthesis. Proc. ACM Program. Lang. 6, OOPSLA2, Article 166 (oct 2022), 30 pages. https://doi.org/10.1145/3563329

Samuel J. Kaufman, Phitchaya Mangpo Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. 2021. A Learned Performance Model for Tensor Processing Units. In MLSys, Alex Smola, Alex Dimakis, and Ion Stoica (Eds.). mlsys.org, Indio, CA, USA, 15 pages.

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. 1988. Optimization by Simulated Annealing. MIT Press, Cambridge, MA, USA, 551–567.

Mikhail Lebedev and Pavel Belecky. 2021. A Survey of Open-source Tools for FPGA-based Inference of Artificial Neural Networks. In IVMEM. IEEE, New York, US, 50–56. https://doi.org/10.1109/IVMEM53963.2021.00015

David A. Levine. 1969. Algorithm 344: Student's t-Distribution [S14]. Commun. ACM 12, 1 (jan 1969), 37–38. https://doi.org/10.1145/362835.362841

Jintao Meng, Chen Zhuang, Peng Chen, Mohamed Wahib, Bertil Schmidt, Xiao Wang, Haidong Lan, Dou Wu, Minwen Deng, Yanjie Wei, and Shengzhong Feng. 2022. Automatic Generation of High-Performance Convolution Kernels on ARM CPUs for Deep Learning. IEEE Trans. Parallel Distributed Syst. 33, 11 (2022), 2885–2899. https://doi.org/10.1109/TPDS.2022.3146257

Naums Mogers, Lu Li, Valentin Radu, and Christophe Dubach. 2022. Mapping Parallelism in a Functional IR through Constraint Satisfaction: A Case Study on Convolution for Mobile GPUs. In CC (Seoul, South Korea). Association for Computing Machinery, New York, NY, USA, 218–230. https://doi.org/10.1145/3497776.3517777

Eliot Moss, Paul Utgoff, John Cavazos, Doina Precup, Darko Stefanovic, Carla Brodley, and David Scheeff. 1997. Learning to Schedule Straight-Line Code. In NIPS. MIT Press, Cambridge, MA, USA, 929–935. https://doi.org/10.5555/3008904.3009034

Auguste Olivry, Guillaume Iooss, Nicolas Tollenaere, Atanas Rountev, P. Sadayappan, and Fabrice Rastello. 2021. IOOpt: Automatic Derivation of I/O Complexity Bounds for Affine Programs. In PLDI (Virtual, Canada). Association for Computing Machinery, New York, NY, USA, 1187–1202. https://doi.org/10.1145/3453483.3454103

Auguste Olivry, Julien Langou, Louis-Noël Pouchet, P. Sadayappan, and Fabrice Rastello. 2020. Automated Derivation of Parametric Data Movement Lower Bounds for Affine Programs. In PLDI (London, UK). Association for Computing Machinery, New York, NY, USA, 808–822. https://doi.org/10.1145/3385412.3385989

Phitchaya Mangpo Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivasa Murthy, Yanqi Zhou, Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Rezsa Farahani, Yu Emma Wang, Berkin Ilbeyi, Blake A. Hechtman, Bjarke Roune, Shen Wang, Yuanzhong Xu, and Samuel J. Kaufman. 2021. A Flexible Approach to Autotuning Multi-Pass Machine Learning Compilers. In PACT, Jaejin Lee and Albert Cohen (Eds.). IEEE, New York, 1–16. https://doi.org/10.1109/PACT52795.2021.00008

Lakshminarayanan Renganarayana and Sanjay Rajopadhye. 2008. Positivity, Posynomials and Tile Size Selection. In SC. IEEE Press, Austin, Texas, Article 55, 12 pages.

Peter Richtárik and Martin Takác. 2012. Parallel Coordinate Descent Methods for Big Data Optimization. CoRR abs/1212.0873 (2012), 43 pages. arXiv:1212.0873 http://arxiv.org/abs/1212.0873

Abhronil Sengupta, Yuting Ye, Robert Wang, Chiao Liu, and Kaushik Roy. 2018. Going Deeper in Spiking Neural Networks: VGG and Residual Architectures. CoRR abs/1802.02627 (2018), 16 pages. arXiv:1802.02627 http://arxiv.org/abs/1802.02627

Anderson Faustino da Silva, Bernardo N. B. de Lima, and Fernando Magno Quintão Pereira. 2021. Exploring the Space of Optimization Sequences for Code-Size Reduction: Insights and Tools. In Compiler Construction. Association for Computing Machinery, New York, NY, USA, 47–58. https://doi.org/10.1145/3446804.3446849

Douglas Simon, John Cavazos, Christian Wimmer, and Sameer Kulkarni. 2013. Automatic Construction of Inlining Heuristics Using Machine Learning. In CGO. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/CGO.2013.6495004

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2016. Matrix Multiplication beyond Auto-Tuning: Rewrite-Based GPU Code Generation. In CASES (Pittsburgh, Pennsylvania). Association for Computing Machinery, New York, NY, USA, Article 15, 10 pages. https://doi.org/10.1145/2968455.2968521

Rafael Sumitani, Lucas Silva, Frederico Campos, and Fernando Pereira. 2023. A Class of Programs That Admit Exact Complexity Analysis via Newton?S Polynomial Interpolation. In SBLP (Campo Grande, MS, Brazil). Association for Computing Machinery, New York, NY, USA, 50–55. https://doi.org/10.1145/3624309.3624311

John Thomson, Michael O'Boyle, Grigori Fursin, and Björn Franke. 2010. Reducing Training Time in a One-Shot Machine Learning-Based Compiler. In Languages and Compilers for Parallel Computing, Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 399–407.

Nicolas Tollenaere, Guillaume Iooss, Stéphane Pouget, Hugo Brunie, Christophe Guillon, Albert Cohen, P. Sadayappan, and Fabrice Rastello. 2023. Autotuning Convolutions Is Easier Than You Think. ACM Trans. Archit. Code Optim. 20, 2, Article 20 (mar 2023), 24 pages. https://doi.org/10.1145/3570641

Xiao Wang, Amit Sabne, Sherman Kisner, Anand Raghunathan, Charles Bouman, and Samuel Midkiff. 2016. High Performance Model Based Image Reconstruction. In Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Barcelona, Spain) (PPoPP '16). Association for Computing Machinery, New York, NY, USA, Article 2, 12 pages. https://doi.org/10.1145/2851141.2851163

Frank Wilcoxon. 1992. Individual Comparisons by Ranking Methods. Springer New York, New York, NY, 196–202. https://doi.org/10.1007/978-1-4612-4380-9_16

Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. In Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (Toronto, Ontario, Canada) (PLDI '91). Association for Computing Machinery, New York, NY, USA, 30–44. https://doi.org/10.1145/113445.113449

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.

Xiaoyang Zhang, Junmin Xiao, and Guangming Tan. 2021. I/O Lower Bounds for Auto-Tuning of Convolutions in CNNs. In PPoPP. Association for Computing Machinery, New York, NY, USA, 247–261. https://doi.org/10.1145/3437801.3441609

Jun Zheng, Suhail S. Saquib, Ken D. Sauer, and Charles A. Bouman. 2000. Parallelizable Bayesian tomography algorithms with rapid, guaranteed convergence. IEEE Trans. Image Process. 9, 10 (2000), 1745–1759. https://doi.org/10.1109/83.869186

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.

Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. 2022. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. In OSDI, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, New York, USA, 233–248. https://www.usenix.org/conference/osdi22/presentation/zhu