Autotuning Convolutions is Easier Than You Think

NICOLAS TOLLENAERE, GUILLAUME IOOSS, INRIA, France
STÉPHANE POUGET, University of California Los-Angeles, USA
HUGO BRUNIE, CHRISTOPHE GUILLON, INRIA, France
ALBERT COHEN, Google, France
P. SADAYAPPAN, University of Utah, USA
FABRICE RASTELLO, INRIA, France

A wide range of scientific and machine learning applications depend on highly optimized implementations of tensor computations. Exploiting the full capacity of a given processor architecture remains a challenging task, due to the complexity of the microarchitectural features that come into play when seeking near-peak performance. Among the state-of-the-art techniques for loop transformations for performance optimization, AutoScheduler [Zheng et al. 2020a] tends to outperform other systems. It often yields higher performance as compared to vendor libraries, but takes a large number of runs to converge, while also involving a complex training environment.

In this paper, we define a structured configuration space that enables much faster convergence to high-performance code versions, using only random sampling of candidates. We focus on two-dimensional convolutions on CPUs. Compared to state-of-the-art libraries, our structured search space enables higher performance for typical tensor shapes encountered in convolution stages in deep learning pipelines. Compared to auto-tuning code generators like AutoScheduler, it prunes the search space while increasing the density of efficient implementations. We analyze the impact on convergence speed and performance distribution, on two Intel x86 processors and one ARM AArch64 processor. We match or outperform the performance of the state-of-the-art oneDNN library and TVM's AutoScheduler, while reducing the autotuning effort by at least an order of magnitude.

CCS Concepts: • **Software and its engineering → Source code generation**; **Dynamic compilers**.

Additional Key Words and Phrases: Code generation, Optimisation space, Microkernel, Convolution

1 INTRODUCTION

Tensor computations are at the core of many applications in scientific computing, signal processing, data analytics and machine learning. Their optimized implementation is therefore of considerable interest. While vendor libraries have originally been the result of extensive manual efforts [Van Zee and van de Geijn 2015], today's leading approaches involve domain-specific code generators. Such code generators are typically controlled by an expert, or by an autotuning algorithm often referred to as an autoscheduler. Focusing on *convolution operations*, the range of available options is the following:

- **Vendor libraries** like oneDNN [Intel 2018] and cuDNN [NVIDIA 2018] have been manually optimized by expert HPC and software engineers. They used to dominate the HPC landscape. But with the growing

Authors' addresses: Nicolas Tollenaere, Guillaume Iooss, INRIA, Grenoble, France, nicolas.tollenaere/guillaume.iooss@inria.fr; Stéphane Pouget, University of California Los-Angeles, Los Angeles, USA, pouget@cs.ucla.edu; Hugo Brunie, Christophe Guillon, INRIA, Grenoble, France, hugo.bruinie/christophe.guillon@inria.fr; Albert Cohen, Google, Paris, France, albertcohen@google.com; P. Sadayappan, University of Utah, Salt Lake City, USA, psaday@gmail.com; Fabrice Rastello, INRIA, Grenoble, France, fabrice.rastello@inria.fr.

1544-3566/2022/11-ART
https://doi.org/10.1145/3570641

diversity of operations and architectures, manual efforts do not scale. In particular, while modern libraries use JIT optimization, they cannot fully adapt to every given tensor shape of a CNN layer in a DNN pipeline.

- **Polyhedral compilers** such as Diesel [Elango et al. 2018], Polly [Grosser et al. 2012], Pluto [Bondhugula et al. 2008], PPCG [Verdoolaege et al. 2013], Tensor Comprehensions, [Vasilache et al. 2018], Tiramisu [Baghdadi et al. 2019] automatically generate multi-level tiled code for affine loop nests. However, a significant limitation is that none of them can directly optimize across tile sizes, which is critical for efficient CNN implementations.
- **Autotuning** can be performed by systems like AutoTVM [Chen et al. 2018b] or AutoScheduler [Zheng et al. 2020a] both part of the TVM domain-specific compiler [Chen et al. 2018a]. A search process guided by a dynamically constructed machine learning model [Chen et al. 2018b] iterates through tiled loop configurations, where code is generated, compiled and executed on the target platform. AutoTVM has been demonstrated to outperform polyhedral compilers [Chen et al. 2018b], and AutoScheduler to outperform AutoTVM [Zheng et al. 2020a].
- **Analytical modeling and optimization**. Recent research has shown that a comprehensive characterization and optimization across all possible tiled loop configurations for CNNs is feasible [Li et al. 2021]. The approach is semi-automatic: manual reasoning to build analytical models of data movement, in conjunction with the automated resolution of nonlinear optimization problems to optimize tile sizes.

Among existing solutions, AutoScheduler [Zheng et al. 2020a] has demonstrated higher performance of optimized codes over both automatic and semi-automatic tools, as well as vendor libraries. AutoScheduler defines a space in which it samples candidate implementations. It runs a batch of candidates on the target platform, and trains and refines a regression cost model using the performance measurements from executed candidates. The cost model is used to select samples for the next batch, focusing on the candidates with the best predicted performance by the cost model. This approach is very effective in generating high-performance code, but the training environment in the autotuning loop is rather cumbersome. The convergence rate is also slow, at least a thousand runs, which takes several hours for each optimized stage in a DNN pipeline.

When analyzing the programs sampled across multiple AutoScheduler sessions, we observed that the cost model tends to rediscover some classical principles of efficient code generation, such as outer product microkernels. A microkernel is an unrolled and vectorized portion of computation, whose data footprint fits inside the innermost level of the memory architecture, i.e., CPU registers. The implementations in vendor libraries (such as oneDNN) relies heavily on a very small set of microkernels, written in assembly code or with vector intrinsics [Li et al. 2021; Van Zee and van de Geijn 2015]. This is effective when the unroll factor divides the problem size, but lacks flexibility overall. Indeed, the best implementations found by AutoScheduler often leverage unconventional microkernels, unrolled along up to 5 dimensions, with a variety of unroll factors.

These observations raise the question of how much of the search acceleration benefits of AutoScheduler's ML modeling could be achieved by the use of expert knowledge embedded into the optimization search space.

Contributions. This paper introduces a structured space capturing such expert knowledge. Its structure derives from the offline (and automatic) identification of a collection of efficient microkernels, embedded into an original, hierarchical tiling scheme. We show that the plain random sampling of candidates in this space allows for much faster convergence than AutoScheduler with comparable performance.

We focus on two-dimensional convolutions on CPU. While accelerators are the dominant platform for training, latency-bound inference tasks are still typically served by CPUs.

The construction of our structured configuration space is as follows: We start with an *offline preselection* of high-performing, automatically-generated microkernels. This step in problem-size independent and performed only once for each target microarchitecture. Then, *online search* operates within a hierarchy of strictly *divisible tiles*. The structure of the online search derives from this divisibility constraint, and from the requirement that every tile decomposes into the microkernels identified offline. Since some problem sizes are not divisible by any tile size corresponding to an efficient microkernel, we allow for the sequential *combination of two microkernels with the appropriate multiplicity* matching the tensor sizes for a specific convolution stage to be optimized. Compared to state-of-the-art libraries, our structured search space enables higher performance for typical tensor shapes encountered in convolution stages in deep learning pipelines. Compared to automatic generators like AutoScheduler, it prunes the search space while increasing the density of efficient implementations.

While our approach applies to a wider class of tensor computations, our experiments focus on CNNs with 2D convolutions. We evaluate the impact of the structure of the search space on the performance distribution.

We describe our implementation and evaluate it on 20+ CNN layers from 2 ML inference models (ResNet-18 and Yolo-9000). We match or outperform the performance of the state-of-the-art oneDNN library [Intel 2018] and TVM's AutoScheduler [Chen et al. 2018a; Zheng et al. 2020a], while reducing the autotuning effort by at least an order of magnitude.

Outline. The rest of the paper is organized as follows: Sec. 2 provides a high-level overview of our approach. Sec. 3 introduces the principles of the structured search space. Sec. 4 presents the overall autotuning strategy. Sec. 5 reports experimental results and compares with state-of-the-art frameworks and libraries. Sec. 6 revisits the principles of our autotuning strategy in the form of an ablation study. Sec. 7 discusses related work before the conclusion in Sec. 8.

2 OVERVIEW OF THE APPROACH

Let us illustrate the overall approach by observing the structure of the generated code and how it conditions the structure of the search space itself. Please refer to Section 3.1 for a formal definition of the concepts used in this section.

Structure of the generated code. Fig. 1 and 2 illustrate the two-level code generation strategy for convolutions and the nature of microkernels in this context.

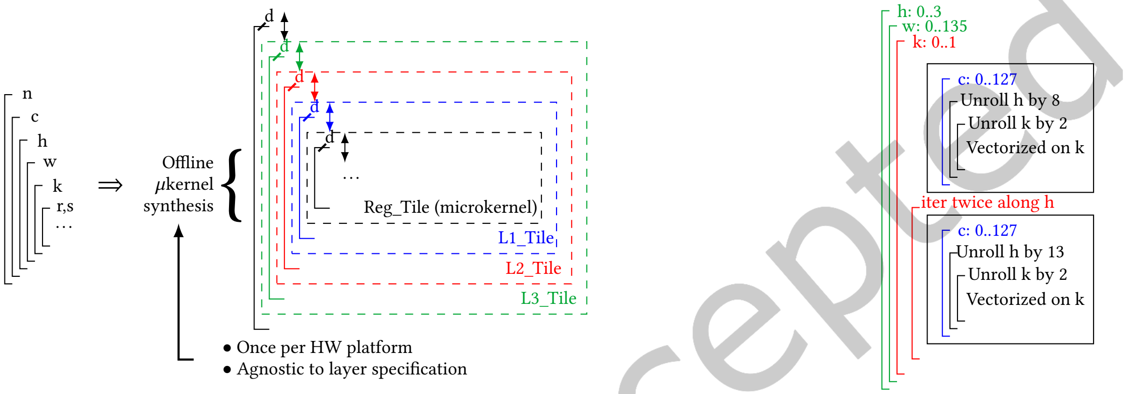
As shown in Fig. 2(a), the generated code can be divided into two parts. The innermost loops, which are register-resident, correspond to the microkernel. Apart from the reduction loop on c, these loops are unrolled and vectorized, in order to use fully the capability of the CPU's computational units. Then, the microkernel is repeated across the whole iteration space. The enclosing loops are the result of *multi-level tiling*.

A 2D convolution is a 7-dimensional nested loop. Its optimized implementation requires multi-level tiling. Given a d-dimensional nested loop ($d = 7$ here), and a 5-level memory hierarchy (main-memory, L3, L2, L1 caches and registers), the total number of nested loops for tiling at all levels is $5d$ (35 here). This is illustrated in Fig. 2(a) as a set of outermost d tile-loops that step through L3-level tiles. Each L3-level tile has d tile-loops to step through a set of L2-level tiles, and so on, with the register-level tiles being marked as a microkernel. In practice, efficient tiled implementations will only have a small subset of *active* tile loops at a level, while the remaining ones are *degenerate* with a range of a single iteration and hence removed from the code. However, we cannot know *a priori* which tile-loops are active versus degenerate; figuring this out is one of the responsibilities of autotuning.

Fig. 2(b) shows the code we generate on one sample convolution for a target platform with a vector size of 16 elements. It uses two microkernels, one corresponding to a slice of the convolution iteration space with tile extents $[H : 8, W : 1, C : 1, K : 2 \times 16]$, and another with tile extents $[H : 13, W : 1, C : 1, K : 2 \times 16]$. The L1-level tile (color-coded blue) spans the full range of 128 iterations along C, which covers the full problem extent along C. An L2-level tile (color-coded red) spans a range of $8 + 2 \times 13 = 34$ along H and a range of $2 \times 32 = 64$ along K (which is the full problem extent). An L3-level tile (color-coded green) spans $4 \times 34 = 136$ along H, 136 along W. At this point the full problem extent has been covered and therefore the outermost tile loops (color-coded black in Fig. 2(a)) are degenerate. This example illustrates how combining two well-performing microkernels can

```
for (n = 0; n < N; n += 1)
 for (k = 0; k < K; k += 1)
  for (c = 0; c < C; c += 1)
   for (h = 0; h < H; h += 1)
    for (w = 0; w < W; w += 1)
     for (r = 0; r < R; r += 1)
      for (s = 0; s < S; s += 1)
        O[n, k, h, w] = K[n, k, c, r, s] * I[n, c, h + r, w + s]
```

Fig. 1. 2D Convolution (unit stride).



(a) Multi-level tiling and split two-level optimization.

(b) Yolo9000 layer 5.

Fig. 2. Code generation sketch using microkernel composition. The left side (a) shows the generated code's generic structure aligned with the memory hierarchy. The right side (b) shows a concrete example for the convolution sizes $K = 64$, $C = 128$, $H = W = 136$ and $R = S = 1$. Note that $136 = (8 + 13 \times 2) \times 4$. Loop colors match the cache level they fit into.

be used to perfectly cover the full iteration space using a small collection of pre-selected microkernels, without needing to use any low-performance code for “partial” tiles.

Structure of the search space. Based on the code structure observations above, we elaborate on general principles about our structuring of the search space.

- **Preselecting microkernels:** *High-performing candidates require high-performing microkernels.* It is essential for any multi-level tiling strategy to eventually decompose the problem into one or more unrolled microkernels with the following properties: fully vectorized arithmetic operations, fully vectorized loads and stores in innermost loops, enough Instruction Level Parallelism (ILP) to saturate vector compute units, while keeping register pressure under control to avoid spilling. We define a sufficiently broad space of possible microkernels, varying the number of dimensions considered for unrolling as well as lower and upper bounds on the unroll factor. We measure the performance of all these microkernels, and retain those approaching the peak performance of the target CPU. This process is independent of the problem size and needs to be performed only once per microarchitecture. Compared to AutoScheduler, this approach eliminates all choices about vectorization (which dimensions to be considered) and unrolling (which dimensions to unroll and how much) from the search space for a specific problem size.

- **Divisibility constraint:** *Partial tiles hurt performance and pollute the search space with sub-par candidates.* In particular, as a consequence of the previous principle, we exclude situations where sub-optimal microkernels would be necessary to handle trailing iterations when the unroll factor for a problem dimension does not divide the size of this dimension. To eliminate such situations, we only consider microkernels and tiles whose sizes divide those of enclosing tiles and the problem itself. This eliminates vasts regions of the search space, compared to AutoScheduler, where the density of high-performing candidates is extremely low.
- **Combination of microkernels:** *Enforcing divisibility is sometimes inconvenient; it can be relaxed by combining microkernels with the appropriate multiplicity.* Indeed, when a problem extent is a large prime number, it may be impossible to enforce strict divisibility and stay within cache capacity constraints. Combining two microkernels of different sizes, repeated an appropriate number of times, allows us to match any problem size or any enclosing tile size. This marginally increases the size of the search space, in cases where divisibility alone does not allow for a sufficiently broad range of tile and microkernel sizes.

We will analyze the effect of these principles on the performance distribution, and show that (i) this search space contains implementations whose performance is on par with AutoScheduler's best candidates, and (ii) the performance distribution is such that even random sampling is sufficient to find good configurations much faster than AutoScheduler.

3 SEARCH SPACE PRINCIPLES

We now present the principles underlying our search space in greater details. Section 3.1 starts with notations and background concepts. In Section 3.2, we discuss the relation between the divisibility constraint, the need to consider a large number of microkernels, and the possibility of combining two of those. Section 3.3 formalizes the representation of candidates within the search space. We introduce the notion of an *optimization configuration*, a list of specifiers, each one corresponding to a (potentially unrolled or vectorized) loop level in the generated code. This notion specializes the notion of schedule in Halide or TVM to reflect the structural principles and domain-specific nature of our search space. We finally discuss in Section 3.4 how to generate code from a configuration.

3.1 Background: Sketching the generated code

Notations and targeted tensor operations. In the rest of the paper, lowercase variables refer to problem dimensions or loop iterations (i, j, k), and uppercase variables to problem sizes or loop bounds on each corresponding dimension (resp. I, J, K). The *iteration space* is the set of integer vectors formed by the iterations of loops enclosing a given computational statement.

In the class of computations we consider, we assume that any dimension is either parallel—i and j—or a reduction—k and all dimensions are permutable (i.e., amenable to loop interchange). While associativity can be used to parallelize reductions, we do not exploit it.

We also assume that a tensor may be accessed multiple times but always with the same subscript expressions, which are *affine functions* of surrounding loop iterators. For example, tensor $\mathbf{A}$ of shape $\{i, k \mid 0 \leq i < I, 0 \leq k < K\}$ may be subscripted by $[i, k]$, corresponding to the access function $(i, j, k \mapsto i, k)$. We also assume that a loop index cannot appear twice inside an access function: for example, $E[i, i]$ is forbidden. These conditions are satisfied by all tensor contractions and convolutions, including strided variants.

Microkernel and tiling. A *microkernel* refers to an efficient region of code composed of a (large) basic block resulting from the full unrolling of innermost parallel loops, enclosed into zero or more perfectly nested reduction loops. It is generally written in assembly language or using vector intrinsics, aiming for the following objectives: (i) effective utilization of vector ALUs; (ii) effective reuse of (vector) registers across iterations through unrolling

```
for ($i_t = 0$; $i_t < I$; $i_t$ += 6)
    for ($j_t = 0$; $j_t < J$; $j_t$ += 32)
        for ($k = 0$; $k < K$; $k$ += 1)
            $\mu$kernel_gemm$_{6,32}(C, A, B, i_t, j_t, k)$
```

Fig. 3. Tiled sgemm with microkernel.

and register promotion; (iii) adequate Instruction-Level Parallelism (ILP) to hide the latency of pipelined functional units (multiply-and-add).

Tiling [Coleman and McKinley 1995; Rivera and Tseng 1999] is a loop transformation that partitions the iteration space into sets, called *tiles* and executed atomically. We only consider programs with rectangular iteration spaces, and rectangular tiles. Tiled code has additional loops compared to the original code: loops over tiles called tile-loops, and loops inside a tile called point-loops. Such a partitioning allows us to control the amount of data accessed per tile, a.k.a. footprint, to make sure it does not exceed a given cache capacity.

Fig. 3 shows a tiled matrix multiplication kernel as an illustrative example. It relies on an (inline) fully-unrolled and vectorized microkernel of size 6×32.

High-performance libraries, such as BLIS, TCCG, oneDNN, rely on the use of a single microkernel with some fixed tile sizes within the microkernel, e.g., 6 and 32 in the example of Fig. 3. When tile sizes do not divide tensor shapes, the traditional approach involves conditional execution or padding to manage partial tiles.

As we will see in rest of this section, *we consider a broader configuration space, using a collection of microkernels for use with different problem sizes*. This also allows us to relax the *divisibility constraint* that must be satisfied in order to avoid partial tiles, by combining multiple fully-optimized microkernels in sequence.

3.2 Divisibility constraint and microkernels

In this section, we demonstrate the importance of combining microkernels instead of relying on (sub-optimal) partial tiles. We consider the multiplication of very small matrices, such that the data footprint fits inside the L1 cache, and we measure performance for a continuous range of problem sizes.

If the microkernel size perfectly divides the problem sizes, we observe a peak in performance. If the microkernel size does not perfectly divide the problem size, the classical options are (i) to have a partial tile, smaller than the microkernel that completes the work along the non-divisible dimensions; or (ii) to *pad* with zeros to make all dimensions divisible, at the cost of extraneous computations and data movements. We also consider a third route: (iii) to combine two different microkernels to cover the space without partial tiles. The method to determine the best performing microkernel will be described in Section 4.1, and the selection algorithm is explained in Section 4.2.

Comparison of different microkernel strategies. Figure 4 compares the sequential performance of small matrix multiplication implementations, for problem sizes $J = K = 128$ and $8 \leq I \leq 49$, on an Intel Xeon Gold 6230R CPU (Cascade Lake-SP, with AVX512). Performance is shown as percentage of the absolute peak, corresponding to the maximal utilization of the two vectorized FMA units of the microarchitecture.

MKL [Wang et al. 2014], *Blis* [Van Zee and van de Geijn 2015] and *libxsmm* [Heinecke et al. 2016] report the performance of these libraries. Notice the peak every 8 elements of I for MKL and a peak every 12 elements for BLIS. This gives us an indication about the size of their microkernel along dimension i. Libxsmm also considers combinations of microkernels, but restricted to predefined sizes such as powers of 2 along the i dimension. Our experiment shows that this is not enough to obtain consistent performance for all problem sizes.

"*Single microkernel, partial tile*" reports the performance of the code generated by our framework, restricted to using the BLIS microkernel only, with an unrolled partial tile to complete non-divisible dimensions. We observe a

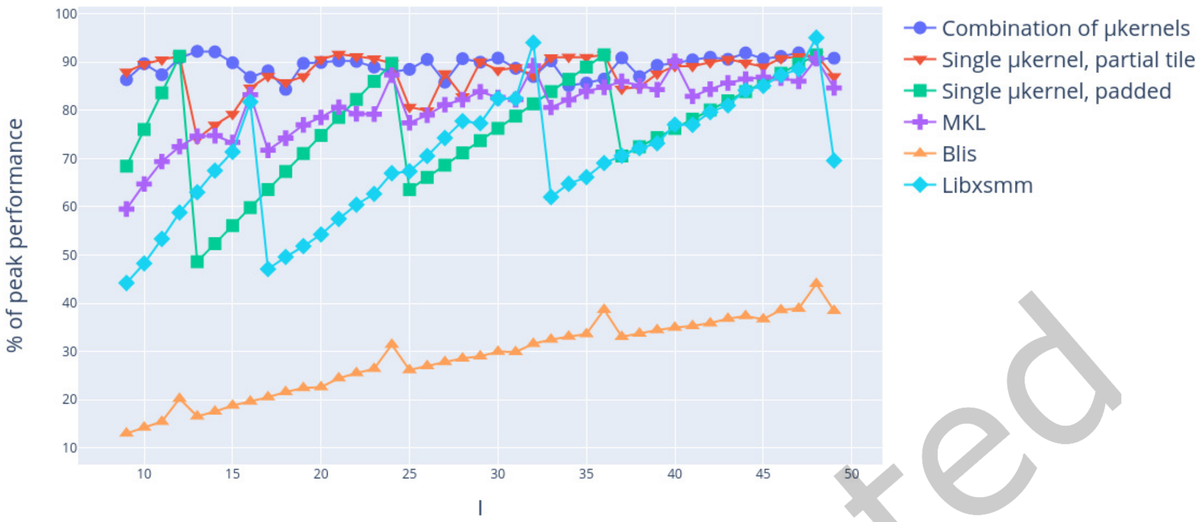


Fig. 4. Performance of small matrix multiplication kernels, for $J = K = 128$ and $8 \leq I \leq 50$.

fluctuation of periodicity 12 in its performance. As expected, for values of I with a low remainder modulo 12, the performance is worse than for a high remainder due to the low performance of the partial tile.

"*Single microkernel, padded*" is also the performance of the code generated by our framework, but using a padding strategy instead of a partial tile. We optimistically assumed that the padding overhead is free. As expected, the performance for low remainders is quite poor due to the significant overhead. However, this penalty decreases on larger sizes.

Finally, "*Combination of microkernels*" corresponds to our microkernel combination strategy, which uses two microkernels with a different size along the I dimension. We observe much more stability and high performance overall for any value of I.

This experiment shows the importance of using all the microkernels available and to combine them, to avoid loss of performance due to padding or partial tiles. This is particularly important for some convolution benchmarks, such as Yolo9000, which have small problem sizes along most dimensions, which amplifies the penalty due to a partial tile and which can have uncooperative divisors (such as $34 = 2 \times 17$ for Yolo9000-12).

In order to build such combination, we also need a larger variation of efficient microkernels, in order to cover as many problem sizes as possible while respecting the divisibility constraint.

3.3 Optimization configuration

As discussed earlier, we embed the following constraints in our configuration search space:

- We configure tile loops over microkernels from a set of pre-selected high-performing versions generated in an off-line kernel synthesis stage.
- As mentioned in Section 3.2, we forbid partial tiles, which are particularly inefficient on the innermost levels of the generated code.
- Because the divisibility constraint would be too strict for some problem sizes, we consider combinations of microkernels.

To formally define this space, we need to describe a precise selection of code generation choices. This is the role of the so-called *optimization configuration*, a specialized form of schedule in Halide or TVM, matching the

```
for ($j = 0$; $j < 128$; $j$+= 16) {
  for ($i = 0$; $i < 72$; $i$+= 6)
    for ($k = 0$; $k < n_k$; $k$+= 1)
      $\mu$kernel_gemm$_{6,16}$
  for ($i = 72$; $i < 128$; $i$+= 7)
    for ($k = 0$; $k < n_k$; $k$+= 1)
      $\mu$kernel_gemm$_{7,16}$
}
```

Fig. 5. Microkernel composition example.Both microkernels must have the same sizes except for one dimension. This dimension may be split into two loop nests, relaxing the divisibility constraint at the level of microkernels. This accommodates for the partitioning of more problem sizes while guaranteeing near-optimal performance for all iterations of the split dimension.

domain-specific structure of the code we aim to generate. A configuration is a list of *specifiers* that describe the layered structure of the generated code *from the outermost loop inwards*:

- R_d inserts the outer loop along dimension d. This loop will iterate over the outer-level tiles along d. The size of these tiles should divide the problem size D. Besides, R_d may appear at most once for a given dimension d.
- $\mathsf{T}_{\alpha,d}$ inserts a tile loop along dimension d. It iterates *exactly* α times along d. Again, α must divide the size of the iteration space along d.
- $\mathsf{U}_{\alpha,d}$ virtually inserts a tile loop with $\mathsf{T}_{\alpha,d}$ then fully unrolls it (register tile). The divisibility constraint holds.
- V_d virtually inserts a tile loop with $\mathsf{T}_{v,d}$ where v the vector length, then vectorizes it. Vectorization occurs at the innermost level only: there may be at most one $\mathsf{V}_\bullet$.
- $\lambda_{\text{seq}_d}\alpha.\,[\ell]$, where $\ell = [(r_i, a_i)]_{1 \le i < s}$ is a list of $s \ge 2$ pairs introducing a sequence of s loops of size r_i along dimension d. Each one iterates over next-level tiles, defining parameter $\alpha = a_i$ for the specifier introducing these tiles. This specifier generates non-perfectly nested tiles, composing microkernels whose sizes do not individually divide the size of a given dimension. For example, splitting a dimension y of size $Y = 34$ into two non-equal parts 22 and 12 with $\ell = [(2, 11), (1, 12)]$ fulfills the divisibility constraint (no partial tiles) while involving high-performance microkernels of size 11 and 12 along y.

Example. The naive implementation of a matrix multiplication would be represented as $[\mathsf{R}_i, \mathsf{R}_j, \mathsf{R}_k]$. A higher performance implementation, based on the BLIS [Van Zee and van de Geijn 2015] microkernel for floats (f32) on AVX2 would be

$$[\mathsf{R}_j, \mathsf{R}_k, \mathsf{R}_i, \mathsf{T}_{\frac{n_c}{16},j}, \mathsf{T}_{\frac{m_c}{6},i}, \mathsf{T}_{n_k,k}, \mathsf{U}_{6,i}, \mathsf{U}_{2,j}, \mathsf{V}_j]$$

The generated code contains a microkernel of size $(i = 6, j = 16, k = n_k)$ known to be efficient, as it requires only 15 vector registers and exposes enough ILP (12 independent multiply-add instructions issued between two accumulation steps) [Van Zee and van de Geijn 2015]. Above it, loops i and j induce a 2D tile of size (m_c, n_c). One may immediately notice that this approach assumes that I is a multiple of m_c, itself being a multiple of 6 (similar constraints apply to j and k).

State-of-the-art libraries rely on fixed-size microkernels and tuned tiles sizes, and thus introduce *partial, sub-optimal* tiles to cope with arbitrary problem sizes that do not fulfill such a divisibility constraint. Assume for example a matrix-multiplication of size $I \times J \times K = 128 \times 128 \times 64$. 128 is not divisible by 6, but $128 = 12 \times 6 + 8 \times 7$, and efficient code can be obtained using the following configuration:

$$[\mathsf{R}_j, \lambda_{\text{seq}_i}\alpha.\,[(12,6),(8,7)]\,, \mathsf{T}_{n_k,k}, \mathsf{U}_{\alpha,i}, \mathsf{U}_{2,j}, \mathsf{V}_j]$$

which leads to the loop structure shown in Fig. 5.

3.4 Code generation

We next show how we generate C code from a computation specification, problem size and the associated configuration. Generating a loop requires knowledge of the sizes of the sub-tiles, and so our code generator proceeds from innermost loop outwards. Calling a *sub-configuration* the suffix of a configuration, at a given step the already generated code (that corresponds to inner levels) is fully specified by the corresponding sub-configuration. In the following, the size of a sub-configuration refers to the size of the corresponding (parameterized) sub-iteration space. For the example from Sec. 3.1, the sub-configuration of the BLIS microkernel (including the reduction loop on k) is: $S_{\mu\texttt{kernel}} = [\mathsf{T}_{n_k,k}, \mathsf{U}_{6,i}, \mathsf{U}_{2,j}, \mathsf{V}_j]$. Its size along i, j, and k is respectively 6, 16 and n_k.

Overview. Our code generator traverses the configuration right to left in a single pass. At every level, we keep track of the following information: (i) the size of the loops that are already generated; (ii) for each dimension, the name of the last index used by a `for` loop (to handle tiling).

Before applying our code generation algorithm, we apply a preprocessing step to get rid of the λ_{seq} specifier and its parameter α. We introduce a new specifier Seq that corresponds to the sequential composition of a list of sub-configurations. In our case, the list of the λ_{seq} specifiers is always of size 2. The corresponding rewriting rule is:

$$[\ \dots,\ \lambda_{\text{seq}_d}\alpha.\,[(i_1, v_1), (i_2, v_2)]\ ,\ S\ ] \Rightarrow [\ \dots, \mathsf{Seq}([\mathsf{T}_{i_1,d}, S[\alpha/v_1]],\ [\mathsf{T}_{i_2,d}, S[\alpha/v_2]])\ ]$$

where S is the sub-configuration following the λ_{seq} specifier, and $S[\alpha/v]$ is this sub-configuration where α was substituted by the value v. We now have a tree of specifiers instead of a list of specifiers, on which we can still iterate from the leaves (innermost loops) to the root of the tree (outermost loops).

Code generation rules. Let us now survey the different specifiers and how code generation operates for each one:

- **Sequence Seq:** combine sequentially the generated code corresponding to the sub-configurations inside the Seq.
- **Vectorization $\mathbf{V}_d$:** based on the hypotheses on the code structure presented in Sec. 3.1, one may determine which operations should be vectorized by traversing the graph starting from the loads:
  - read (T, f) is vectorized if d appears in the access function f.
  - Op (x, y) is vectorized if one of its operands (x or y) is vectorized. If one of them is a scalar, it is broadcasted.
  - write (v, T, f) is vectorized if v is a vector and d appears in the access function f. These conditions must be both true or false, else this is an error.

  The C code uses Intel intrinsics to manipulate vectors.
- **Unroll $\mathbf{U}_{k,d}$:** Unroll the computation over the d dimension k times by duplicating the generated code of its sub-configuration, while updating the value of the loop index on the d dimension in each duplication.
- **Tiling $\mathbf{T}_{k,d}$ or $\mathbf{R}_d$:** Add a loop over the generated code of its sub-configuration that iterates k times, and whose value is increased by the value of the sub-configuration. In the case of R_d, one may deduce the correct number of iterations by comparing the size of the sub-configuration with the problem sizes. This changes the current loop index in use over the d dimension.

4 STRUCTURED SEARCH SPACE CONSTRUCTION AND EXPLORATION

Let us now present the overall autotuning strategy. We restrict ourselves to 2D convolutions in the following (shown in Fig. 1). Our strategy can be generalized to any program in the class of programs described in Section 3.1, provided the identification of a suitable class of microkernels, identifying a vectorization dimension (here k), a microkernel reuse dimension (here c) and a dimension along which we can compose two microkernels (here h).

4.1 Offline stage

The offline stage consists of identifying the best performing microkernels, which will effectively structure the search space (through divisibility constraints) and serve as building blocks for code generation.

Microkernel space definition. As mentioned in Section 3.1, a good microkernel must satisfy several constraints to be efficient: usage of vector units, good ILP to hide latency, and keeping vector register pressure under control.

In the context of 2D convolutions, these guidelines translate into the following constraints on the formation of a microkernel and its enclosing loop:

- The dimension k is the one being vectorized, because it contains the simplest access pattern among w, h and k. This allows to store the elements of the output tensor **O** and the parameter tensor **K** in the vector registers.
- The microkernel must have a loop along the c dimension surrounding it. This allows to reuse partially accumulated reductions in the output array promoted to vector registers. Keeping register pressure under control translates into imposing constraints on the dimensions of the microkernel.

Let us consider a microkernel whose unrolling factors are $size_x$ along the x dimension. Notice that these unrolling factors correspond to the sizes of the microkernel, except for the vectorized dimension k, where the size of the microkernel is $(vector\_size \times size_k)$. Let us count the number of vector registers it requires. The output tensor uses $size_w \times size_h \times size_k$ vector registers and its elements must stay in them to have reuse. The input tensor accesses are not vectorized, thus they need at least 1 vector register to perform a broadcast. The parameter tensor requires $size_r \times size_s \times size_c \times size_k$ vector loads, and can erase them once they are used. Thus, it is better to have specific vector registers for them, but it is not required.

From this reasoning, we consider the following collection of microkernels, considering an AVX512 architecture with 32 vector registers:

$$[\mathsf{U}_{size_s,s}, \mathsf{U}_{size_r,r}, \mathsf{U}_{size_c,c}, \mathsf{U}_{size_w,w}, \mathsf{U}_{size_h,h}, \mathsf{U}_{size_k,k}, \mathsf{V}_k]$$

with the following set of constraints:

- $16 \leq size_w \times size_h \times size_k + size_r \times size_s \times size_c \times size_k \leq 36$ (constraint on the footprint of the output and the parameter tensors)
- $14 \leq size_w \times size_h \times size_k \leq 28$ (constraint to prioritize the output tensor)
- $1 \leq size_w, size_h, size_k, size_c \leq 16$
- $(size_r, size_s) \in \{1, 3, 5, 7\}$ such that if $size_r, size_s > 1$ then $size_r = size_s$.

This makes a total of 3059 microkernels, and we remark that the AVX512 BLIS microkernel ($(k, c, w, h, r, s) = (2, 1, 12, 1, 1, 1)$) is one of them.

Microkernel evaluation. To evaluate performance, we repeat the resulting unrolled basic block many times along the c dimension ($\mathsf{T}_{512,c}$) and run the microkernel on a matching problem size. The results for a slice of the space on AVX512 are shown in Fig. 6 (on an Intel Xeon Gold 6130, frequency set to 2.1 GHz, Debian GNU/Linux, kernel version 4.19, and hardware counters monitored with PAPI v5.7.0).

Within this collection, 540 microkernels reach at least 80% of peak performance. We observe that the graph is roughly convex with some local fluctuations. We sort these microkernels into *classes*: a class is a set of microkernels above the 80% performance threshold with identical sizes except for $size_h$. Since the graph is roughly convex, the values of $size_h$ in a given class typically form an interval. For example, $\{[\mathsf{U}_{size_h,h}, \mathsf{U}_{2,k}, \mathsf{V}_k], 8 \leq size_h < 15\}$ is one of the classes of microkernels that is selected for AVX-512, as shown with the leftmost red vertical rectangular

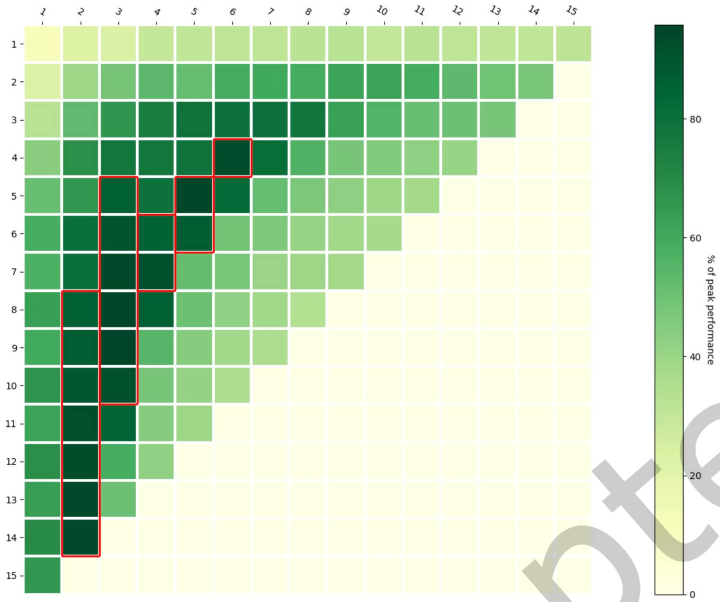


Fig. 6. Performance of microkernels in isolation for AVX512 in percentage of the machine peak, for the slice of the space where $C = H = R = S = 1$. Microkernel sizes—α along the k dimension (horizontal axis) and β along the h dimension (vertical axis)—vary between 1 and 15. Only the upper-left triangle was evaluated, because high register pressure induces spills, dramatically impacting performance on the other half. The red-bordered microkernels are the highest performing, thus are the ones selected (offline).

contour on Fig. 6. When combining microkernels to mitigate the divisibility constraint, we will pick two of those from the same class (by using a $\lambda_{\mathrm{seq}_h}\alpha.[\ell]$ specifier).

The measurement of all these microkernels takes about 50 minutes to complete on this architecture. This step is problem-size agnostic and needs to be done only once per target architecture.

4.2 Online stage

Given a problem size, we will now describe how to derive a structured search space of optimization candidates from the offline selection of microkernels. In particular, we will show how to select and combine two different microkernels to satisfy the divisibility constraint, which is a fundamental issue when the size of a problem dimension does not have many small prime divisors.

Microkernels and combination. The search space construction starts by considering all classes of microkernels and selecting those whose sizes divide the problem sizes. Then, we look for the combination of two microkernels differing only along the h dimension that allows to cover the size of the h dimension. For all pairs of microkernels in the same class, of sizes h_1 and h_2, and given a problem size H, we look for a number of repetitions a and b of these microkernels such that: $(a \times h_1 + b \times h_2)$ divides H. For example, if $H = 34$ and considering two microkernels of the same class of size $h_1 = 11$ and $h_2 = 12$, one may combine two microkernels of size 11 followed by a microkernel of size 12, for a total of 34.

If no single microkernel or combination of microkernels would be found with this process, the fallback would be to use a suboptimal microkernel, which is what we wanted to avoid by ruling out partial tiles. Fortunately, this situation never happens on the architectures and micro-architectures we considered. Indeed, the classes of microkernels identified in Section 4.1 are large enough to accommodate for any possible size through the combination of two microkernels (as long as the size of the convolution dimension of interest is greater than or equal to the smallest microkernel in the selected class). For example, for the microkernel class $\{[\mathsf{U}_{size_h,h}, \mathsf{U}_{2,k}, \mathsf{V}_k], 8 \leq size_h < 16\}$, all problem sizes H above 8 can be obtained by a linear combination of two integral elements h_1 and h_2 from the interval $[8, 15]$ (e.g. $17 = 8 + 9$).

Completing the configuration. For each single microkernel that divides the problem sizes, or microkernel combination that divides the problem sizes, we can enumerate all the possibilities of completing this microkernel selection into a full configuration:

- Set the configuration (the list of specifiers) to the one corresponding to the chosen microkernel or combination of microkernels.
- Then, one needs to complete the configuration along all dimensions for which the problem size is greater than the microkernel size. For each dimension d, consider a divisor n of the dimension's size D divided by the size of the matching microkernel dimension. We can complete the configuration by appending the corresponding specifier $\mathsf{T}_{n,d}$ to the left of the current configuration. By performing this operation recursively, and by considering all possible choices of dimension and divisor, we obtain the set of all possible configurations built on top of the selected microkernel.
- To benefit from register reuse, we impose that the first dimension above the microkernel is c.
- When considering a sequence of two microkernels at dimension d with the combination $a \times h_1 + b \times h_2$, we can insert $\lambda_{\mathrm{seq}_d} \alpha.\ [\ell]$ at any occurrence of dimension d in the configuration. Thus, we also consider all possibilities of placement of this insertion. The value of the list ℓ is $[(a, h_1), (b, h_2)]$.

Example. Consider the following class of microkernels (among others):

$$\{[\mathsf{U}_{\beta,h}, \mathsf{U}_{2,k}, \mathsf{V}_k], 8 \leq \beta \leq 15\}$$

for AVX512 and the Yolo9000-13 problem sizes $(K, C, H/W, R/S) = (512, 256, 34, 3)$.

The problem size H on dimension h is $34 = 2 \times 17$, hence there is no single microkernel from the considered class that matches one of its divisors. Next, we consider combinations of 2 microkernels from that class; $2 \times 11 + 12$ is one such combination.

We need to complete this configuration. By comparing the sizes of the combination of microkernel and the problem, there is factor of $16 = 2^4$ along the k dimension, no factor left along the h dimension, and the whole problem size left, along all other dimensions.

We have to pick c as the first dimension above the microkernel, so let us pick 256 as his factor. The configuration is now:

$$[\mathsf{T}_{256,c}, \mathsf{U}_{\beta,h}, \mathsf{U}_{2,k}, \mathsf{V}_k]$$

We continue completing the configuration by picking a dimension and a factor at every level. At some point, we need to decide at which level to place the sequential combination between the microkernel $\lambda_{\mathrm{seq}_h} \beta.\ [(2, 11), (1, 12)]$, which separates the two portions of the generated code that combines two different microkernels.

Finally, an example of full completions (among others) is:

$$[\mathsf{T}_{16,k}, \lambda_{\mathrm{seq}_h} \beta.\ [(2, 11), (1, 12)]\,, \mathsf{T}_{17,w}, \mathsf{T}_{3,s}, \mathsf{T}_{3,r}, \mathsf{T}_{2,w}, \mathsf{T}_{256,c}, \mathsf{U}_{\beta,h}, \mathsf{U}_{2,k}, \mathsf{V}_k]$$

The construction of the structured search space is summarized in Fig. 7. We will see in the next section that the resulting search space is sufficiently small and sufficiently dense with good candidates to yield excellent results with random search only.

Offline stage: microkernel exploration (Sec. 4.1).

- Build the set of microkernel candidates with unroll factors satisfying a predefined, architecture-specific set of constraints (range of unroll factors, footprint of the output and parameter tensors).
- Generate fully vectorized code for every candidate and run all of them to table their performance.
- Sort the fastest ones into classes of microkernels (with identical sizes except for the $size_h$ dimension).

Online stage: construction of the optimization search space (Sec. 4.2).
Given the exact problem sizes:

- Build the set of (combinations of) microkernels from the classes, that divide the problem sizes.
- If the previous set is empty, fallback to a sub-optimal microkernel.
- For a given microkernel/combination:
  - For each dimension:
    - Compute the number of outer loop iterations for every dimension at (coarser) microkernel granularity, by dividing each problem size by the corresponding microkernel/combination size.
    - Enumerate all the ordered factorizations of these outer loop sizes satisfying the divisibility constraint. Each integer factor in the ordered factorization for a given dimension yields one level of tiling in the generated code.
  - Select a nesting order between the tiling levels collected across all dimensions, to form the tiling strategy above the microkernel.
  - If we consider a combination of microkernels, place the $\lambda_{\mathrm{seq}_d}\alpha.\,[\ell]$ somewhere above the microkernel.

Fig. 7. Construction of the structured optimization space.

5 PERFORMANCE RESULTS

In this section, we evaluate the effectiveness of random sampling as a search algorithm. We compare the performance of the generated code with oneDNN [Intel 2018] (Intel library, V2.3), AutoScheduler [Zheng et al. 2020a] (Ansor, autotuning, in TVM, 22 July 2021) and Mopt [Li et al. 2021] (analytical modeling). We also compare its convergence rate against AutoScheduler, in terms of the number of configurations generated and run.

Setup. The experiments were carried out on three architectures: (a) an 18-core Intel Xeon Gold 5220 Cascade Lake processor (frequency set to 2.2 GHz) with 1 socket and 1 AVX512 fused multiply-add unit per core; and (b) a 32-core Intel Xeon Gold 6130 Skylake processor (frequency set to 2.3 GHz) with 2 sockets and 2 AVX512 fused multiply-add units per core. Both architectures have 32KB L1 cache and 1024KB L2 cache per core. The first processor has a 24.75MB shared L3 cache, while the second one has a 22MB shared L3 cache. (c) a 32-core ARM ThunderX2 CN99xx processor (frequency set to 2.2 GHz) with 2 sockets and 2 Neon vector units per core. This high-end ARM microarchitecture has 32KB L1 cache, 256KB L2 cache per core and a 32MB shared L3 cache.

The OS is Debian GNU/Linux with kernel version 4.19, monitoring hardware counters using PAPI version 5.7.0. We compile the generated C code using gcc with the flags `-O3 -march=native -fno-align-loops` for the Intel x86 architectures. For the ARM AArch64 architecture, we use clang instead of gcc because we observed it produced faster code on our benchmarks. For AutoScheduler, we used the recommended template `conv2d_NHWC` from the TVM library. This template is significantly faster than the alternative `NCHWc`.

We evaluate performance over the convolutions of two networks: Yolo-9000 [Redmon and Farhadi 2017] and ResNet-18 [He et al. 2016].The sizes of their convolution layers can be found in Fig. 8.

To measure performance in a consistent, we embed the generated code in the TVM framework by using a *tensorize* operator. We measure performance using the TVM function `evaluator`, with parameters `repeat` set to 5 and `min_repeat_ms` set to 100ms. This means that TVM repeats the operation for at least 100ms, then it repeats

Benchmark	Problem sizes
	(K, C, H/W, R/S)
Yolo9000-0	32, 3, 544, 3
Yolo9000-2	64, 32, 272, 3
Yolo9000-4	128, 64, 136, 3
Yolo9000-5	64, 128, 136, 1
Yolo9000-8	256, 128, 68, 3
Yolo9000-9	128, 256, 68, 1
Yolo9000-12	512, 256, 34, 3
Yolo9000-13	256, 512, 34, 1
Yolo9000-18	1024, 512, 17, 3
Yolo9000-19	512, 1024, 17, 1
Yolo9000-23	28269, 1024, 17, 1

Benchmark	Problem sizes
	(K, C, H/W, R/S)
ResNet18-1*	64, 3, 224, 7
ResNet18-2	64, 64, 56, 3
ResNet18-3	64, 64, 56, 1
ResNet18-4*	128, 64, 56, 3
ResNet18-5*	128, 64, 56, 1
ResNet18-6	128, 128, 28, 3
ResNet18-7*	256, 128, 28, 3
ResNet18-8	256, 128, 28, 3
ResNet18-9	256, 256, 14, 3
ResNet18-10*	512, 512, 14, 3
ResNet18-11*	512, 256, 14, 1
ResNet18-12	512, 512, 7, 3

Fig. 8. Convolution benchmarks and sizes. The stride is 1 by default, unless marked with a * (stride 2 convolutions). Dimension k of Yolo9000-23 was padded to 28272 (a multiple of 16) to vectorize it on AVX512.

this process 1 + 5 times, i.e., the first iteration is discarded. Among the remaining 5, TVM removes the exterma and takes the average of the rest. Consistently with the majority of the reported performance experiments [Zheng et al. 2020a], we consider a hot cache hypothesis: memory is not flushed between each run.

Performance measurement. Fig. 9 presents the performance results, reported as a fraction of the peak performance of the multicore CPU. The performance of random sampling after 1000 runs is well above oneDNN and Mopt, and is comparable to AutoScheduler after 1000 runs (which is the recommended amount of time for it to converge [Zheng et al. 2020a]). For Yolo9000_00 and ResNet_01, the reduction size C is very small, so our microkernel-based approach does not have much reuse potential above the selected pair of microkernels. In comparison, AutoScheduler exploits the kernel dimensions R and S to increase the size of the reduction. For Yolo9000_23, the output of AutoScheduler is not even vectorized, which explains the huge performance difference.

A few Mopt results are missing due to reproducibility issues with the available artifact. We reported such situations with a performance at 0% of the machine peak. The performance of oneDNN is surprisingly low, despite our efforts to explore the relevant configuration and performance evaluation settings. However, we have confirmed with the Intel developers that such results are in line with their expectations.

Convergence rate. Fig. 10 compares the convergence rate of AutoScheduler and our random sampling method. We have run 6000 candidates in our space and we randomly picked 3000 of them, in order to produce the 8 traces in the graph on the right. To understand these graphs, let us recall how the autotuning process in AutoScheduler works: every 64 measurements, it rebuilds its cost model using the performance data collected so far. In particular, the first 64 candidates are picked randomly in the AutoScheduler search space, and model is updated every 64 runs.

Thanks to the structure of our space (resulting from staging the selection of microkernels and imposing the divisibility constraint), we observe that random sampling alone converges much faster than AutoScheduler despite the lack of a cost model. The high density of good candidates in our structured space results in a high probability of reaching performance close to the maximum after 10–20 runs only.

We also observed that AutoScheduler's output is not very stable. Sometimes it converges very late, eliminating much hope of reducing the number of runs below 3000. And the final result itself is not very stable, with up

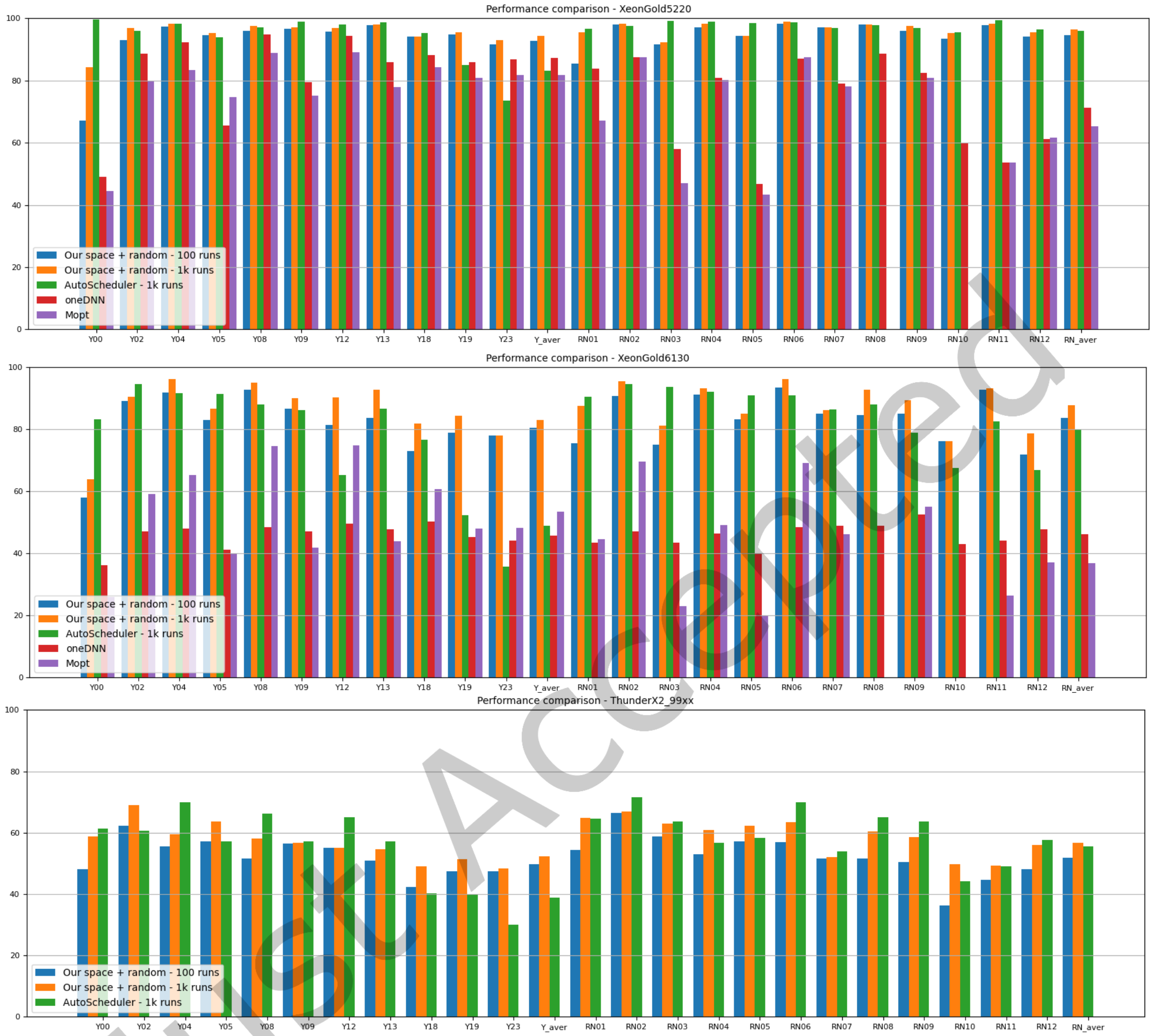


Fig. 9. Sequential performance comparison with AutoScheduler, oneDNN, Mopt for AVX512 (Intel Xeon Gold 5220 and 6130), and with AutoScheduler only for Neon (ARM ThunderX2), shown as percentage of machine peak. The averages given for each CNN are weighted by the amount of computation in every layer.

to 10% performance variation across autotuning experiments. AutoScheduler seems to suffer from insufficient flexibility in adapting its exploitation/exploration ratio, which results in its search algorithm getting stuck in local maxima for too long.

About compilation time, the offline microkernel code generation and compilation stage takes about 50 mn on a single machine (and only needs to be run once). Selecting configurations is almost instantaneous, and executing them takes 30 mn for 1000 configurations (less than 2 seconds to compile and measure a single configuration).

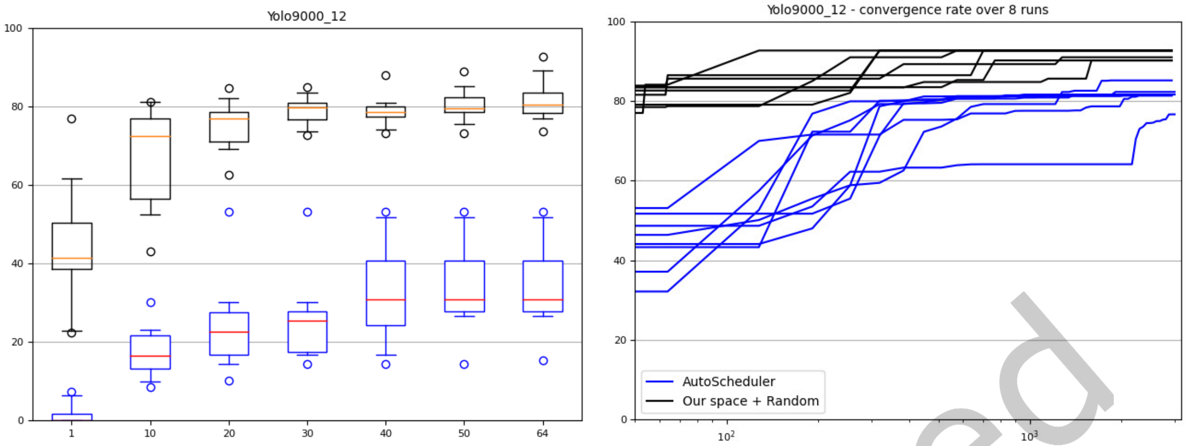


Fig. 10. Comparison of the convergence rate of 8 random samplings in our space, against 8 independent executions of AutoScheduler (in blue), for sequential code generation, on Yolo9000_12, targeting an Intel Xeon Gold 6130. (i) The left figure shows the maximum of the performance of the first 64 chosen implementations. The boxplot for AutoScheduler summarizes the 8 executions, while the boxplot for random sampling represents 20 executions. (ii) The right figure shows the best candidate found by AutoScheduler after each batch of 64 runs, and compares it to the best candidate found by random sampling for an equivalent number of runs.

AutoScheduler needs to refine and retrain a model every 64 measurements, so it takes about 2h to perform 3000 measurements. So, the average time spent per measurement is similar for both sides. It also needs to start over for every new problem size, while running 20 random samples in our search space takes a few tens of seconds per problem size.

6 ABLATION STUDY OF THE SEARCH SPACE PRINCIPLES

Important research questions remain, such as what aspect the structured search space contributes to improving the density of good candidates, and to what extent each component of its design contributes to the performance of the generated code.

6.1 Performance distribution in the search space

We would first like to characterize the density of good candidates. To achieve this, we simply perform a random sampling of the search space and report the performance distribution of the generated code. The methodology for this random sampling is the following: list all microkernels and combinations of microkernels that divide the problem sizes, then draw from this list uniformly, then draw a divisor of c (the reuse dimension) to nest the microkernel in a reuse loop, then draw uniformly over the set of pairs (dimension, factor) for the levels above the reuse loop, and (if applicable) insert the $\lambda_{\mathrm{seq}_d}\alpha.\,[\ell]$ specifier at the appropriate level. We perform these draws until completion of the configuration. Notice that this algorithm is not uniform over the space of configurations: it has a bias in favor of the larger factors for a dimension, which is preferable as it tends to avoid inadequately small tiles.

The resulting distributions are shown on Fig. 11. We observe that some problem sizes are easier to optimize than others. Reaching good performance is particularly easy for ResNet-11, for example. On the contrary, some problem sizes such as ResNet-01 and Yolo9000-00 are much harder to optimize, due to a small c dimension which makes the microkernel reuse strategy less efficient. In such cases, exploiting other dimensions (r and s) for

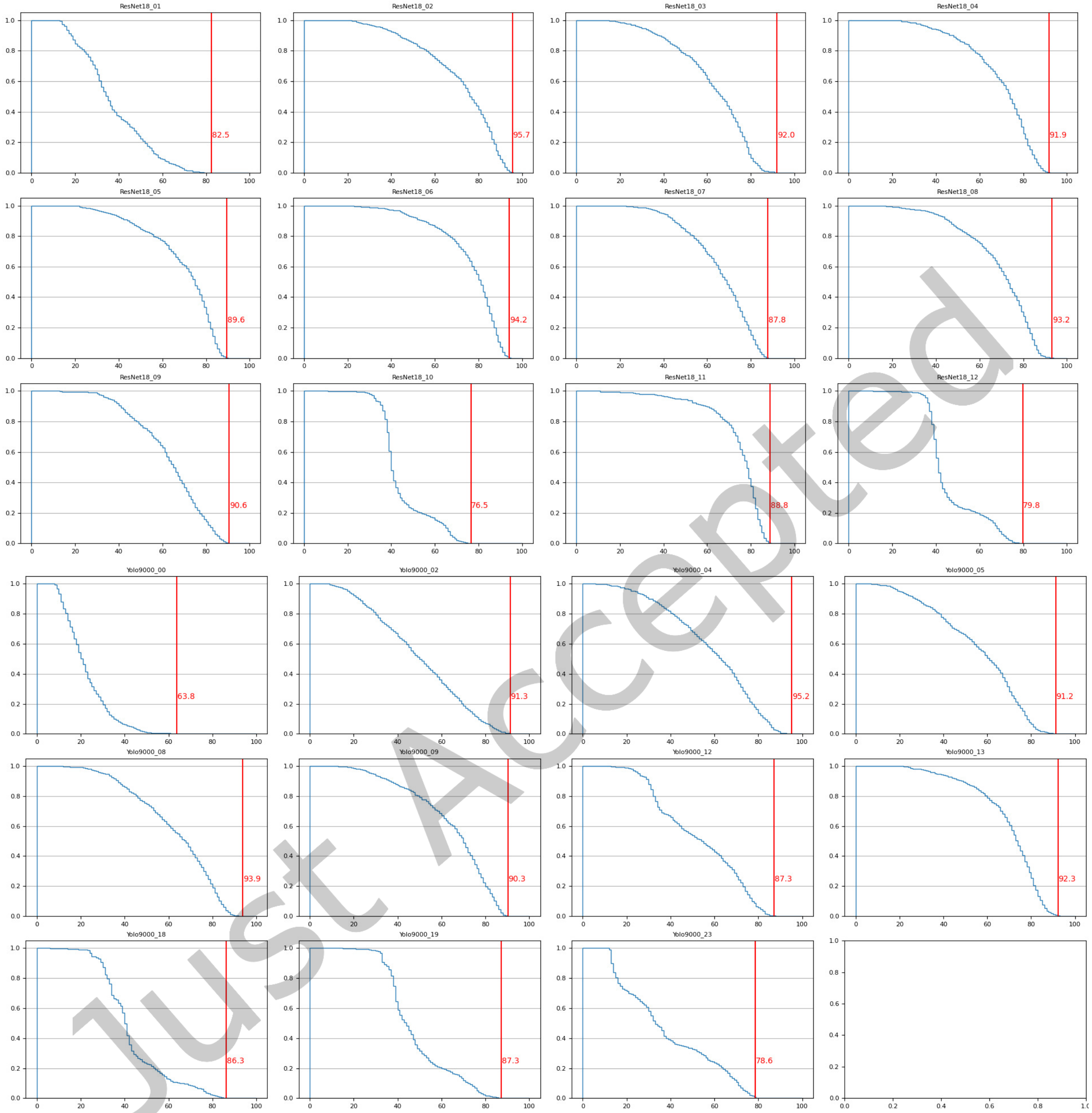


Fig. 11. Cumulative performance distribution for a random sampling algorithm, for 1000 sequential configurations, on an Intel Xeon Gold 6130 CPU. The horizontal axis is the percentage of machine peak. The vertical axis is the ratio of draws that has a performance above a given percentage of machine peak. The more area on the right side below the curve, the better the distribution is. The red bar marks the maximal performance observed over these 1000 draws.

reuse may be needed. Except for those difficult cases, the distribution is clearly favorable to random search. This explains why random search quickly approaches the best performing candidate after a few draws.

6.2 Evaluation of the combination of microkernels

The next question is to evaluate the performance benefits of combining microkernels. We consider the randomly drawn configurations from the previous subsection, and we split them into two sets: the set of configurations that use a single microkernel, and the set of configurations that use a combination of microkernel. Figure 12 compares the distribution of both sets of configurations on a variety of problem sizes.

We observe that the performance with and without combinations of microkernels is comparable when both possibilities are available. However, the last three Yolo9000 benchmarks only has combination configurations. Indeed, the size of the h dimension is the prime number 17 for these three benchmarks. Since 17 is above the unrolling limit (it breaks all register pressure limits), the only microkernels available in our space are those unrolled along the k and c dimensions only. And these yield around 30% of peak performance, while combining microkernels reach 85%.

To complement this analysis, we focused on the Yolo9000-18 benchmark and considered a microkernel with an unrolling factor of 17 on dimension h ($\mathsf{U}_{17,h}\mathsf{V}_k$). We ran 500 random configurations while forcing the use of this microkernel. We observed a maximum performance of 68% of the machine peak, clearly limited by register spilling.

This study confirms the importance of combining microkernels, especially when the problem sizes are too small and do not have small prime factors. In the majority of cases, it does not significantly impact the performance distribution.

6.3 Evaluation of the tile size divisibility constraint

Let us finally investigate the impact of the divisibility constraint at tile level, i.e., the pros and cons of forcing tile sizes to divide each other and the problem size along a given dimension. Note that we still enforce that any microkernel is executed as a whole, i.e., tiles are composed of complete microkernels; in other words, the size of tile dimension must always be a multiple of the corresponding microkernel size along that dimension.[1] If divisibility is not enforced at the tile level, some control flow may be needed for early termination of one or more tile dimensions.

In order to compare both spaces, we consider a different random selection algorithm, with two variations, for the divisible configuration space and for the non-divisible configuration space. This ensures we have an identical sampling bias across both spaces, to make the comparison as fair as possible.

The sampling algorithm is the following:

- First, we list all the microkernels and combination of microkernels that divide the problem sizes, then we select one of them uniformly.
- For each dimension d, we randomly pick the level of tiling l_d on this dimension, between 1 and 4 (4 being the height of the memory hierarchy, not including the register level).
- *For the non-divisible space:* we select uniformly l_d tile sizes between twice the microkernel sizes and the problem sizes, then we sort them in increasing order.
- *For the divisible space:* we consider k_d the ratio between the problem size and the microkernel size on dimension d. We build all the decompositions of k_d in l_d elements (greater than 1, if possible), and we select uniformly one of these decompositions.

[1] We evaluated the impact of this microkernel-specific constraint in Section 3.2.

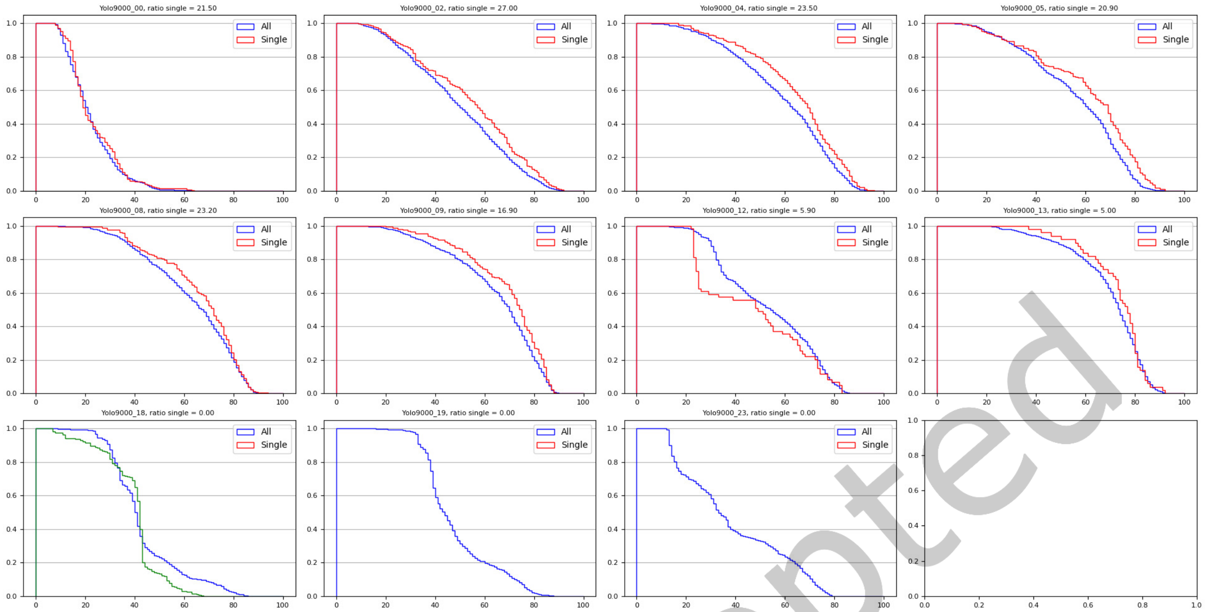


Fig. 12. Study of the impact of the combination of microkernels on the distribution on an Intel Xeon Gold 6130. We report the cumulative distribution of the space where combinations of microkernels are allowed (All) and where these combinations are forbidden (Single). The ratio reported is the percentage of configurations using a single microkernel, on the totality of the draws. Note that all the draws for the last 3 Yolo9000s are combinations of microkernels. For Yolo9000-18, we have added (in green) 1000 runs that use a single suboptimal microkernel. This microkernel falls outside of our classes of high-performing microkernels, but divides exactly the problem sizes. This is a situation where the combination of microkernels is particularly useful (nearly 90% of peak instead of 70%).

- Finally, to select the permutation, we consider the set of pairs (dimension, tile sizes or factor), plus the $\lambda_{\mathrm{seq}_d}\alpha.\,[\ell]$ specifier in case of combination of microkernels, and we pop uniformly elements of this set, until completion of the configuration.

Notice that a slight bias discrepancy remains in the tile size selection: the divisible space samples ratios between two levels of tiling, while the non-divisible space samples tile size directly. So, for example, it will be impossible for the divisible space to have a tile size whose value is strictly between half the problem size and the problem size, while for the non-divisible space no such constraint exist.

Figure 13 shows the distribution of 1000 random configurations, for the divisible space and for the non-divisible space. We observe that the maximal performance is similar across for both distributions. Also, observing the portion corresponding to the best performing configurations, we do not observe a significant trend. We conclude that the divisibility constraint does not induce performance loss in practice.

7 RELATED WORK

TVM AutoScheduler. AutoScheduler [Zheng et al. 2020a] is TVM's [Chen et al. 2018a] state of the art integrated autotuner for tensor operations (in particular CNNs). It is formed of several components:

- The *program sampler* builds randomly the loop structure (sketch) from some specific derivation rules. These rules reduce the size of the search space, for example by imposing constraints to the order of the loops. In

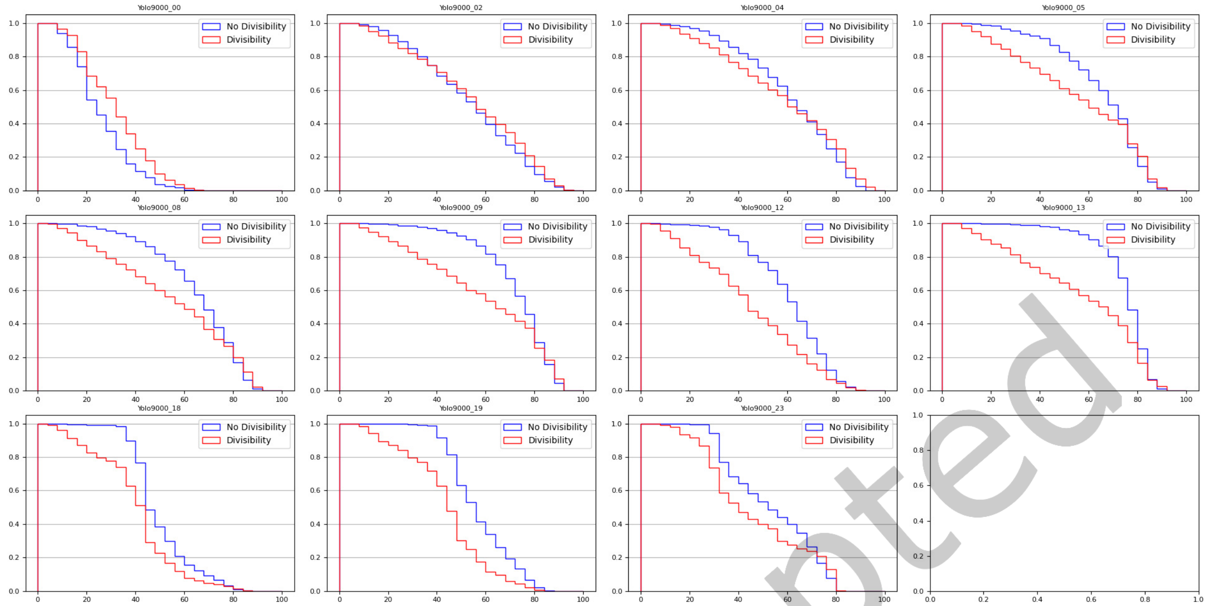


Fig. 13. Study of the impact of the divisibility constraint on an Intel Xeon Gold 6130. We choose randomly 1000 configurations in a space with the divisibility constraint on the tile sizes above the microkernel, then we choose 1000 other configurations without the divisibility constraint.

the case of a single tensor operation, any program generated through AutoScheduler's program sampler can be expressed in terms of the specifiers introduced in Section 3.3.

- The *performance tuner* proceeds iteratively over batches of 64 sampled programs. It collects newly sampled programs, and 20% of the best-performing programs so far. Then, it resorts to evolutionary search to mutate these programs. The mutated programs are evaluated through a cost model (a gradient boosting decision tree), to estimate their performance without having to execute them. From these estimations, a batch of 64 new programs is built and run.
  Performance measurements on the new batch allow retraining (and improving) the cost model. In particular, because the cost model is untrained on the first 64 runs, its selection is effectively random.
- The *task scheduler* partitions the different compute operations. All the operators of a given partition are optimized as a whole. In our context of a single operation, this part is not relevant.

Optimization of affine programs. To optimize affine programs, some methods are based on analytical models and operation research. This is the main approach used by polyhedral based compilers [Baghdadi et al. 2019; Bondhugula et al. 2008; Elango et al. 2018; Grosser et al. 2012; Vasilache et al. 2018; Verdoolaege et al. 2013] that leverage parametric integer linear programming. Although such approaches are well suited to expose parallelism [Feautrier 1992a,b] and coarse grain locality [Bondhugula et al. 2008], we believe it may not be the right formalism for tile size selection or register level optimizations. On the other hand, the ability to count points in a polyhedron [Barvinok 1994] allows to automatically generate (non-linear) cost models, which in turns enabled Li et al. [Li et al. 2019] to build and analytical model for the selection of a permutation scheme.[2]

Cloog [Bastoul 2004] is a powerful algorithm to automatically generate imperative code for scanning a union of polyhedra. Polyhedral compilers leverage such code generation capabilities, but face the challenge of dealing with a very general class of imperfect nests and transformations. It is difficult in such a broad context to compete with domain-specific optimizations. Our code generator involves simple polyhedron scanning algorithms, and the divisibility constraint enables generation of high-quality compiler friendly code without heroic efforts [Grosser et al. 2015].

Optimization of machine learning programs. There exist many compilers specialized for machine learning: PlaidML [Chen et al. 2019] using polyhedral techniques, XLA [Google [n. d.]] for TensorFlow [Abadi et al. 2016], Halide [Ragan-Kelley et al. 2013], or TVM [Chen et al. 2018a]. TVM, as opposed to most approaches, does not rely on numerical libraries. Its strategy is to select the best schedule using autotuning with an ML-based performance model. Contrary to our approach that decouples the search into microkernel optimization and loop tiling/permutation search, the TVM search space is flat. In TVM, optimizations related to strength reduction and register tiling are left to the compiler. TVM has been extended with FlexTensor [Zheng et al. 2020b] and Ansor (a.k.a. AutoScheduler) [Zheng et al. 2020a]. We also compare to AutoTVM, the auto optimizer of TVM. Telamon [Beaugnon et al. 2017] tackles this problem by building a very large, flat search space where optimization choices are tied together by dependency constraints. Then the exploration combines an elaborate performance model to prune the search space with feedback from actual executions.

A recent paper [Gibson and Cano 2022] considers the problem of transferring a schedule found by AutoScheduler across problem sizes. This transfer is performed by relaxing the ratio of the last tile along each dimension, in order to match other similar problem sizes. They report a significant speed-up in convergence, which is coherent with our observations. Indeed, the transfer conserves the inner loops, which contain a performant microkernel, which contributes significantly to obtain a good strategy.

Linear algebra and CNN libraries. Frameworks such as TBLIS [Matthews 2018] or TCCG [Springer and Bientinesi 2016] aim at creating portable optimized code for BLAS or tensor contraction kernels. These frameworks implement an efficient predefined scheduling scheme which is very effective, in particular for matrix multiplication [Goto and Van De Geijn 2008]. These frameworks take advantage of advanced optimizations: tensor transposition, tensor blocking or sub-viewing, data prefetching, vectorization, block scheduling, unrolling and register promotion. The register tile shape is predefined using expert knowledge on instruction level and register pressure. Thanks to aggressive autotuning and JIT/AoT code versioning, MKL [Wang et al. 2014] and oneDNN [Intel 2018] are the best available Intel libraries which implement all these techniques.

8 CONCLUSION

We presented a structured search space for tensor compiler construction and autotuning. Our approach allows a simple random search to match or outperform the state-of-the-art tool, AutoScheduler, at a fraction of the cost. This search space is based on the automatic generation and preselection of near-peak performance microkernels, and on the imposition of divisibility constraints on tile sizes and unroll factors. These principles allow pruning the search space, increasing the density of near-optimal candidates. The divisibility constraint can be relaxed, in cases where it does not allow a sufficient number of tiling scenarios, by allowing the sequential combination of microkernels.

[2] Note that the MobileNet results presented by Li et al. are actually not MobileNet but 2D convolutions with identical shapes as the CNN's original depthwise separable convolutions; we choose to leave these layers out to avoid propagating the confusion any further, but our results on those shapes are consistent with the results presented earlier.

Our results show that in exploring a search space to look for the best candidates, the structure of the space and its domain-specific pruning can be as important as the metric and the search strategy. This may sound obvious, but much of the recent work focuses on the latter design dimensions, using elaborate analytical modeling or learning processes. Of course, it would be interesting to further improve the effectiveness of domain-specific code generators and autotuners, by combining rather than choosing between these strategies. This is the topic of our current efforts, building on analytical as well as empirical, ML-based modeling. We also would like to broaden the search space to include transformations such as packing or prefetching, to carefully study their performance impact and determine when they are needed. And of course this study needs to be extended beyond CPU architectures, and beyond convolutions.

Acknowledgments. This work was supported in part by the Bpifrance Programme d'Investissements d'Avenir (PIA) as part of the ES3CAP project. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI'16)*. USENIX Association, USA, 265–283.

Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *IEEE/ACM International Symposium on Code Generation and Optimization, (CGO 2019)*, Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley (Eds.). IEEE, 193–205.

Alexander I. Barvinok. 1994. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research* 19, 4 (1994), 769–779.

Cedric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. IEEE Computer Society, 7–16.

Ulysse Beaugnon, Antoine Pouille, Marc Pouzet, Jacques Pienaar, and Albert Cohen. 2017. Optimization Space Pruning without Regrets. In *Proceedings of the 26th International Conference on Compiler Construction* (Austin, TX, USA) *(CC 2017)*. Association for Computing Machinery, New York, NY, USA, 34–44. https://doi.org/10.1145/3033019.3033023

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, New York, NY, USA, 101–113.

Huili Chen, Rosario Cammarota, Felipe Valencia, and Francesco Regazzoni. 2019. PlaidML-HE: Acceleration of Deep Learning Kernels to Compute on Encrypted Data. In *37th IEEE International Conference on Computer Design, ICCD 2019, Abu Dhabi, United Arab Emirates, November 17-20, 2019*. IEEE, 333–336. https://doi.org/10.1109/ICCD46524.2019.00053

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018a. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 578–594.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018b. Learning to Optimize Tensor Programs. *Advances in Neural Information Processing Systems* 31 (2018), 3389–3400.

Stephanie Coleman and Kathryn S McKinley. 1995. Tile size selection using cache organization and data layout. *ACM SIGPLAN Notices* 30, 6 (1995), 279–290.

Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. 2018. Diesel: DSL for Linear Algebra and Neural Net Computations on GPUs. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM, 42–51.

Paul Feautrier. 1992a. Some Efficient Solutions to the Affine Scheduling Problem: I. One-dimensional Time. *International Journal of Parallel Programming* 21, 5 (Oct. 1992), 313–348. https://doi.org/10.1007/BF01407835

Paul Feautrier. 1992b. Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time. *International Journal of Parallel Programming* 21, 6 (Dec. 1992), 389–420. https://doi.org/10.1007/BF01379404

Perry Gibson and José Cano. 2022. Transfer-Tuning: Reusing Auto-Schedules for Efficient Tensor Program Code Generation. In *31st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Chicago.

Google. [n. d.]. XLA : optimiser le compilateur pour le machine learning. https://www.tensorflow.org/xla?hl=fr

Kazushige Goto and Robert Van De Geijn. 2008. High-Performance Implementation of the Level-3 BLAS. *ACM Trans. Math. Software* 35, 1, Article 4 (2008), 14 pages.

Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letter* 22, 4 (2012).

Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Transactions on Programming Languages Systems* 37, 4, Article 12 (July 2015), 50 pages. https://doi.org/10.1145/2743016

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '16)*. IEEE Press, Article 84, 11 pages.

Intel. 2018. oneAPI deep neural network library (oneDNN). https://01.org/.

Rui Li, Aravind Sukumaran-Rajam, Richard Veras, Tze Meng Low, Fabrice Rastello, Atanas Rountev, and P. Sadayappan. 2019. Analytical cache modeling and tilesize optimization for tensor contractions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Michela Taufer, Pavan Balaji, and Antonio J. Peña (Eds.). ACM, 13 pages.

Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P. Sadayappan. 2021. Analytical Characterization and Design Space Exploration for Optimization of CNNs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 928–942.

Devin A. Matthews. 2018. High-Performance Tensor Contraction without Transposition. *SIAM Journal on Scientific Computing* 40, 1 (2018), C1–C24.

NVIDIA. 2018. CuDNN: GPU Accelerated Deep Learning. https://developer.nvidia.com/cudnn.

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

Joseph Redmon and Ali Farhadi. 2017. YOLO9000: Better, Faster, Stronger. In *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 6517–6525.

Gabriel Rivera and Chau-Wen Tseng. 1999. A comparison of compiler tiling algorithms. In *International Conference on Compiler Construction*. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 168–182.

Paul Springer and Paolo Bientinesi. 2016. Design of a high-performance GEMM-like Tensor-Tensor Multiplication. arXiv:1607.00145

Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Software* 41, 3, Article 14 (June 2015), 33 pages.

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. arXiv:1802.04730 [cs.PL]

Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimization* 9, 4, Article 54 (Jan. 2013), 23 pages.

Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. *Intel Math Kernel Library*. Intel, 167–188.

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. 2020a. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020b. FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 859–873. https://doi.org/10.1145/3373376.3378508