Efficient convolution optimisation by composing micro-kernels

Nicolas Tollenaere, Auguste Olivry, Guillaume Iooss, Hugo Brunie, Albert Cohen, P Sadayappan, Fabrice Rastello

Efficient convolution optimisation by composing micro-kernels

Nicolas Tollenaere* Auguste Olivry* Guillaume Iooss* Hugo Brunie†
Albert Cohen‡ P. Sadayappan§ Fabrice Rastello*

October 14, 2021

Abstract

Optimizing the implementation of tensor computations is essential to exploiting the full capacity of a given processor architecture on a wide range of scientific and machine learning applications. However, the complexity of the microarchitectural features that come into play when approaching the peak performance of the processor makes it very hard. Focusing on 2D convolutions, we observe a common weakness in all tensor compilers and libraries related to efficiently covering the wide variety of problem sizes occurring in real-world applications.

We propose TTile, a domain-specific code generator and autotuner for implementing efficient convolutions. Similarly to BLIS [30], TTile nests multiple levels of tiling above a vectorized tensor contraction microkernel. But unlike traditional approaches, we explore of a variety of microkernels and compose them to fit exactly the tensor shapes of a convolution. While this helps achieving consistently high performance on virtually all possible tensor sizes, our method also introduces more degrees of freedom in the optimization space, which makes it challenging for autotuning strategies. To address this, we leverage an analytical model of data movement [22, 25], and combine it with feedback-directed autotuning. We evaluate TTile as a stand-alone compiler and also as a complement to TVM [8] on recent Intel x86 microarchitectures.

1 Introduction

Tensor computations are at the core of many applications in scientific computing, data analytics and machine learning. Their optimized implementation is therefore of considerable interest. The current options for optimizing the implementation of a tensor operator, such as a 2D convolution, are:

- **Polyhedral compilers** like Diesel [11], Polly [16], Pluto [6], PPCG [32], Tensor Comprehensions, [31], Tiramisu [2] automatically generate multi-level tiled code for any affine loop computation such as 2D convolutions. However, a significant limitation is that none of them can directly optimize across tile sizes, which is critical for efficient CNN implementations.
- **Vendor libraries** like oneDNN [19] and cuDNN [24] have been manually optimized by expert HPC and software engineers. While these implementations use JIT optimization, they cannot fully adapt to every given tensor extents of a CNN layer in a DNN pipeline.

*Inria
†Lawrence Berkeley National Laboratory
‡Google France
§University of Utah

- **Autotuning** can be performed by systems like AutoTVM [9] or AutoScheduler [34] both part of the DNN specific compiler TVM [8]. An expert user provides a multi-level tiled loop structure along with a specification of a search space including permutation among subsets of loops and parametric tile sizes. A search process guided by a dynamically constructed machine learning model [9] iterates through tiled loop configurations, where code is generated, compiled and executed on the target platform. AutoTVM has been demonstrated to outperform polyhedral compilers [9], and AutoScheduler to outperform AutoTVM [34], but both of them still do not match reference libraries like oneDNN.
- **Analytical modeling and optimization**. Recent research has shown that a comprehensive characterization and optimization across all possible tiled loop configurations for CNNs is feasible [22]. The approach is semi-automatic: manual reasoning to build analytical cost models for data movement and pruning, in conjunction with the automated resolution of nonlinear optimization problems to optimize tile sizes. The innermost loops use a manually created *microkernel*, similar to the BLIS one [30] and vendor libraries. Over a set of 20+ CNN stages from 2 DNN pipelines, performance was shown to be consistently higher than state-of-the-art autotuning [8] and comparable or better than the state-of-the-art oneDNN library [19].

The best performing implementations in the state-of-the-art are based on a single microkernel. In this paper, we challenge that assumption, by claiming that **limiting ourselves to a single microkernel for all problem sizes is too restrictive**. Indeed, we show that there is a collection of well performing microkernels, and some of them might better fit the considered problem sizes.

We also claim that we should use **combinations of microkernels** of different sizes to cover exactly the whole space, as an alternative of partial tiles and the padding technique. This is especially critical for computation whose sizes are small: indeed the effect of padding or of a partial tile is significant on the performance. Therefore, combining microkernels allows us to have more consistant and performant implementation. In particular, such situation happens for the convolutional layers of neural networks.

Our main contribution is an end-to-end compiler flow for the optimization of tensor operations, with these properties:

No partial tiles: It is possible to compose *microkernels of different shapes* to cover the iteration space, obviating the need for mixing full tiles (optimized) with partial tiles (suboptimal or unoptimized).

Hybrid compilation strategy: We compose techniques with complementary strengths and weaknesses, such as domain-specific compilation, autotuning and analytical modeling, in order to use them effectively while compensating their limitations.

- To provide an example of the limitations of analytical modeling, some recent efforts [22, 25] developed analytical models for optimizing CNNs, establishing tight data-movement bounds in an idealized model of computation. However, their experimental results on real processors show that it is not as effective. A long-time barrier to accurate performance modeling is that architectural features like out-of-order execution or hardware prefetchers are virtually impossible to incorporate into a model for code optimization.
- Another example of limitation, concerning autotuning techniques, is the explosion of the size of the search space: the full space of possible tiled loop configurations, choice of tile sizes and microkernels is enormous and only a tiny fraction is feasible to search. Autotuners resort to manually pruning the search—e.g. scheduling templates in AutoTVM [8]—and/or restricting its expressiveness at the expense of missing high-performing variants—e.g. narrowing to a single microkernel.

Our hybrid strategy overcomes these limitations by combining them: we use experimental evaluation of microkernels to prune the search space early, and we also use analytical modeling to guide autotuning.

Decoupling of the search space: Both the modeling work of Olivry et al. [25] and the AutoTVM

autotuning framework [8] take a uniform or *"flat"* perspective of the tiled iteration space of a computation. In contrast, we use a *"split"* approach where a small set of innermost loops is decoupled from the remaining outer tile loops for the purpose of optimization. The inner set of loops constitute a microkernel that we separately optimize using extensive autotuning to generate a collection of high-performance variants. We then use a combination of analytical modeling and much smaller scale autotuning to optimize the outer tile loops.

While our approach applies to a wider class of tensor computations, we focus on CNNs with 2D convolutions in the evaluation. The main contributions of the paper are as follows:

- Instead of a single fixed (hand-coded) microkernel used by prior approaches [30, 22], we conduct a comprehensive search among the space of possible microkernel configurations, identifying a set of microkernel instances as base building blocks for use in generating optimized code for every specific convolution instance.
- For a specific convolution and microkernel, a judicious mix of automated analytical modeling along with limited autotuning allows to optimize the tile loops enclosing the microkernel.
- For a specific convolution, a combination of several microkernels can be considered using imperfectly nested loops—or *"beyond perfect"* tile loops.
- Experiments with all 20+ CNN layers from 2 ML inference models (ResNet-18 and Yolo-9000) matches or outperforms the state-of-the-art oneDNN library [19] and TVM [8] autotuning frameworks. We observe that we are able to automatically match or outperform the performance of vendor libraries without copying and packing. This goes against the established methodologies [30, 19] and raises open questions about the impact of this optimisation on performance.

The rest of the paper is organized as follows: Sec. 2 provides a high-level overview of our approach. Sec. 3 shows the importance of the divisibility constraint, which motivates the choices made in the design of the search space in Sec. 4. Sec. 5 details the search strategies and Sec. 6 describes our code generator. Sec. 7 reports experimental comparisons against state of the art frameworks and libraries. Sec. 8 discusses related work before the conclusion in Sec. 9.

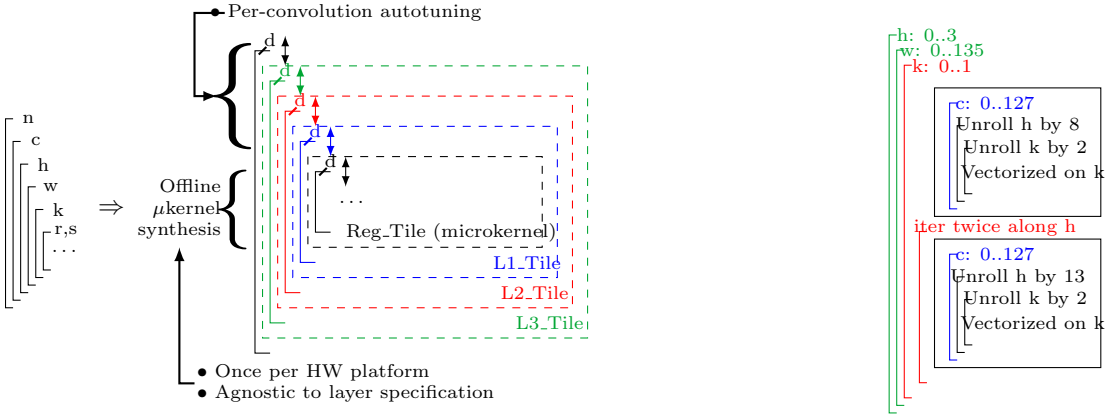
2 Overview of the Approach

Let us now present a high-level overview of our optimization approach. Its rationale derives from the following observations based on the advances made as well as the challenges faced by prior optimization efforts:

- Accurate *fine-grained* performance modeling of a multi-level tiled loop computation down to the innermost levels is extremely challenging and has not been achieved for complex computations like CNNs by any prior effort, but *coarse-grained* analytical modeling of the memory hierarchy has been shown to be effective, e.g., the work of Li et al [22]. Their use of a microkernel made the analytical modeling coarse-grained and sufficiently accurate, but by using a fixed microkernel they limited the design space they explored.
- Autotuning is an effective approach to overcome the challenges of accurate performance modeling. The main blocker to its application to multi-level tiled loops is the enormous explosion of the optimization space and its effective traversal. Adding degrees of freedom with the ability to select among multiple microkernels, and to combine them, makes the problem even more challenging. AutoTVM does find good solutions but only by using expert-engineered scripts that limit the search space.
- A key insight behind the optimization framework presented in this paper is that we can explore the

```
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[k, h, w] = K[k, c, r, s] * I[c, h + r, w + s]
```

Figure 1: 2D Convolution (unit stride).



(a) Multi-level tiling and split two-level optimization. **(b) Yolo9000 layer 5.**

Figure 2: Code generation sketch using microkernel composition. Convolution sizes are $K = 64$, $C = 128$, $H = W = 136$ and $R = S = 1$. Note that $136 = (8 + 13 \times 2) \times 4$. Loop colors in (b) match the cache level they fit into.

full design space effectively using a split two-level strategy:

1) Develop a collection of *microkernels* for a given target hardware platform via extensive one-time autotuning. The set of microkernels is agnostic to the actual sizes of CNN layers to be optimized. At this level, accurate performance modeling is infeasible, but the optimization space is quite manageable through autotuning via the execution of all potential variants of interest (detailed in Sec. 5.1). We then combine them in order to exactly fit a problem size without any partial tiles (detailed in Sec. 5.3).

2) Use a combination of autotuning and analytical modeling to dramatically prune the space of possible tile loop permutations and degenerate loops (explained shortly). This is detailed in Sec. 5.2 to Sec. 5.4.

We illustrate our integrated approach to optimized code generation for convolutions on Fig. 1 and Fig. 2. 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 in a band are active versus degenerate. We first identify the microkernels that are performing well in isolation. Then we pick, among those, one or two combined microkernels that fit the considered problem size. Then, we use analytical modeling to identify the active loops in each band and the permutation within active loops in a band. This modeling is similar to other recent efforts on analytical model-driven optimization of CNN (e.g. the work of Olivry et al. [25] or that of Li et al. [22]), so they will just be summarized in this paper. Finally, we use autotuning to search across the dramatically pruned space of outer-level tile configurations.

Fig. 2(b) shows the code generated by our optimization framework 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 extents have been covered and therefore the outer-most level of tiling loops (color-coded black in Fig. 2(a)) are degenerate. This example illustrates how combining two well-performing microkernels can be used to divide a problem size. It also shows that only a subset of the tile loops at any level are non-degenerate, thus demonstrates the importance of having an analytical model to prune the search space, before exploring it using autotuning.

3 Divisibility constraint and microkernels

In this section, we demonstrate the importance of combining microkernels instead of relying on (suboptimal) 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 sizes divide exactly the problem sizes, then it fits perfectly, and we observe a peak in performance. If the microkernel sizes does not divide exactly, the classical options are (i) to have a partial tile, smaller than the microkernel, that finishes the coverage of the iteration space; or (ii) to *pad* the space in order to continue using the microkernel one last time, at the cost of additional computation. In this paper, we take a third route: (iii) to combine two of the best performing microkernels to cover the space without partial tiles. The method to determine the best performing microkernel will be described in Section 5.1, and the selection algorithm is explained in Section 5.3.

Figure 3 compares the sequential performance of small matrix multiplication implementations, for problem sizes $J=K=128$ and $8\leq I\leq 49$, on a Intel Xeon Gold 6230R CPU (Cascade Lake-SP, with AVX512). The performances are shown as percentage of the absolute peak performance, corresponding to the maximal utilisation of the two vectorized FMA units of the architecture.

MKL, *Blis* and *libxsmm* 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 the i dimension. Libxsmm also considers combination of microkernels, but restricted to predefined sizes such as multiples 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*" is the performance of code generated by our framework, but only

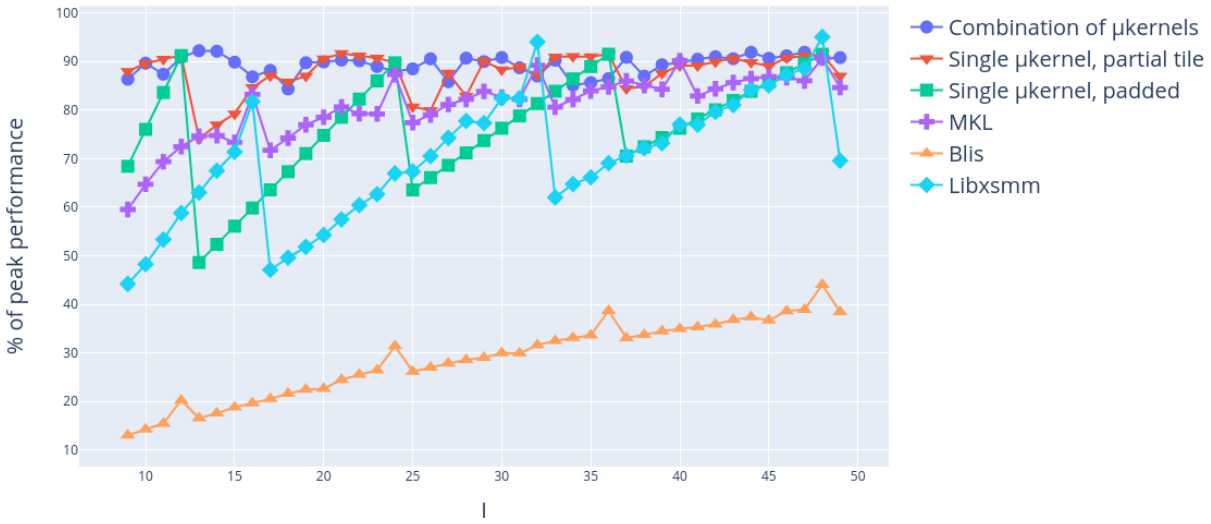


Figure 3: Performance of small matrix multiplication kernels, for $J = K = 128$ and $8 \leq I \leq 50$.

using the BLIS microkernel, with an unrolled partial tile. We observe a fluctuation of periodicity 12 in its performance. Notice that for values of I with a low modulo 12, the performances are worse than for the high modulo 12, because of 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 assumed that the padding overhead is free. As expected, the performance for low modulo are quite low, due to the significant additional amount of computation performed. However, this penalty decreased with the size of I.

Finally, "*Combination of microkernels*" corresponds to our microkernel combination strategy. The performances are more stable for any value of I.

This 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). Therefore, we build our optimisation space around this constraint, as described in Sec.4.

4 Defining the Optimization Space

In the following, 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 data in (vector) registers across iterations through unrolling and register promotion; (iii) adequate Instruction-Level Parallelism (ILP) to hide the latency of pipelined functional units (multiply-and-add).

The *iteration space* is the set of integer vectors taken by the loop indices enclosing a given computational statement. *Tiling* [28, 10] is a loop transformation that partitions the iteration space into sets,

```
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)
            μkernel_gemm_{6,32}(C, A, B, i_t, j_t, k)
```

Figure 4: Tiled sgemm with microkernel.

called *tiles* and executed atomically. We only consider programs with rectangular iteration spaces, and rectangular tiling. Tiled code has additional loops compared to the original code: loops over tiles, and loops inside a tile. This 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. 4 shows a tiled matrix multiplication kernel as an illustrative example. It relies on an (inline) fully-unrolled and vectorized microkernel of size 6×32.

We use lowercase for problem dimensions (i, j, k), i.e. loop iterations, and uppercase to name the (possibly symbolic) upper bound on each dimension (resp. I, J, K), a.k.a. problem size. We also assume that any dimension is either parallel—i and j—or a reduction—k and all dimensions are permutable (loop interchange). While associativity can be used to parallelize a reduction, we do not exploit it.

In the class of computations we consider, 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 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.

As mentioned previously in Sec. 3, 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. 4. When tile sizes do not divide tensor shapes, the traditional approach involves conditional execution or padding to manage partial tiles. *We consider a broader optimization space, using a collection of microkernels so that their combination eliminates the need for partial tiles.* We relax the *divisibility constraint* that must be satisfied in order to avoid partial tiles, enabling the ability to compose multiple, fully-optimized microkernels.

Our code generator is driven by a so-called *optimization scheme*. Conceptually, it can be seen as as a specialized abstraction, higher level than TVM schedules. An optimization scheme 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 sizes 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 $\mathsf{V}_\bullet$.
- $\lambda_{\mathsf{seq}_d}\alpha.[\ell]$, where $\ell = [(r_i, a_i)]_{1 \leq i < s}$ is a list of $s \geq 2$ 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

```
for (j = 0; j < 128; j+= 16) {
  for (i = 0; i < 72; i+= 6)
    for (k = 0; k < n_k; k+= 1)
      μkernel_gemm_{6,16}
  for (i = 72; i < 128; i+= 7)
    for (k = 0; k < n_k; k+= 1)
      μkernel_gemm_{7,16}
}
```

Figure 5: Microkernel composition example.

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 $[\mathsf{R}_i, \mathsf{R}_j, \mathsf{R}_k]$. An implementation for higher performance, based on the BLIS [30] microkernel for floats (f32) on AVX2 is:

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

The generated code contains a microkernel of size $(i = 6, j = 16, k = n_k)$ known to be quite efficient as it requires only 15 vector registers and exposes enough ILP (12 independent multiply-add instructions issued between two accumulation steps) [30]. 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 for j and k). State of the art libraries rely on fixed-size microkernels and tuned tiles sizes, and thus introduce partial *non-optimized* 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 scheme:

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

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

5 Optimization space exploration

We propose a novel optimization algorithm capable of exploring the expressive optimization space introduced in the previous section. This algorithm uses a combination of analytical modeling and autotuning; it is split into the offline optimization of the microkernel (register level), before specializing on a particular convolution and deciding on a tiling structure (cache level). Among all the possible combinations for a tiled convolution, we apply the following pruning strategy:

- we only use the best performing microkernels: the performance of this portion of the code limits the performance of the whole application, so we need to ensure that it runs as efficiently as possible;

- we forbid partial tiles, which is a source of slowdown, especially 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 speed up the search, we use a metric that focuses on the variations which maximize the sizes of the reduction loop above the microkernel, and the operational intensity of the computation.

Finally, we consider multiple parallelisation strategies, allowing to collapse and parallelize across multiple non-reduction loops.

The full algorithm is summarized by Fig. 6:

Step 1. We measure the performance of many microkernels in isolation. The *set of microkernel candidates* is formed of the best-performing ones (Sec. 5.1). This phase is problem size agnostic but specific to each target architecture.

Step 2. For each microkernel candidate, accounting for the problem size and cache sizes, we determine the *best loop permutation* (list of loop dimensions) enclosing the microkernel, that is, the main structure of the loop nest(which dimensions are tiled at each level). Tile sizes are not determined at this point. This permutation is obtained through operational research, using an analytical model of the footprint, data movement and reuse across tiles (Sec. 5.2).

Step 3. From the selected microkernels and loop permutations, we generate the *space of optimization schemes* (Sec. 5.3). In general, we require any tile size picked at a given dimension to be a multiple of the size of its sub-tiles and to divide the (full) problem size along that dimension. Except for microkernels where such a restriction would be impractical: e.g. Yolo9000-8 has a problem size $H = 17$, which is a prime number and all microkernels of size 17 have terrible performance. We leverage the λ_{seq} specifier to compose microkernels of size 5 and 6 to solve the problem: microkernels of size $\{\mu S = 3, \mu H = 5, \mu K = 4\}$ and $\{\mu S = 3, \mu H = 6, \mu K = 4\}$ can be combined by sequencing two iterations of the latter before one iteration of the former, fulfilling the divisibility constraint without compromising performance (Sec. 5.3).

Step 4. At this point, the search space is already tractable for autotuning (a few hours per convolution), but we can do better. A simple metric (detailed in Sec. 5.4) sorts all the resulting schemes, from which we can retrieve any number of candidate implementations. We pick the top 200.

Step 5. Optionally, we may integrate the top-performing scheme as a sub-tree in a TVM program. We use this ability to facilitate performance comparisons, and to automatically upgrade the optimal sequential scheme into a parallel implementation (see Sec. 5.5).

5.1 Microkernel definition and evaluation

We identified 4 different unrolling schemes to form efficient microkernels for a 2D convolution. Each one is suitable for a different class of tensor shape and convolution operator. In all 4 schemes, dimension k is selected for vectorization because it contains the simplest access pattern among w, h and k. Unrolling takes place along k and h, and optionally along the convolution kernel (stencil) dimensions r and s to form the 4 schemes:

- $[\mathsf{T}_{512,c}, \mathsf{U}_{\beta,h}, \mathsf{U}_{\alpha,k}, \mathsf{V}_k]$
- $[\mathsf{T}_{512,c}, \mathsf{U}_{3,s}, \mathsf{U}_{\beta,h}, \mathsf{U}_{\alpha,k}, \mathsf{V}_k]$
- $[\mathsf{T}_{512,c}, \mathsf{U}_{3,r}, \mathsf{U}_{\beta,h}, \mathsf{U}_{\alpha,k}, \mathsf{V}_k]$
- $[\mathsf{T}_{512,c}, \mathsf{U}_{3,r}, \mathsf{U}_{3,s}, \mathsf{U}_{\beta,h}, \mathsf{U}_{\alpha,k}, \mathsf{V}_k]$

where α and β are the sizes of the microkernel along the k and the h dimension, respectively.

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 the first scheme on

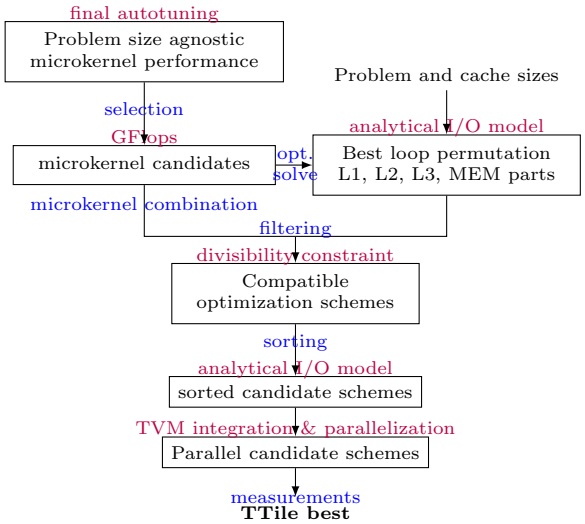


Figure 6: Flow of the optimization algorithm.

AVX512 are shown in Fig. 7 (on an Intel Xeon Gold 6130, frequency set to 2.1 GHz, Debian, kernel v4.19, and hardware counters monitored with PAPI v5.7.0).

We observe that the graph is roughly convex with some local fluctuations. Many microkernels are near-optimal for a given unrolling scheme. We select the ones above 85% optimal and partition them into classes of fixed sizes for C, S, R, and where H belongs to an interval defining the class. For example, $\{[\mathsf{U}_{\beta,h}, \mathsf{U}_{2,k}, \mathsf{V}_k], 8 \leq \beta < 15\}$ is one of the class of microkernels that is selected for AVX-512, as shown with the leftmost red vertical rectangular contour on Fig. 7.

This step is problem size agnostic and needs to be done only once per target architecture.

5.2 Loop permutation above the microkernel

The next step is to find a suitable permutation of tiling loops, for fixed values of parameters and for each candidate microkernel. For example, for Yolo9000 layer 0, the permutation that is chosen for microkernel $[\mathsf{U}_{8,h}, \mathsf{U}_{2,k}, \mathsf{V}_k]$ is (from outer to inner): $\big[K, H, W, H, H, S, R, W, C\big]$. This permutation is found using an analytical model of the data movement across all cache levels, similar to the one taken in recent work by Li et al. [22] and Olivry et al. [25]. For a given permutation, we derive an analytical expression of the data movement volume at each cache level, as a function of tiling loop extents and cache sizes. This is done by computing the footprint of each array at each level of the loop nest, as well as the level at which the total memory footprint exceeds the cache size.

First, a pre-processing step prunes the space of all possible loop permutations. Indeed, we have (7!) loop permutation for each of the 4 remaining level of memory, and many permutations are equivalent in term of data movement cost, or can be shown to have worse data reuse than others. Following the methodology proposed by Olivry et al. [25], we manage to reduce the number of considered permutations from 7! to only 6 (for each level of the memory hierarchy).

Then, for given values of array sizes, the analytical model is fed to a non-linear problem optimizer

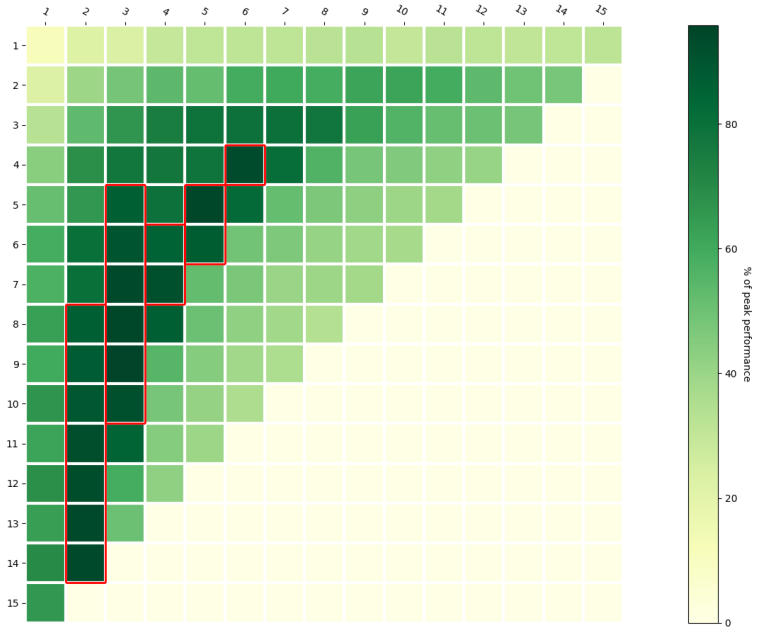


Figure 7: Performance of microkernels in isolation for AVX512 in percentage of the machine peak, for $R = S = 1$. Microkernel sizes—α along the k dimension (horizontal axis) and β along the h dimension (vertical axis)—vary between 1 and 15. Only the upper-left triangle was evaluated. Red-bordered microkernels are the ones selected (offline) for our algorithm.

which select the best permutation overall. The result is actually richer than that, as the solver also produces (generally non-integral) tile sizes that minimize the overall data movement [25]. Yet we observed that these tile sizes are not necessarily useful, as finer-grained performance considerations come into play when getting closer to the peak performance of the machine. We thus only retain the loop permutation and delay tile size selection to a later autotuning step.

Interestingly, we observe that the permutation remains stable when varying the unroll factor of microkernels along the h dimension. This is easily understood, considering that the loop permutation mostly relates to cache-level optimization while microkernels operate at the register level. As a result, we use the same permutation and only perform the analysis once per unrolling factor along k, r and s. As a fortunate side-effect, this means that when combining two microkernels, since they only differ in their unrolling factor along h, we can be sure that the loop order above both of them is the same.

5.3 Space of the valid optimization schemes

Microkernels and combination We consider each class of microkernels and select those whose sizes divide the problem sizes. Then, we look for the combination of two microkernels differing only along the h dimension that allow to cover the size of the h dimension. For all pairs of microkernel 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 AVX512 microarchitectures we considered. Indeed, the classes of microkernels identified in Section 5.1 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 $\{[\mathsf{U}_{\beta,h}, \mathsf{U}_{2,k}, \mathsf{V}_k], 8 \leq \beta < 16\}$, assuming that the problem size along the K dimension is even, 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 scheme Given (i) a single microkernel that divides the problem sizes, or a microkernel combination that divides the problem sizes, and (ii) a loop permutation, completion into an optimization scheme proceeds as follows:

- Set the optimization scheme (the list of specifiers) to the one corresponding to the chosen microkernel or combination of microkernels.
- For each dimension, consider the divisors of the problem size divided by the microkernel size. These divisors need to be allocated to the different strip-mined occurrences of the dimension across the whole permutation. There are multiple solutions and the algorithm considers all of them. Notice that there are (by construction) at most 4 occurrences of a dimension in a loop permutation, which limits the amount of possibilities.
- For each element d in the loop permutation, consider the product π of the divisors allocated to this occurrence and insert the corresponding specifier $\mathsf{T}_{\pi,d}$ to the left of the current scheme.
- When considering a sequence of two microkernels at dimension d with the combination $a \times h_1 + b \times h_2$, insert $\lambda_{\mathsf{seq}_d} \alpha.\,[l]$ at any occurrence of dimension d in the scheme, and consider all possibilities of placement of this insertion. The value of the list ℓ is $[(a, h_1), (b, h_2)]$.

Example Consider the range of microkernels:

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

for an AVX512 architecture, and the Yolo9000-13 problem sizes $(K, C, H/W, R/S) = (512, 256, 34, 3)$. Assume that the corresponding loop permutation found was:

$$[[K, H], [W, H], [H], [S, R, W, C]]$$

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. Now, we need to distribute the multiples of the other dimensions across the different levels of tiling described by the loop permutation:

- The k dimension is trivial: there is only one loop above the microkernel and the microkernel size along this dimension has a footprint of 32. Thus, we need to tile by a factor of 16 on the outer loop to reach 512.
- The c dimension is also trivial: the only loop need to be tiled by a factor of 256. Likewise for the r and s dimensions, the tiling factor should be 3 for both.
- The h dimension is already managed by the combination of microkernels. We have 3 locations where we can place the $\lambda_{\mathsf{seq}_h} \beta.\, [(2, 11), (1, 12)]$ compositon of two microkernels. Let us consider the outer one.
- The w dimension has 34 to be distributed across 2 level of tiling. There are 4 combinations: 34×1, 17×2, 2×17 and 1×34. Let us consider the second one.

The resulting optimization scheme (among many) is:

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

5.4 Further pruning of the optimization space

At this stage, we have a space of sequential scheme candidates, whose size range from a few hundreds to tens of thousands of potential schemes, depending on both the size of the problem and the architecture. Our objective is to prune this space by combining two criteria. The first criterion is to maximize the *Operational Intensity*, that is, the number of operations divided by the volume of data movement. The second criterion is to reduce the overhead of *control flow, control misprediction, and data misprefetching*. In particular, the larger the size of the reduction loop above the microkernel (along dimension c) is, the better the scheme is in relation to this criterion.

So, we propose the following metric, to focus on a region where there is at least one of the best performing schemes:

- First, select 40% of the schemes that have the largest reduction sizes (on c) above the microkernel.
- Then, sort these schemes according to their volume of data movement. The smallest volume is the best candidate, and we select up to 200 candidates.

5.5 Integration and parallelization with TVM

So far, we focused on optimising sequential performance. We have interfaced our tool with TVM to facilitate comparisons with the state of the art and to generate parallel code. Because TVM does not have currently a notion of microkernel, we use its *tensorize* construct to import a generated C code

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

Figure 8: Convolution benchmarks and sizes. The kernels marked with a * are stride 2, else stride 1. Dimension k of Yolo9000-23 was padded to 28272 (a multiple of 16) to vectorize it on AVX512.

(see Sec. 6) corresponding to the innermost loops inside TVM's Python intermediate representation of a convolution. These innermost loops are the ones up to dimensions h and w, which are usually part of the L1 tiling level. By construction, these loops will include the microkernel and some of the L1-resident loops, including the innermost reduction loop (on c) enclosing the microkernel.

This allows us to express completely a sequential scheme in the TVM framework. Then, we consider the loops above the tensorized section, and we consider several strategies to introduce parallelism:

- We consider the biggest band of non-reduction loops (to have enough iterations), collapse it and parallelize the resulting loop.
- We regroup the reduction loops at the innermost level and collapse/parallelize the rest of the loops.
- We regroup the reduction loops at the outermost level and collapse/parallelize the rest of the loops.

We add these strategies as a part of the optimization space, and consider all variations when evaluating the performance of the candidates.

6 Code generation

We now describe how to generate C code from a computation specification, problem size and the associated optimization scheme. Generating a loop requires to know the size of the sub-tiles, so our code generator proceeds from innermost outwards. Calling a *sub-scheme* the suffix of an optimization scheme, at a given step the already generated code (that corresponds to inner levels) is fully specified by the corresponding sub-scheme. In the following, the size of a sub-scheme refers to the size of the corresponding (parameterized) sub-iteration space. Taking the example from Sec. 4, the sub-scheme 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.

6.1 Overview of the code generation algorithm

Our code generator iterates right to left on the optimization scheme 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 strategies. In our case, the list of the λ_{seq} specifier is always of size 2. The corresponding rewriting rule is:

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

where S is the sub-scheme following the λ_{seq} specifier and $S[\alpha/v]$ is this sub-scheme 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).

6.2 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-schemes inside the Seq.
- **Vectorization V_d:** Considering the definition of the computation described in Sec. 4, one may determine which operations should be vectorized by traversing the graph starting from the loads:
  - $\mathsf{read}\,(T,f)$ is vectorized if d appears in the access function f.
  - $\mathsf{Op}\,(x,y)$ is vectorized if one of its operands (x or y) is vectorized. If one of them is a scalar, it is broadcasted.
  - $\mathsf{write}\,(v,T,f)$ 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 $\mathsf{U}_{k,d}$:** Unroll the computation over the d dimension k times by duplicating the generated code of its sub-scheme, while updating the value of the loop index on the d dimension in each duplication.
- **Tiling $\mathsf{T}_{k,d}$ or R_d:** Add a loop over the generated code of its sub-scheme that iterates k times, and whose value is increased by the value of the sub-scheme. In the case of R_d, one may deduce the correct number of iterations by comparing the size of the sub-scheme with the problem sizes. This changes the current loop index in use over the d dimension.

7 Performance results

In this section, we evaluate the performance of the code generated by TTile with several state-of-the-art approaches: oneDNN [19] (Intel library, V2.3), AutoTVM [8] (autotuning, v0.8.dev0 of June 2021), Autoscheduler [34] (Ansor, autotuning, in TVM) and Mopt [22] (analytical modeling).

Setup The experiments were carried out on two architectures: (a) a 18-core Intel Xeon Gold 5220 Cascade Lake processor (frequency set to 2.2 GHz) with 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) 2

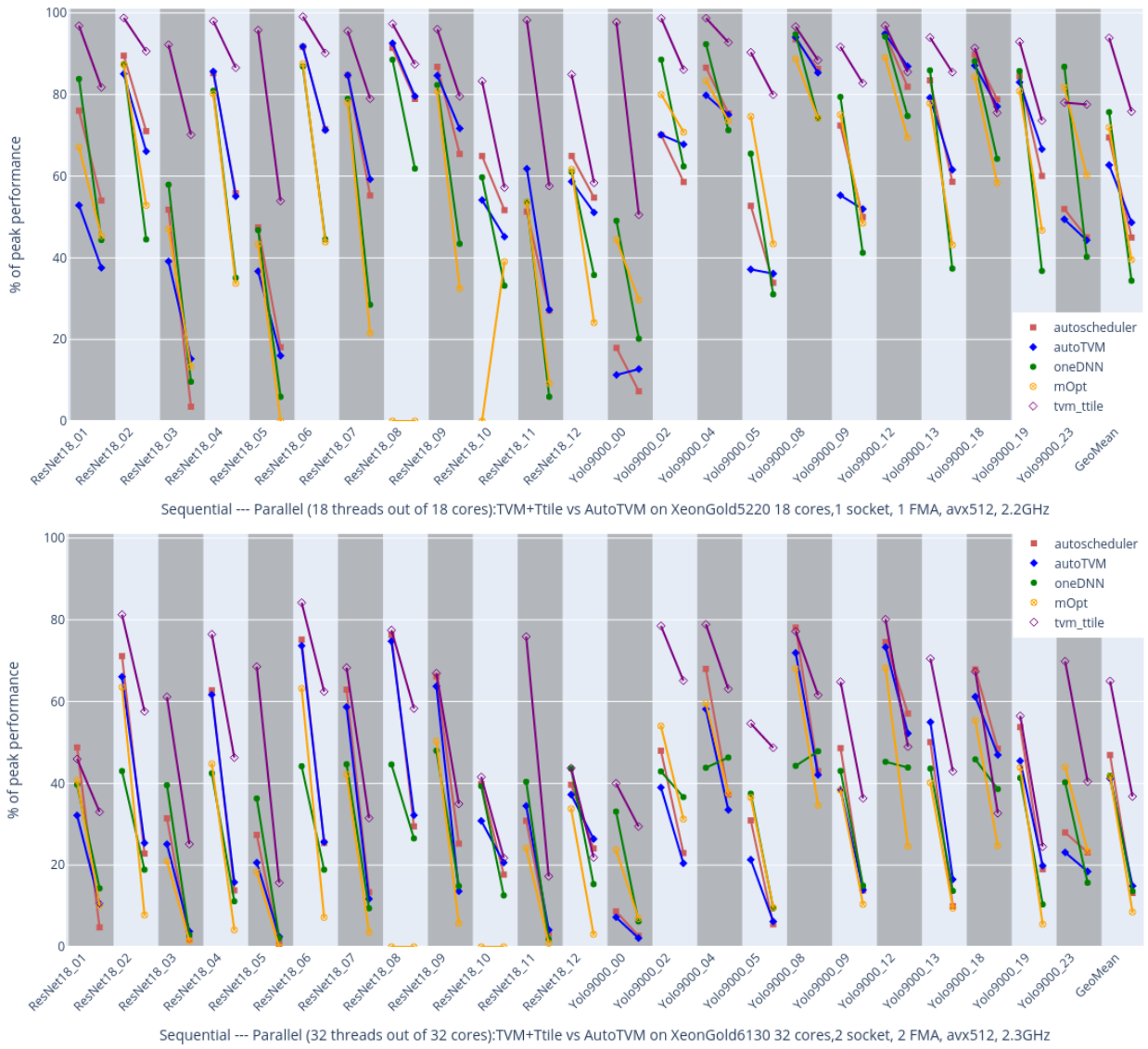


Figure 9: Comparison with AutoTVM, AutoScheduler, oneDNN, Mopt for AVX512 (Intel Xeon Gold 5220 and 6130), shown as percentage of machine peak. The left side of each line is the sequential performance and the right side is the parallel performance for the same convolution sizes. The last entry is the geometric mean for each tool, for the sequential and parallel case.

Benchmark	Microkernel
Yolo9000-0	[8H2K]
Yolo9000-2	[3S4H4K]
Yolo9000-4	[3R10H]+2[3R12H]
Yolo9000-5	[8H2K]+2[13H2K]
Yolo9000-8	[3S4H4K]
Yolo9000-9	[8H2K]+5[12H2K]
Yolo9000-12	2[3R11H]+[3R12H]
Yolo9000-13	2[10H2K]+[14H2K]
Yolo9000-18	[8H2K]+[9H2K]
Yolo9000-19	[8H2K]+[9H2K]
Yolo9000-23	[8H3K]+[9H3K]

Benchmark	Microkernel
ResNet18-1*	[14H2K]
ResNet18-2	[14H2K]
ResNet18-3	[14H2K]
ResNet18-4*	[14H2K]
ResNet18-5*	[14H2K]
ResNet18-6	[3S4H4K]
ResNet18-7*	[14H2K]
ResNet18-8	[14H2K]
ResNet18-9	[3R14H]
ResNet18-10*	[3R7H4K]
ResNet18-11*	[7H4K]
ResNet18-12	[3S7H4K]

Figure 10: Microkernels of the best parallel scheme found by TTile on an Intel Xeon Gold 6130. The corresponding unrolling factors and split factors of (combined) microkernels are reported with a compact notation where "2[3R11H]+[3R12H]" stands for the combination of 2 microkernels: the first is used twice as much as the second; "[3R11H]" stands for a microkernel where dimension r is unrolled 3 times and h is unrolled 11 times.

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.

The OS is Debian GNU/Linux with kernel version 4.19, monitoring hardware counters using PAPI version 5.7.0. All codes were compiled using gcc with the flags `-O3 -march=native -fno-align-loops`. For AutoTVM and AutoScheduler, we used the recommended template `conv2d_ NCHWc` from the TVM library. AutoTVM's machine learning model ran over 1000 trials in order to find its best configuration.

We evaluate performance over the convolutions of two networks: Yolo-9000 [27] and ResNet-18 [18].The sizes of their convolution layers can be found in Fig. 8. To measure performance with AutoTVM and TVM+TTile, we used the TVM function `evaluator`, with parameters `repeat` and `number` set to 1. Runs of oneDNN use the input tensor layout *nchw*. Every convolution is run 20 times on 20 different copies of the tensors (to avoid reusing data in the last-level cache for repeated experiments). We eliminated the 5 best and the 5 worst results and take the median of the 10 remaining results. We also consider a cold cache and we flush it between every run.

Performance measurement Fig. 9 presents the performance results, reported as a fraction of the peak performance of the multicore CPU. A few MOpt results are missing due to reproducibility issues with the author's artifact. We reported such situations with a performance at 0% of the machine peak. We also report the geometric mean for each tool and in the sequential/parallel cases, across all the available benchmarks. We notice that TVM+Ttile is well above the other state-of-the-art tools in average, which demonstrates the viability of our approach. The performance results for oneDNN seem surprisingly low, despite our efforts to explore the relevant configuration settings for oneDNN. However, we have confirmed with the developers that these performance were coherent with their expectation.

In term of total compilation time to find the best implementation found, over all benchmarks, AutoTVM and AutoScheduler both take around 5h. MOpt takes around 1h30 and TTile takes around

2h. OneDNN is performing JIT compilation thus does not have an offline exploration phase.

Microkernel variations We report in Fig. 10 the microkernel used by our best performing schemes. We notice a variety of 16 different microkernels used for 23 different convolutions. For example, the size of the microkernel picked for ResNet18-1 is "$[14H2K]$", i.e., 14 unrolled iterations along the h dimension and 2 iterations along the k dimension. In comparison, Yolo9000-4 uses a combination of two microkernels "$[3R10H] + 2[3R12H]$". Both microkernels are of size 3 along the r dimension, but differ in the size along the h dimension. The second microkernel is also used twice as often than the first microkernel. As another example illustrating this result, the best scheme found for Yolo9000-5 was introduced as the example of Sec. 2.

Interpretation We can draw several observations from these experiments. First, the divisibility constraint is the key restriction of our optimization space, that we have slightly relaxed with the possibility to combine microkernels. Despite this restricted, we still manage to find an implementation with excellent performance. This confirms that the divisibility constraint is effective at aggressively pruning implementations (including good ones) while preserving enough of the best-performing ones.

Also, the diversity of the microkernels available and their combination is the key point that differentiates our optimization space with those of the state-of-the-art tools. The fact (i) that we managed to find better implementations, sometimes by a factor of 2, and (ii) the variability of microkernels used by the best implementation we found, support the claims made in Sec. 3. In order to reach the best performance, we should not restrict ourselves to a single microkernel. Also, combining microkernels happens to outperform competing alternatives such as partial tiles (limited or no unrolling when hitting tensor boundaries) or padding.

As a final observation, copying and packing the footprint of register tiles in microkernels is generally assumed to be essential to performance (on both tensor contractions and convolutions) [30, 19]. Our results challenge this assumption by obtaining an efficient implementation without considering this transformation in our search space. This shows that this transformation should not be applied systematically for all convolution kernels. We plan to study when the packing transformation should be applied in our future work.

8 Related work

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 [16, 6, 32, 31, 11, 2] that leverage parametric integer linear programming. Although such approaches are well suited to expose parallelism [12, 13] and coarse grain locality [6], 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 polyhedra [3] allows to automatically generate (non-linear) cost models. We use the Barvinok [3] library for this purpose, generalizing the approach of Li et al. [21] for the selection of a permutation scheme.

Cloog [4] 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 allows to generate high-quality compiler friendly code without heroic efforts [17].

Optimization of machine learning programs There exist many compilers specialized for machine learning: PlaidML [7] using polyhedral techniques, XLA [14] for TensorFlow [1], Halide [26], or TVM [8]. 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 [35] and Ansor (a.k.a AutoScheduler) [34]. We also compare to AutoTVM, the auto optimiser of TVM. Telamon [5] 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.

Linear algebra and CNN libraries Frameworks such as TBLIS [23] or TCCG [29] 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 [15]. 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 [33] and oneDNN [19] are the best available Intel libraries which implement all these techniques.

9 Conclusion

We presented TTile, a tool which generates high-performance code for tensor computation, and we evaluated it on static-shape 2D convolutions. The generated code exploits a large set of performant microkernels and combines them in order to avoid partial tiles. The methodology itself is a combination of different kind of state-of-the-art methods, including analytical modeling, autotuning and experimentation. We present performance results that demonstrate higher performance than oneDNN in the vast majority of cases, and consistently higher performance than the state-of-the-art TVM tensor compiler and autotuner and MOpt. We plan to release publicly a version of our tool in the next few months.

Future work includes extension of the search space to include additional transformations such as packing or prefetching, in order to carefully study their impact on the performance and determine when they are needed. We also plan to generalize the method to a wider set of tensor operators and layout configurations, using MLIR [20]. This will also facilitate the exploration of interactions with graph-level optimizations, leveraging optimization opportunities currently accessible only to JIT compilers such as XLA.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale

machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI'16)*, pages 265–283, USA, 2016. USENIX Association.

[2] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, (CGO 2019)*, pages 193–205. IEEE, 2019.

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

[4] Cedric Bastoul. 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, page 7–16. IEEE Computer Society, 2004.

[5] Ulysse Beaugnon, Antoine Pouille, Marc Pouzet, Jacques Pienaar, and Albert Cohen. Optimization space pruning without regrets. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, page 34–44, New York, NY, USA, 2017. Association for Computing Machinery.

[6] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 101–113, New York, NY, USA, June 2008. Association for Computing Machinery.

[7] Huili Chen, Rosario Cammarota, Felipe Valencia, and Francesco Regazzoni. 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*, pages 333–336. IEEE, 2019.

[8] 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. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594. USENIX Association, October 2018.

[9] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31:3389–3400, 2018.

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

[11] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. 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*, page 42–51. ACM, 2018.

[12] Paul Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[13] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[14] Google. Xla : optimiser le compilateur pour le machine learning.

[15] Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 blas. *ACM Transactions on Mathematical Software*, 35(1), 2008.

[16] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letter*, 22(4), 2012.

[17] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral ast generation is more than scanning polyhedra. *ACM Transactions on Programming Languages Systems*, 37(4), July 2015.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 06 2016.

[19] Intel. oneAPI deep neural network library (oneDNN). `https://01.org/`, 2018.

[20] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, Los Alamitos, CA, USA, mar 2021. IEEE Computer Society.

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

[22] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P. Sadayappan. 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*, pages 928–942, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Devin A. Matthews. High-performance tensor contraction without transposition. *SIAM Journal on Scientific Computing*, 40(1):C1–C24, 2018.

[24] NVIDIA. Cudnn: Gpu accelerated deep learning. `https://developer.nvidia.com/cudnn`, 2018.

[25] Auguste Olivry, Guillaume Iooss, Nicolas Tollenaere, Atanas Rountev, P. Sadayappan, and Fabrice Rastello. IOOpt: Automatic derivation of I/O complexity bounds for affine programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1187–1202, New York, NY, USA, 2021. Association for Computing Machinery.

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

[27] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 6517–6525. IEEE Computer Society, July 2017.

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

[29] Paul Springer and Paolo Bientinesi. Design of a high-performance GEMM-like Tensor-Tensor Multiplication, 2016.

[30] Field G. Van Zee and Robert A. van de Geijn. Blis: A framework for rapidly instantiating blas functionality. *ACM Transactions on Mathematical Software*, 41(3), June 2015.

[31] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions, 2018.

[32] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization*, 9(4), January 2013.

[33] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. *Intel Math Kernel Library*, pages 167–188. Intel, 05 2014.

[34] 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. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879. USENIX Association, November 2020.

[35] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems*, pages 859–873. ACM, 03 2020.