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*

Abstract

Tiling is a key loop transformation for optimizing tensor computations such as CNNs (Convolutional Neural Networks). Tile optimization involves an explosively large search space for multi-level tiling, including all possible permutations of the tiling loops and all possible valid tile sizes. In this paper, we develop a comprehensive methodology for finding optimized tile configurations with imperfectly nested micro-kernels ("beyond perfect") and outer tile loops optimized via analytical modeling. Experimental results on over 30 CNN benchmarks from three popular DNN pipelines demonstrate the effectiveness of the presented optimization approach by comparing with the Intel oneDNN library.

1 Introduction

Data locality optimization is critical for high performance. Tiling is among the most critical loop transformations for data locality optimization of loop-based programs. Although tiling has been extensively studied in the compiler research community, effective tiled code generation for high performance remains a challenging problem. A fundamental problem is that the design space of possible multi-level tiled configurations is huge. Consider the code for a 2D convolution operator in machine learning. As detailed later in the paper, it is a 7-dimensional loop nest. There are 7! possible permutations for the loop nest. Tiling for 3 levels of cache results in four bands of loop-nests, one per level of cache and an outermost band. For each of the tiling loops, many possible choices for tile-size exist.

Tensor computations are at the core of many applications in scientific computing, data analytics and machine learning. The optimized implementation of tensor computations 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 [13], Polly [18], Pluto [8], PPCG [29], Tensor Comprehensions, [28], Tiramisu [4] can automatically generate multi-level tiled code for any affine loop computation such as 2D convolutions. However, a significant limitation of polyhedral compilers is that none of them can directly optimize across tile sizes.
- Vendor libraries like oneDNN [1] and cuDNN [23] provide implementations that have been manually optimized by expert software developers. While these implementations use JIT optimization, they cannot customize tile configurations in a fully adaptive manner depending on the tensor extents of different CNN stages in a DNN pipeline.

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

- Auto Tuning can be performed by systems like AutoTVM [10]. A 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 [11] iterates through a number of tiled loop configurations, where code is generated and compiled, followed by execution on the target platform. The code generated by AutoTVM has been demonstrated to achieve higher performance than that generated by polyhedral compilers [10], but still below reference libraries like oneDNN.
- Manual/semi-automatic analytical modeling/optimization. Recent research has shown that a comprehensive characterization and optimization across the space of all possible tiled loop configurations for CNNs is feasible [21]. A manual generation of analytical cost models for data movement and pruning by manual reasoning was used in conjunction with solution of nonlinear constrained optimization problems to optimize tile sizes. For the innermost loops, a manually created *micro-kernel* was used, similar to the BLIS micro-kernel [27]. Over a set of 30+ CNN stages from 3 DNN pipelines, performance was shown to be consistently higher than state-of-the-art autotuning AutoTVM [10] and comparable or better than the state-of-the-art oneDNN library [1].

These approaches to optimize core tensor computations differ along two key attributes: (i) performance, and (ii) productivity. Optimization approaches that are fully automated, e.g., polyhedral compilers, offer the highest productivity benefits for developers, but currently achievable performance is lower than other approaches. Libraries like oneDNN [1] and the manually developed optimization scheme of Li et al. [21] achieve the highest performance but require considerable manual effort. AutoTVM achieves lower performance than oneDNN [1] or the code from the optimization scheme of Li et al. [21] but requires much less manual effort – a good script to constrain the search space for the autotuning. In this paper, we develop an approach to optimize tensor computations that achieves both maximum user productivity via complete automation, as well as maximum performance. Figure 1 summarizes previously developed approaches and the new approach (TTile) presented in this paper.

Approach	Automation	Performance
Polyhedral Compilers	High	Medium
Library (e.g.oneDNN)	Low	Very High
Autotuning (e.g., TVM)	Medium-High	High
Li et al. [21]	Medium	Very High
TTile (ours)	High	Very High

Figure 1: Comparison of degree of automation versus performance for alternative approaches to optimizing CNN

Comparing the prior approaches that do well on both productivity (degree of automation in generating efficient code) and performance:

- AutoTVM uses heuristic search guided by a machine learning model where the design space is the entire set of tile sizes for the multi-level tiled loop nest and LLVM is used to compile the loop nest.
- Li et al. use a fully analytical approach by formulating and solving a nonlinear optimization problem to find the set of tile sizes that minimize data movement overheads. But unlike the TVM approach, they do not optimize for all tile sizes but use a fixed manually pre-designed micro-kernel for the innermost loops, similar to the BLIS approach [27], implemented using vector intrinsics.

In this paper, we present an automated micro-kernel optimization and code-generation approach for tensor computations that improves on these prior approaches. While the developed approach applies to a

wider class of tensor computations (defined later in the paper), we focus on the important CNN operator in this paper. The key features of the presented approach and the contributions of this paper are as follows:

- Instead of a single fixed (manually developed) micro-kernel used by prior approaches [27, 21], a comprehensive search among the space of possible micro-kernel configurations is made to identify a set of micro-kernel instances as base building blocks for use in generating optimized code for specific CNN instances.
- For a given CNN instance and a given micro-kernel, automated analytical modeling is used to formulate and solve a constrained nonlinear optimization problem for the tile loops surrounding the micro-kernel.
- For a specific CNN instance, a combination of several micro-kernels can be considered using imperfectly nested loops - or "beyond perfect" tile loops.
- Experimental data from a set of 30+ CNN stages from three networks (ResNet, MobileNet, Yolo9000) demonstrate the superior single-core performance achieved by the presented approach to micro-kernel optimization over the state-of-the-art oneDNN library.

The rest of the paper is organized as follow: Sec. 2 recaps some necessary background. Sec. 3 motivates the design of the search space and Sec. 4 details the search strategies. Sec. 5 describes or code generator. Sec. 6 reports experimental comparisons against state of the art frameworks and libraries. Sec. 7 discusses related work before the conclusion in Sec. 8.

2 Background

Our framework need to consider several key concepts.

A *micro-kernel* (μ*kernel*) is an efficient portion of a code which corresponds to the inner-most loops that do not involve any data-movement between the different cache levels. Its efficiency is mostly dictated by the CPU characteristics it turns on. It is generally directly written in assembly, or using intrinsic instructions. In order to be efficient, we want our μkernel to have the following properties: (i) vectorization units should be used; (ii) the data are stored in (vector) registers and reused across iterations (avoid extensive use of register spilling); (iii) there is enough parallelism between the instructions (hide pipeline latency of multiply-add). We target super-scalar architectures and expect out-of-order mechanism to exploit exposed instruction level parallelism.

An *iteration space* is the set of integer values taken by the loop indices surrounding a given statement. A *tiling* [25, 12] is a loop transformation that partitions the iteration space into sets, called *tiles*, such that each tiles are executed atomically between each other. In this paper, we will only consider programs that have *rectangular* iteration space, and rectangular tiling. The tiled code has additional loops compared to the original code: over the tiles, and inside a tile. This division of the computation allows us to control the amount of data usage per tile (footprint), such that the footprint does not exceed a given memory capacity. We will consider a hierarchy of tiling which will allow us to fit the data locally used in the smaller and faster memories.

3 Optimization search space

Let us first introduce the class of loop nests we consider then build a search space of optimization strategies for this class.

```
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_t)
```

Figure 2: Tiled matrix multiplication with μkernel.

3.1 Kernel specification

Figure 2 shows a tiled matrix multiplication kernel as an illustrative example. It relies on an (inline) fully-unrolled and vectorized μkernel of size 6×32.

Dimensions and iteration space We only consider rect-angular-shaped iteration spaces. By convention, we use lowercase to name dimensions (i, j, k) and uppercase to name the (possibly symbolic) upper bound on this dimension (respectively I, J, K). We also assume that the considered dimensions are either (i) parallel (ex: i and j) or (ii) a reduction (ex: k). While associativity can be used to parallelize a reduction, we do not exploit it.

Tensors Tensors are rectangular-shaped multidimensional arrays that are operated upon. In a given program statement, we assume every tensor may occur once or more but always with the same subscript, which is an *affine function* of the iteration space to the tensor's rectangular shape. For example, tensor A of shape $\{i, k | 0 \leq i < I, 0 \leq k < K\}$ may be subscripted by $[i, k]$, corresponding to the access function $(i, j, k \mapsto i, k)$. We also assume that a index cannot appear twice inside an access function: for example $E[i, i]$ is forbidden. These assumptions apply to tensor contractions and convolutions, including all strided variants. A mapping function from a tensor definition domain to a memory location is used during the last phase of the code generation to linearize the tensors. For example, $C[i, j]$ will be replaced by `C[J*`i`+`j`]`.

Class of kernels We consider perfectly nested affine loop programs such that:

- The iteration domain is rectangular and is fully permutable.
- All the occurrences of a tensor in a statement have the same access function.
- An index cannot appear twice in an access function of a tensor.

3.2 Optimization strategy

Classical high-performance libraries, such as BLIS, TCCG, oneDNN, rely on a fixed optimization pattern. This pattern is based on a single hand-written μkernel selected from a tiny set, uses a fixed packing strategy and conditional execution or padding to manage partial tiles. In our case, we consider a different space of strategies: we allow combination of micro-kernels, and we forbid partial tiles. This section define this class of optimization strategy we will consider.

The *optimization scheme* is a list of *specifiers* that describe the layered structure of the generated code, *from the outermost loop inwards*. A specifier can be either:

- R_d, to insert the outer loop along dimension d. Assuming a tiling along d, this loop will iterate over the outer-level tiles along d (it will step by 1 otherwise). As we will see, our scheme does not consider partial tiles: outer-level tiles sizes along d should divide D. Besides, R_d may appear at most once for a given dimension d, and must be the first (leftmost) specifier involving this dimension.
- $\mathsf{T}_{k,d}$, to insert a tile loop along dimension d. It iterates *exactly* k times along d on the next-level tiles (if any, and it will step by 1 otherwise). Again, there is no partial tile and the size of the iteration space along d covered by one full execution of this loop should be a multiple of k.
- $\mathsf{U}_{k,d}$, is semantically equivalent to insert a tile loop with $\mathsf{T}_{k,d}$ and fully unroll it (register tile). Divisibility constraint still hold: partial register-tiles or partial vectors are not allowed.
- V_d, is semantically equivalent to insert a tile loop with $\mathsf{T}_{v,d}$ (with v the vector length on the targeted architecture) and vectorize it. Vectorization occurs at the innermost level only: there may be at most one $\mathsf{V}_\bullet$, which must be at the end of the list.
- $\lambda_{\mathsf{seq}_d}\alpha.\,[\ell]$, where $\ell = [(r_i, a_i)]_{1\leq i<s}$ is a list of $s \geq 2$ pairs, introduces a sequence of s loops of size r_i along dimension d. Each of the s loop iterates over the next-level parametrized-by-α tiles. For loop nest $1 \leq i < s$, parameter α (that can be used in a specifier as a placeholder for k) is set to $\alpha = a_i$. As we will see later, this specifier that leads to generating a non-perfectly nested loop, allows to use highly-optimized μkernels which size do not divide the problem size: Splitting a dimension (e.g. $Y = 34$) into two non-equal parts (e.g. 22 and 12 with $\ell = [(2, 11); (1, 10)]$) allows to fulfill the divisibility constraint (no partial tiles) and use high-performance μkernels (here of size 11 and 10 along y).

Note that we do not define a packing specifier. Our experiments demonstrated that packing was not a performance-critical transformation for convolutions. Diverging from the default strategy in the optimization of matrix product and tensor contractions, packing should not be an automatic choice. This is good news, as it helps simplifying both code generation and the search for an efficient optimization strategy. Of course, we will have to extend our framework to enable packing again when broadening the applications of our optimization algorithm beyond convolutions.

Example For example, the naive implementation of a matrix multiplication would be: $[\mathsf{R}_i, \mathsf{R}_j, \mathsf{R}_k]$. A slightly less naive implementation of a matrix multiplication based on the BLIS [27] μkernel 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 μkernel of size ($i = 6, j = 16, k = n_k$) known to be quite efficient as it requires only 15 vectors (see [27]) and exposes enough instruction-level parallelism (12 multiply-add can be done independently between two accumulation steps). Around it, a loop along i creates a tile of size m_c, and another along j a surrounding tile of size n_c. As one can observe, and as explained later in this section, this code needs to assume 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 μkernels and tuned tiles sizes, and thus introduce partial *non-optimized* tiles to cope with arbitrary problem size that do not fulfill the 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:

```
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}
}
```

4 Exploration of the optimization space

While we have been able to define an optimization space for a general class of tensor operations, building an effective exploration strategy involves additional domain knowledge and properties. From now on, we focus on the important case of convolution kernels with static shape. Figure 3 shows a template for a 2D convolution.

We propose a novel optimization algorithm capable of exploring our very expressive optimization space. The optimization algorithm is divided into 4 steps, building on analytical modeling, empirical evidence, control overhead mitigation and code size constraints. It is summarized by Figure 4:

1. First, we measure the performance of many variations of μkernels in isolation. *The set of μkernel candidates* are the one that are performing best (Section 4.1). This phase is problem size agnostic but is specific to each targeted architecture.

2. Then, for each μkernel candidate, accounting for the problem size and cache sizes, we determine the *best loop permutation* (list of loop dimensions) enclosing the μkernel (Section 4.2), that is, the main structure of the loop nest is determined (nesting of tiles along which dimensions), but not the actual tiles sizes. This permutation is obtained through operational research by using an analytical model of the footprint, data movement and reuse across tiles. Similar size μkernels are grouped into classes at this step as they yield the same permutation in the analytical model. The output permutation is also decomposed into 4 parts, the L1-, L2-, L3-fitting loops, and the remaining loops whose footprint does not fit the last level cache.

3. From the selected μkernels, we generate the *space of compatible optimization schemes*. For a μkernel (or a combination of – see below), we complete the associated permutation with specific tile sizes and unroll factors (Section 4.3). The main compatibility challenge, that is the so-called *divisibility constraint*, stems from our choice to *never generate any partial tile* (associated with control flow overhead, code duplication or padding overhead). We thus 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. In particular, a μkernel which size does not divide the problem size shall not lead to any compatible scheme. In practice, problem size might not be friendly with the size of the high-performance selected micro-benchmarks. As an example, Yolo9000-8 has a problem size of $H = 17$, but while in particular some of the micro-benchmarks of size $H \in \{5, 6\}$ have good performance, all of the micro-benchmarks of size $H = 17$ have quite poor performance. To address this problem, we also consider, thanks to the λ_{Tseq} specifier, combinations of μkernels: two candidate μkernels can be combined if their size differs only on one dimension. For our Yolo9000-8 for example, μkernels of size $\{S = 3, H = 5, K = 4\}$ and $\{S = 3, H = 6, K = 4\}$ can be combined to fulfill the divisibility constraint without compromising the performance.

```
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 3: Convolution (batch size $N = 1$).

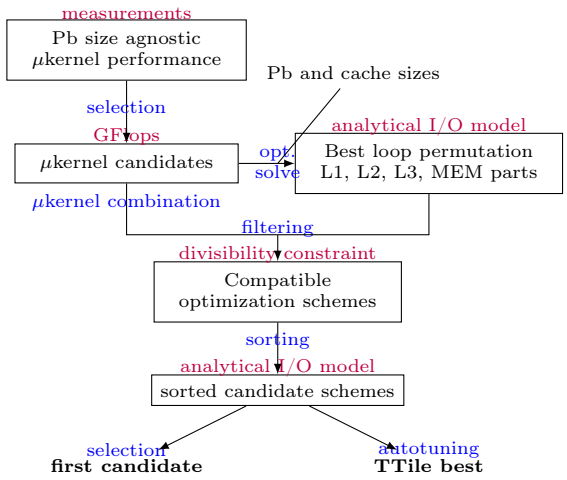

Figure 4: Flow of the optimization strategy selection.

4. Thanks to our decoupled approach and because of the strong imposed constraint of divisibility, the size of the obtained space of viable schemes, ranges from a few hundreds to several thousands. While, this is still acceptable for ahead-of-time exploration (a few hours for one problem), we can do better. A simple metric detailed in Section 4.4 can be used to sort all those schemes allowing to either remove the need for any autotuning or allowing JIT selection.

4.1 μkernel performance evaluation

We identified 4 optimization schemes to build the space of all possible μkernels. The optimization schemes selection is driven by CPU architecture and CNN structure: the code should contain vectorized `load` and `store` instruction to maximize the performance throughput. The dimension k is selected to be vectorized because it contains the simplest access patterns among all w, h and k.

We evaluate the performance in isolation of the variations of convolution shapes and context. The code of these μkernels are generated automatically using TTile, from these optimization schemes:

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

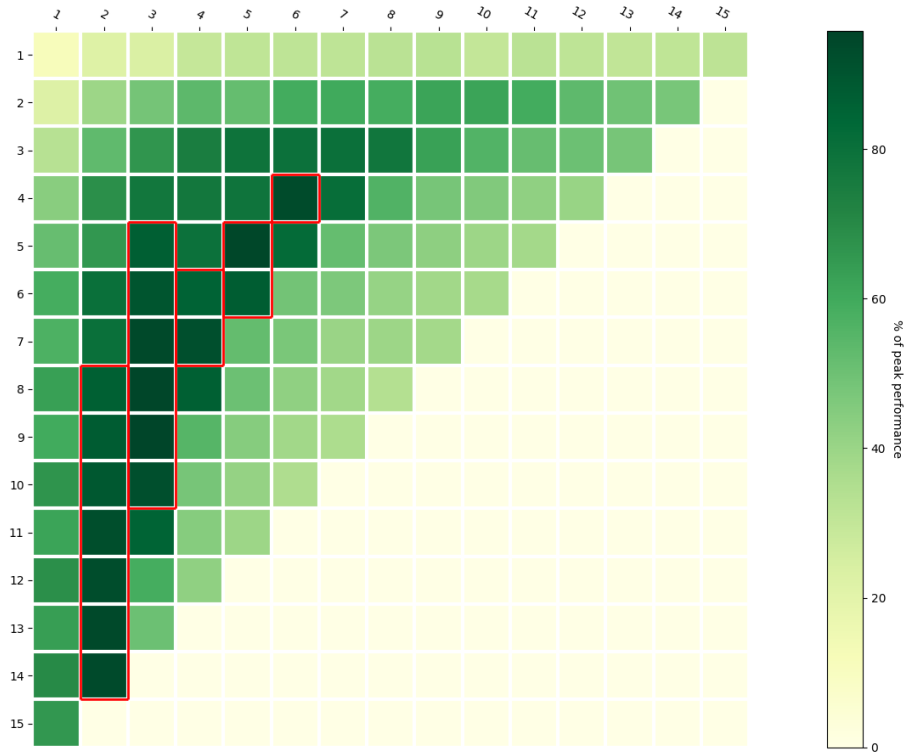

Figure 5: Performance of μkernels in isolation for AVX512 (Intel i9-7940X Skylake-X), in percentage of the machine peak. Here R and S are fixed to 1. We tested all combination of R/S in 1,3 but report the results only for 1,1 here for space consideration. The sizes of the μkernels (α along the K dimension (horizontal axis) and β along the H dimension (vertical axis)) vary between 1 and 15. Only the upper-left triangle was evaluated. In red are the selected class of best μkernels that will be used by our optimization strategy selection.

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

where α and β are the sizes of the μkernel along the k and the h dimension, respectively. The r and s dimensions are small dimensions whose value is often 1 or 3. Thus, we do not need to explore it extensively.

In order to evaluate the performance of one of these μkernel, we repeat it along the c dimension ($\mathsf{T}_{512,c}$), and we run its code on a problem size equal to the footprint of the μkernel optimization strategy. The results on AVX512 are shown in Figure 5. For these experiments, the frequency have been fixed at 3.1 GHz, the OS is Arch Linux (kernel version 5.10.10), and hardware counters have been monitored using PAPI version 6.0.0.1.

We observe that the performance graph is roughly convex with some local fluctuations. Also, placing

both dimensions r and s in the μkernel gives at most 70% of the peak performance, which is much less than the 95% we can potentially obtained for the other patterns. So, these μkernels should be avoided.

Class of candidate μkernels Many μkernels are near the peak performance. We select the ones which are above 85% and classify them using the following ad-hoc strategy: The μkernels of a given class have the same characteristics (same size on C, H, S, R), except for their size on dimensions H that belongs to an interval. For example, $\{[\mathsf{U}_{\beta,h}, \mathsf{U}_{2,k}, \mathsf{V}_k], 8 \leq \beta < 15\}$ is one of the class of μkernels that is selected for AVX-512, as shown with the leftmost red vertical rectangular contour on Figure 5.

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

4.2 Loop permutation above the μkernel

Given the convolution size and a μkernel, we compute a suitable permutation of the loop above the μkernel that minimizes the amount of data movement needed by the caches and in particular the L1 cache. This permutation is divided into parts, each part corresponding to a cache level.

For example, for the Yolo9000-0 benchmark, the output on the μkernel $[\mathsf{U}_{8,h}, \mathsf{U}_{2,k}, \mathsf{V}_k]$ is (from outer to inner):

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

The first list are the loops above the L3 cache, the second list are the loops above the L2 while being L3-resident, the third list is the loop above the L1 and L2-resident and the fourth list are the loops above the μkernel and L1-resident. Each dimension appears at most once in each list.

The permutation is found by running a separate tool, whose functioning is detailed in a paper currently under review. The approach is similar to that of Li et al. [21], but generalized to the larger class of affine programs [16]. We will only summarize the main ideas behind it.

For a given permutation, the tool is able to derive an analytical expression of the data movement volume at each memory 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. A pre-processing step avoids trying all possible loop permutations (here $(7!)^3$): many of them are equivalent in term of data movement cost, or provably worse. At each memory level, the number of considered permutations drops from 7! to only 6.

Then, for given value of array sizes, this analytical model is fed to a non-linear problem optimizer which finds (potentially fractional) tile loop sizes for each dimension at each level that minimize overall data movement. Since actual tile size selection is a very complex problem (with divisibility constraints, partial tiles, cache replacement policies...), what is retained in the output is only the loop permutation, i.e. the order of loops that are non-degenerate at each tiling level.

Low level architecture details such as vector units and register capacity are not part of the analytical model, so the μkernel configuration is part of the input.

In order to avoid performing this analysis for every μkernel for a convolution size, we assume that the loop permutation stays the same for any μkernel inside the same class.

4.3 Space of valid optimization schemes

One of the main assets of our technique, enforced in the optimization scheme, is that we forbid partial tiles. This means that, the size of a rectangular tile along a dimension must be the divisor of the problem size along this dimension, and a multiple of the size of a potential tile below it on the same dimension.

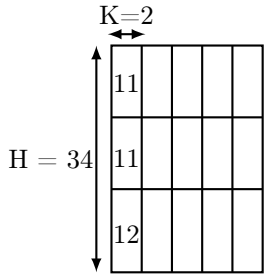

Figure 6: Example of coverage of a problem size by using a combination of two μkernels: $34 = 2 \times 11 + 12$.

However, this constraint is difficult to satisfy for some problem sizes. We avoid this problem by using two μkernels of different sizes, and by combining them sequentially in order to get back to a divisor of the problem size. Figure 6 shows a graphical example of such a combination.

Combining μkernels sequentially In order to juxtapose legally two μkernels, all their sizes must be the same, except for one dimension. In our case, we consider the combination of μkernels inside the same class, thus, in practice, the h dimension is generally the one that varies.

For all pair of μkernel in the same class, of sizes h_1 and h_2 along the changing dimension, and given a problem size H, we try to find a number of repetitions a and b of these μkernels such that: $(a \times h_1 + b \times h_2)$ divides H.

For example, if $H = 34$ and we are considering two μkernels of the same class of size $h_1 = 11$ and $h_2 = 12$, then we can combine two μkernels of size 11 followed by a μkernel of size 12 in order to reach 34.

If we cannot reach a divisor of the problem size, then we consider that it is not possible to tile using the considered pair of μkernels, and we continue on the next candidates.

When we manage to find a valid combination, we need to complete it into a full optimization scheme.

Completing a μkernel into a full scheme Given (i) a single μkernel that divides the problem sizes, or a μkernel combination that divides the problem sizes, and (ii) a loop permutation, we combine them into an optimization scheme. The algorithm proceeds as follows:

1. We place the chosen μkernel, or the pattern of the combination of μkernels on the right of the optimization strategy (innermost loops).

2. For every dimension, we consider the divisors of the problem size divided by the μkernel size. These divisors needs to be allocated to the different strip-minded occurrences of the dimension across the whole permutation. There are many solutions and we consider 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.

3. For each element d in the loop permutation, we consider the product π of the divisors allocated to this occurrence and we build the corresponding specifier and we place a $\mathsf{T}_{\pi,d}$. These specifiers arranged in the same order of the given loop permutation form the strategy scheme above the μkernel.

4. If we consider a sequence of two μkernels at dimension d with the combination $a \times h_1 + b \times h_2$, we can place the $\lambda_{\mathsf{seq}_d}\alpha.\,[l]$ at any occurrence of dimension d in the loop permutation, and consider all variations. The value of the list ℓ is $[(a, h_1), (b, h_2)]$.

Example Let us consider the μkernel class

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

for AVX512, and let us consider Yolo9000-13 problem sizes $(K, C, H/W,\ R/S) = (512, 256, 34, 3)$. Let us assume that the corresponding loop permutation found was

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

The problem size on the dimension H is $34 = 2 \times 17$, thus there is no single μkernel from the considered class that matches one of its divisor. Thus, we consider combinations of 2 μkernels from that class. We can either consider combination equals to 17 or to 34. Let us consider the later case, and let us consider $2 \times 11 + 1 \times 12$ as one of these combination.

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

- The k dimension is trivial: there are only one loop above the μkernel and the μkernel size along this dimension already has a footprint of 32. Thus, we need to tile by a factor of 16 on the outer loop.
- 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 (3 for both).
- The h dimension is already managed by the combination of μkernels. We have 3 locations where we can place the $\lambda_{\mathsf{seq}_h}\beta.\,[(2, 11); (1, 12)]$ corresponding to the switch between the two μkernels. Let us consider the outer one.
- The w dimension has 34 to be distributed across 2 level of tilings. There are 4 combinations: 34×1, 17×2, 2×17 and 1×34. Let us consider the second one.

Thus, one of the optimization scheme that we generate is:

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

4.4 Pruning the optimization space

The optimization space is large and, even if spending several hours or days per convolution instance to evaluate all schemes can be acceptable, it turns out that one can drastically reduce the search space (up to not having to rely on any tuning at all) without trading too much performance. We propose the following metric, which allows us to focus on an area where there are at least one of the best performing scheme, based on observations made on Figure 7:

- First, maximize the reduction size, i.e. the size of the tiling on c just above the μkernel.
- Then, select those close to 120% of occupancy of the L1 cache.

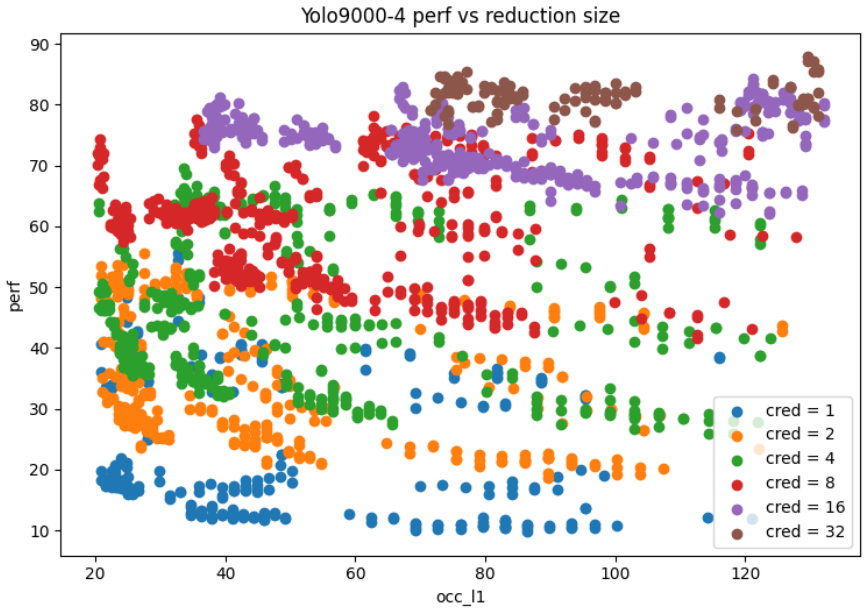

Figure 7: Exhaustive exploration of the optimisation space for Yolo9000-4, and classification according to the tile size of the reduction loop c above the μkernel and L1 cache occupancy.

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

Benchmark	Problem sizes (K, C, H/W, R/S)
MobileNet-1	32, 32, 112, 3
MobileNet-2*	64, 64, 112, 3
MobileNet-3	128, 128, 56, 3
MobileNet-4*	128, 128, 56, 3
MobileNet-5	256, 256, 28, 3
MobileNet-6*	256, 256, 28, 3
MobileNet-7	512, 512, 14, 3
MobileNet-8*	512, 512, 14, 3
MobileNet-9	1024, 1024, 7, 3

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

The cache occupancy can be easily evaluated from the scheme, because the loop permutation tells us at what level the L1 cache should be saturated.

We evaluate all the schemes of our set using this metric and select the 50 best ones. Then, we generate code (see Section 5) before measuring performance. The final results are shown in Section 6.

5 Code generator (TTile)

Let us now describe how to generate a C code from a kernel specification, problem size and the associated optimization scheme.

5.1 Sub-scheme and associated size

As illustrated in the example of Section 3.2, generating a loop requires to know the size of the sub-tiles. For this purpose, 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 (parametrized) sub-iteration space.

Taking the example from Section 3.2, the sub-scheme of the BLIS μkernel (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.

5.2 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:

- the size of the loops that are already generated;
- for every dimension, the name of the last index used by a loop (to handle tiling).

Before applying our code generation algorithm, we apply a preprocessing step to get rid of the λ_{Tseq} specifier and its parameter α. We introduce a new specifier Seq that corresponds to the sequential composition of a list of strategies. For our considered optimization strategies, the list of the λ_{Tseq} specifier is always of size 2. The corresponding rewriting rule would be:

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

where S is the sub-scheme following the λ_{Tseq} 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).

5.3 Code generation rules

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

Sequence Seq We combine sequentially the generated code corresponding to the sub-schemes inside the Seq specifier.

Vectorization $\mathbf{V}_d$ We consider the SSA graph of the computation described in Section 3.1. We determine which operations should be vectorized by propagating the vectorization in this 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 operand (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 we raise an error.

The C code uses Intel intrinsics to manipulate vectors.

Unroll $\mathsf{U}_{k,d}$ We 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 We 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. If R_d is used, we can deduce the correct number of iterations by comparing the footprint of the sub-scheme with the problem sizes. This changes the current loop index in use over the d dimension.

6 Performance results

We target an Intel i9-7940X Skylake-X using AVX512. The frequency have been fixed at 3.1 GHz, the OS is Arch Linux (kernel version 5.10.10), and hardware counters have been monitored using PAPI version 6.0.0.1.

We evaluate the generated code for the scheme selected by our optimization algorithm, and compare performance with Intel oneDNN. Figure 9 reports single-thread performance for all the benchmarks in Figure 8. Performance results are the median of 100 executions. The whole process, using TTile to find the 50 best permutation for each benchmark, according to our metric 4.4, and running all of them to get the performance results, takes about 15 mins. This is much faster than state-of-the-art feedback-directed autotuning approaches.

We use the following layouts, from outer to inner dimensions: for the output tensor, W, H, K; for the input tensor, $W + S, H + R, C$; and for parameters, S, R, C, K. These layouts are widely used on CPU. They have been chosen to facilitate vectorization along dimension K, which is a parallel dimension and has the desirable property of being a big power of two in every benchmark—except for yolo9000_23. Thus K should be the inner dimension when it subscripts a tensor. OneDNN has the ability to reorder dimensions, changing the layout for optimal performance. Thus, we report both end-to-end performance results including the cost of oneDNN's layout transformations, and convolution-only results for oneDNN performance evaluation. This potentially allows oneDNN to start with a more elaborated layout than our basic one, for example one with interleaved dimensions (instead of the default layout which is linearized) that matches the accesses pattern of the convolution loop nest. The results on figure 9 highlight that the code we generate remains competitive and outperforms oneDNN in the majority of the cases (17 times out of 32 convolutions).

We also report the combination of μkernels used by the best performing scheme. The variety of configuration supports the argument that we should not restrict the code generation to a few μkernels but instead embrace the diversity of their candidate.

After ordering the list of configurations following our metric defined in section 4.4 (no execution needed), we report the performance of the "best" candidate configuration and the performance of the best one among the first 50 (out of several thousands). We also report the best out of an exhaustive search. This last result shows that our metric is good enough to quickly find the best configuration in many cases.

As an example, the best optimization strategy for Yolo9000-4 on AVX512 is: $[\lambda_{\mathsf{seq}_h}\alpha.\,[(1,12);(4,14)]\,;\mathsf{T}_{136,w};\mathsf{T}_{8,k};\mathsf{T}_{2,h};\,\mathsf{T}_{3,r};\mathsf{T}_{64,c};\mathsf{U}_{3,s};\mathsf{U}_{\alpha,h};\mathsf{V}_k]$.

Benchmark	oneDNN with reorder	oneDNN conv only	TTile (us) (first/50 best/exhaustive)	μkernel (50 best)
Yolo9000-0	25%	**61%**	27%/<u>31%</u>/42%	11[H11K2]+[H15K2]
Yolo9000-2	70%	**85%**	82%/**85%**/85%	9[H9K2]+5[H11K2]
Yolo9000-4	79%	86%	86%/**92%**/92%	[S3H12]+4[S3H14]
Yolo9000-5	47%	**83%**	66%/<u>76%</u>/76%	[H6K4]+4[H7K4]
Yolo9000-8	82%	**86%**	81%/81%/87%	[S3H10]+2[S3H12]
Yolo9000-9	66%	**88%**	77%/<u>86%</u>/86%	3[H8K2]+4[H11K2]
Yolo9000-12	82%	**86%**	37%/66%/74%	[H8K2]+2[H13K2]
Yolo9000-13	73%	85%	75%/**88%**/88%	[H10K2]+2[H12K2]
Yolo9000-18	68%	**78%**	16%/63%/65%	[S3H5K4]+2[S3H6K4]
Yolo9000-19	72%	**82%**	30%/39%/75%	[H8K2]+[H9K2]
Yolo9000-23	66%	**79%**	9%/56%/-	[H8K3]+[H9K3]
ResNet18-1*	68%	**83%**	39%/47%/-	[H7K4]
ResNet18-2	73%	82%	80%/**92%**/-	[S3H14]
ResNet18-3	44%	74%	60%/**75%**/-	7[H6K4]+2[H7K4]
ResNet18-4*	63%	71%	67%/**88%**/-	2[S3H9K2]+[S3H10K2]
ResNet18-5*	34%	76%	83%/**83%**/-	[H7K4]
ResNet18-6	76%	83%	83%/**95%**/-	2[S3H9K2]+[S3H10K2]
ResNet18-7*	61%	77%	37%/**78%**/80%	2[R3H4K4]+[R3H6K4]
ResNet18-8	77%	**84%**	37%/40%/79%	2[S3H9]+[S3H10]
ResNet18-9	67%	**80%**	38%/<u>78%</u>/-	[H14K2]
ResNet18-10*	38%	58%	31%/**64%**/-	[H7K2]
ResNet18-11*	43%	80%	84%/**88%**/-	[H7K4]
ResNet18-12	39%	61%	41%/**66%**/-	[R3H7K4]
MobileNet-1	67%	79%	82%/**87%**/-	3[H8K2]+8[H11K2]
MobileNet-2*	52%	64%	61%/**79%**/-	3[S3H10]+2[S3H13]
MobileNet-3	81%	86%	84%/**93%**/-	[S3H14]
MobileNet-4*	57%	62%	71%/**88%**/-	2[S3H9K2]+[S3H10K2]
MobileNet-5	77%	**80%**	34%/<u>78%</u>/85%	[H14K2]
MobileNet-6*	57%	71%	37%/**76%**/-	[H7K4]
MobileNet-7	66%	**78%**	32%/<u>70%</u>/-	[H14K2]
MobileNet-8*	38%	58%	31%/**63%**/-	[H7K4]
MobileNet-9	33%	**58%**	23%/<u>43%</u>/-	[H7K4]

Figure 9: Performance results of the generated convolution kernel for AVX512 (Intel i9-7940X Skylake-X), in percentage of machine peak. (a) **Bold** results are the best among all. <u>Underlined</u> results correspond to TTile beating oneDNN with reorder, but not oneDNN convolution only. (b) We also report the performance of the first candidate, as noted by our metric. (c) We took the time to explore exhaustively some of the optimization space of some benchmarks and we report the best performance found. (d) The μkernel reported are the unrolled loops above the vectorization on k, and the potential split between two μkernels, for the best configuration found. For example, "11[H11K2]+[H15K2]" means the combination of 11 times a μkernel with the h dimension unrolled 11 times and the k dimension twice, and a μkernel with the h dimension unrolled 15 times and the k dimension twice.

7 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 [18, 8, 29, 28, 13, 4] that exploit the power of parametric integer linear programming. Although such approaches are suited to expose parallelism [14, 15] and coarse grain locality [8], we believe it may not be the right formalism for tile size selection or register level optimizations.

On the other hand, counting points in a polyhedral, with the Barvinok [5] library, is useful to automatically generate (non-linear) cost models that can be used to optimize neural networks at the graph level [28]. Our implementation uses the Barvinok [5] library to generalize the approach of Li et al. [20] for the selection of an optimized schedule to our class of problems.

Cloog [6] is a powerful algorithm from the polyhedral model that allows to automatically generate imperative code for scanning a union of polyhedra. Polyhedral compilers such as [4, 18] leverage such code generation algorithms. But the approach faces the challenge of dealing with a very general class of imperfect nests and transformations, making it difficult to compete with domain-specific optimizations, eliminating control flow and overhead and missed optimization opportunities. Our code generation involves simple polyhedron scanning algorithms, and the divisibility constraint allows to generate high-quality compiler friendly code without heroic efforts [19].

Optimization of machine learning programs There exist many compilers specialized for machine learning:
PlaidML [9] using polyhedral techniques, XLA [2] for TensorFlow [3], Halide [24], or TVM [10]. TVM, as opposed to most approaches does not rely on the use of architecture-specific CNN/linear-algebra libraries. The strategy of TVM is to select the best schedule using autotuning with a ML-based performance model. Contrary to our approach that decouples the search into micro-kernel optimization and loop tiling and permutation search, the TVM search space is flat. In TVM, optimizations related to strength reduction and register tiling are left to the compiler. Telamon [7] 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 [22] or TCCG [26] 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-matrix multiplication [17]. These frameworks take advantage of advanced optimizations: tensor transposition, tensor blocking, or sub-viewing, data prefetching, vectorization, block scheduling, unrolling and scalar 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 [30] and oneDNN [1] are the best available Intel libraries which implement all those techniques today.

8 Conclusion

The main contribution of this paper is a method to find a good optimization strategy for the single-threaded execution of a convolution kernel with static shape. Unique among code generators and optimization methodologies for convolutions, our strategy considers a wide range of μkernels, omits to pack

tensors for stride-1 access in μkernels, and forbids partial tiles. Instead of partial tiles and the associated overhead, our method consists in enforcing strict divisibility constraints while allowing to combine two well-performing μkernels when problem or tile sizes cannot be decomposed (without control flow or padding overhead) over a single, efficient μkernel. As a result, we consider a larger optimization space than state-of-the-art approaches while also allowing simpler, more efficient code generation. We propose a staged optimization algorithm to explore this optimization space effectively. It starts by characterizing the best performing μkernels on a given architecture, independently of the shape of the convolution. Then the algorithm specializes a on a given convolution kernel, leveraging an analytical model of the footprint, data transfer and reuse at different cache levels. Single-thread execution on AVX512 achieves performance competitive with Intel oneDNN, actually over-performing it in a majority of cases.

Having a highly-optimized CPU execution is the first-step to build a full multithreaded convolution framework. We plan to extend our model to generate a multithreaded implementation of a convolution. Another target is to support any kind of layout with reordering and data copying. Last but not least, we also plan to support other type of tensor operations such as tensor-contraction, which can be seen as a generalization of convolution and matrix-multiplication.

References

[1] oneAPI deep neural network library (oneDNN). https://01.org/.

[2] Xla : optimiser le compilateur pour le machine learning.

[3] 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, 2016.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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)*, June 2008.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] Paul Feautrier and Christian Lengauer. Polyhedron model. In *Encyclopedia of Parallel Computing*, pages 1581–1592. 2011.

[17] Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 blas. 35(1), 2008.

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

[19] 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.

[20] 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.

[21] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P. Sadayappan. Analytical characterization and design space exploration for optimization of cnns, 2021.

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

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

[24] 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.

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

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

[27] 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.

[28] 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.

[29] 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.

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