Decoupling the optimization space of tensor computation for a better understanding of performance on Intel CPU

Nicolas Tollenaere

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Mathématiques et Informatique
Unité de recherche : Laboratoire d'Informatique de Grenoble

Découpler l'espace d'optimisation des calculs de tenseurs pour une meilleure compréhension de la performance sur CPU

Decoupling the optimization space of tensor computation for a better understanding of performance on Intel CPU

Présentée par :

Nicolas TOLLENAERE

Direction de thèse :

Fabrice RASTELLO Directeur de recherche, Université Grenoble Alpes	Directeur de thèse
Guillaume IOOSS Chargé de recherche, INRIA	Co-encadrant de thèse

Rapporteurs :

François IRIGOIN
PROFESSEUR, Centre de Recherche en Informatique (CRI) - Mines Paritech
Brice VIDEAU
INGENIEUR DOCTEUR, Argonne National Laboratory

Thèse soutenue publiquement le **14 décembre 2022**, devant le jury composé de :

Fabrice RASTELLO DIRECTEUR DE RECHERCHE, Inria	Directeur de thèse
François IRIGOIN PROFESSEUR, Centre de Recherche en Informatique (CRI) - Mines Paritech	Rapporteur
Brice VIDEAU INGENIEUR DOCTEUR, Argonne National Laboratory	Rapporteur
Henri-Pierre CHARLES DIRECTEUR DE RECHERCHE, CEA	Examinateur
Michel STEUWER PROFESSEUR ASSOCIE, Université d'Édimbourg	Examinateur
Oleksandr ZINENKO INGENIEUR DOCTEUR, Google Brain	Examinateur
Noël DE PALMA PROFESSEUR DES UNIVERSITES, Université Grenoble Alpes	Président

Invités :

Guillaume Iooss
CHARGE DE RECHERCHE, Inria

Decoupling the optimization space of tensor computation for a better understanding of performance on Intel CPU

Nicolas Tollenaere

March 14, 2023

Contents

Abstract

This work focuses on the problem of optimizing a class of programs we call tensor computation, which includes matrix multiplication, tensor contraction (which is a generalization of matrix multiplication), and convolution. One of the key points of our methodology is the use of what we call a *microkernel*, a small optimized block of code that serves as a building block for the whole program. We present a pipeline called TTiLE that automatically generates a specialized code for a given problem size.

We separate the optimization scheme into two phases : first we select the microkernel with an empiric search. Then we select the outer levels. In this last phase, we leverage some of the work done in performance modelization. This methodology allows us to match the performance of state-of-the-art tools on recent architectures.

To explore our design space and evaluate our search strategy, we crafted a dedicated code generation and evaluation platform.

In the process of this exploration, we had to reconsider the way we evaluate a search process. We introduce a clear cut between what we call the *search space* and the *search heuristic*. We characterize a search space by its distribution, which is evaluated empirically by random sampling. This distribution is then used as a baseline - a search heuristic is worth using only if, for the same number of tests, it can converge toward a better candidate than a random search.

This evaluation process allows us to characterize which choices do improve the search by rising the density of good candidates in the space, and which ones do not. Some of the choices we did early in our research, such as generating only perfectly nested or nearly perfectly nested loops, proved to have little to no positive impact on the final performance.

Our results show that the combination of a very straightforward code generation scheme, a restriction to use only selected microkernels at the inner level, and a random search of the outer levels of the loop nest converges very quickly toward candidates that are competitive with the state of the art (mainly AutoTVM/Ansor, but also Halide and OneDNN) on Intel CPU, both in parallel and sequential settings.

The main takeaway of this thesis is the coupling between space definition and exploration. That is, the performance of a given space exploration heuristic (be it guided by a deterministic model, or based on a learning algorithm) strongly depends on the configuration of the space we are looking into. A very loose restriction of a space where a huge proportion of candidates are inadequate imposes the need for a sophisticated search. On the contrary, restricting more tightly the space of possible implementations by constraining some design choices can make even naive search strategies, such as random selection, competitive with more elaborated ones.

Résumé

Ce travail se rapporte au problème de l'optimisation d'une classe de programmes que nous appelons Calculs de tenseur et qui inclut les multiplications de matrices, la contraction de tenseur (une généralisation de la multiplication de matrices) et la convolution.

Notre approche s'appuie sur des observations introduites dans GotoBlas et BLIS. Notre méthodologie est basée sur l'utilisation de ce que nous appelons des micro-noyaux. Cela consiste à utiliser un petit morceau de code ultra-optimisé qui sert de brique de base à l'ensemble du programme. Nous présentons un algorithme appelé TTiLE qui génère automatiquement du code spécialisé pour un programme de la classe que nous optimisons et une taille de problème donnée.

Le schéma d'optimisation est séparé en deux phases : nous sélectionnons d'abord le micro-noyau sur la base d'une recherche empirique. Ensuite, nous sélectionnons les niveaux supérieurs du nid de boucle. Dans cette deuxième phase nous tirons parti de l'état de l'art existant en modélisation de performance. Cette méthodologie nous permet de concurrencer les outils les plus récents sur des architectures récentes.

Afin d'explorer l'espace des solutions possibles et d'évaluer notre stratégie de recherche nous avons développé une bibliothèque de génération de code dédiée ainsi qu'une plateforme d'expérimentation.

Au cours de ces travaux, nous avons du reconsidérer la manière dont nous évaluons un processus de recherche. Nous introduisons une distinction nette entre ce que nous appelons l'espace de recherche et l'heuristique de recherche. Nous caractérisons l'espace de recherche par sa distribution. Cette distribution est évaluée empiriquement à l'aide d'un échantillonage aléatoire. Une fois évaluée, nous nous en servons comme d'une référence. Une heuristique est utile uniquement si, pour un nombre d'essai donné, le meilleur résultat obtenu est significativement meilleur que celui qu'on obtiendrait avec une recherche aléatoire.

Ce processus d'évaluation nous permet de caractériser quels choix améliorent réellement l'espace de recherche en augmentant la densité de bons candidats. Certains choix faits tôt dans notre recherche tels que s'astreindre à ne générer que des boucles parfaitement imbriquées se sont révélés n'avoir que peu d'impact.

Nos résultats prouvent que la combinaison d'une génération de code très simple, de la restriction à des micronoyaux sélectionnés au niveau interne et d'une recherche aléatoire pour les niveaux externes du nid de boucles converge très rapidement vers des candidats compétitifs avec l'état de l'art (essentiellement TVM mais aussi Halide et OneDNN) sur des CPU Intel, à la fois en parallèle et en séquentiel. Des expériences préliminaires montrent de bons résultats sur des architectures ARM également.

La leçon prinicipale de cette thèse est le couplage entre la définition de l'espace de recherche et son exploration. Plus précisément, la performance d'une heuristique de recherche (qu'elle soit guidée par un modèle déterministique ou

basée sur un algorithme d'apprentissage) dépend fortement de la configuration de l'espace exploré. Un espace peu contraint où une proportion écrasante de candidats sont mauvais rend nécessaire une exploration puissante. Au contraire, un espace plus contraint où certains choix sont restreints en avance peut rendre des stratégies naïve comme une recherche aléatoire compétitive face à des algorithmes beaucoup plus avancés.

Chapter 1

Introduction

Compilation techniques usually care about finding heuristics that would yield the best possible performance for most cases, hiding as much as possible the low level details to final users. This is because the space of possible programs is humongus, so we cannot know in advance which bottlenecks we are going to face. There are other parameters to consider. Depending on the specific use-case, compilation speed, reusability and readability of code can be considered as important or sometimes more important than pure performance. As a result, apart from some simple and well-known optimization such as deadcode elimination or common sub-expression elimination that have been implemented for decades, performance optimization in general-purpose languages and compilers can be considered a lower priority than, say, compilation speed, or language-expressivity. This is not the case we are dealing with though. Some applications are considered worth spending more time, effort and low-level consideration into.

A good example of that is linear algebra. This field of application has led to many researches in the past decades, notably with the BLAS and LAPACK libraries [NVI18] [GVDG08]. This is mostly because an operation such as matrix multiplication is ubiquitous, and because a lot of codebases using matrix multiplication are running for several days. This means that even marginal performance improvements are considered a significant win. In the last decade, the rise of deep learning has further reinforced the importance of linear-algebra super-optimization. Indeed, many layers in a deep neural network are akin to linear-algebra operators, one of them being convolution. This relation between linear algebra and convolution will be made explicit in Section 2.2.2 and is also explained in [ZFL18a]. In a nutshell, a convolution implementation can rely on standard matrix multiplication kernels, modulo some data reshaping, see [CPS06].

There are many ways one can provide highly-optimized operations. These solutions have to make tradeoffs between competing requirements:

- Leverage expert knowledge
- Allow users maximum flexibility

- Portability from one architecture to another
- Minimize engineering costs.

Leveraging expert knowledge means exploiting as much as possible the expertise built over several decades of optimizing linear algebra. Flexibility implies that the user can specify many different kinds of applications, including some that were not anticipated by the vendor. At last, we want to minimize the engineering time we need to spend on hand tuning these applications. At one end of this spectrum, we can find *general-purpose language with optimizing compilers*. Users are given maximum flexibility but, on the other hand, there is not much high-level information that compilers can exploit and they often have to do pessimistic assumptions to account for edge cases and semantic preservation. As a result, general-purpose compilers need sophisticated heuristics and models to retrieve performance. At the other end, there is the solution of providing a *library of hand-tuned routines*. In this case, users are constrained by the choices of the vendor and cannot do anything on their own when a particular routine is missing, when their architecture is not supported, or when the implementation does not yield enough performance on their particular setting. However, as library vendors have complete control of the code, they are free to incorporate all the expert knowledge they can into the implementation. This means the engineering cost is high because this work has to be redone for each new operation and each new architecture. Another limitation of this approach is that one can not specialize the code for the problem sizes.

Other solutions try to find some sweet spots between these two extremes. *Domain Specific Languages (DSL)* are another example of this tradeoff. Instead of providing a full-fledged programming language, vendors provide a dedicated language for the kind of operations they strive to optimize. This allows both to let some degree of flexibility to the user, and to maintain enough invariants so that expert knowledge can be applied without too much effort. DSL can be stand-alone, which allows them to provide custom syntax to their users. One example of this is Halide [RBA+13] or Polymage [MVB15]. They can also be embedded in another language. This is called an *embedded domain specific language (EDSL)*. In the latter case, one can leverage other libraries and tooling available in the embedding language. This also avoids users having to learn a whole new language for the sake of a particular implementation, which can hamper adoption.

My work is focused on the special case of tensor computation, a class of programs that both have applications in a large set of fields, and have nice properties that make optimization somewhat easier. This class of programs includes tensor contraction, matrix multiplications (which is a special case of tensor contraction), and convolutions. Amongst tensor computation, we have put a special effort into convolution, which is a common operation in deep learning and image recognition. Besides being the last "hot stuff" in the compilation and optimization community with articles such as [LXSR+21, CMJ+18] focused on the optimization of this particular operator, convolutions used in practice often exhibit small dimensions which make at least some of the state-of-the-art

optimization schemes inefficient. This is one of the work hypotheses behind my thesis: that at least some of the usual assumptions are invalid in the context of convolution on CPU, and therefore that we can relax these constraints and find another way through.

Nevertheless, this work extends on the principles of BLAS and BLIS libraries (which are described in Chapter 2) and thus contrast with other types of approaches on the matter. Inspirations were taken from many other works such as Telamon [BPP+17], Ioopt [OIT+21] and the place of the final implementation on the previously discussed spectrum (libraries - generic language with compiler) is a matter of discussion. We will see that it leans towards an autotuning solution in some aspects, and on a more "library-like" or maybe DSL-like design in others.

We decompose the problem of building an efficient convolution in two parts : (i) building a space of possible implementations (called a *search space* from now on), (ii) exploring this space to choose one specific implementation. These two phases are coupled in the sense that the search space eliminates implicitly all possible implementations that it does not contain. We will discuss the tradeoff we have to make between allowing a lot of candidates, which implies letting many bad candidates in the space, and building our space in a opiniated manner, which hopefully prunes a lot of bad candidates beforehand and therefore accelerates convergence towards good ones, but makes it easy to miss some good potential candidates.

In this work, we focus on CPU implementation. Nowadays a huge part of the high-computing work is done on GPUs. However, deep learning convolutions are used in two different settings: training, where the network tries to learn to discriminate features from its learning set, and inference, once it has done its training, and applies it to new instances. GPU is mostly used for the training phase, where the goal is to reach the highest possible throughput. When doing inference, CPUs are relevant because the amount of computation and the level of parallelism are lower, and we care more about latency. This is why optimizing convolutions for CPU is an important task.

This document starts with a background chapter that introduces key notions to the design we are going to present. This part will explain concepts such as microkernels or streaming which then act as building blocks for the solutions I have implemented.

A prerequisite to conducting this study was to have an appropriate setting for testing many different code configurations. Thus, a part of my contribution was to design (yet) a(nother) code generation library that would expose all necessary options to twist the code I want to test as needed while being a more lightweight solution than writing C code by hand. The details of this code generation can be found in Chapter 3. The strategy was to start with the simplest code generation that would fit our needs. This code generation was then enriched on the fly with new primitives when they proved useful for performance.

Once this tool was available, the next step was to integrate it into an experimental platform that would measure performance and verify semantic correctness. This is described in Chapter 4.

Then the Chapter 5 describes the optimization process per-se. We will show how we bake our core assumptions into the design of the search space - the space of all possible implementations for the problem size we are dealing with. Then we will discuss different ways we tried to explore this space and the conclusions that arise from it. Chapter 6 presents our results in more details.

This document ends with a discussion on the matter: on a personal level, the way we did our research, which inspirations were used and what could have been done better. The chapter also reflects on the matter of optimizing programs at a higher level.

Chapter 2

Background

Since linear algebra optimization has applications in many domains of computing, it has been a research subject for decades. Therefore, many techniques have been explored and we strive to build upon them. The point of this section is to define the key notions that we are going to use through this work, as well as present the state of the art and the different solutions that already exist. A lot of the work done on optimizing linear algebra has been focused on parallelization, either by using multiple cores on the same machine or by relying on a computer cluster. However, our work focuses more on the optimization of linear algebra on a single core.

This optimization relies on well-known optimization techniques such as unroll-and-jam, vectorization, exposing instruction-level parallelism, or tiling. One of the most important notions is the definition of a *microkernel* defined in Section 2.1.1. It consists in a small piece of highly-optimized code that serves as a building block. It is central to the way we decouple the problem of building a good implementation for the kind of algorithm we optimize. We also define what is a convolution from an operational point of view, and how the optimization of this operator relates to the general field of matrix multiplication optimization.

2.1 Existing tools and related work

As we said, many tools were developed for linear algebra optimization in the last decades. In this section we are going to present a few of them. We start by the most important : GotoBLAS [GG08] (for Basic Linear Algebra Subroutines/ BLAS Like Interface Software) [Lou88]. This family of works has helped define some of the concepts we are going to explore in the next sections of this document. Therefore we will spend a lot of time explaining these concepts, especially those of microkernels and *hierarchical tiling*.

2.1.1 GotoBLAS/BLIS : a systematic building of efficient matrix multiplication

We start by describing the BLAS routine. BLAS means *Basic Linear Algebra Subroutines* and consists of a set of standard interfaces and implementations for some basic linear algebra routines. It dates back as far as 1979 [Lou88]. GotoBLAS [GVDG08] is a specific implementation which settles some of the principles of implementing a high performance matrix multiplication. In 2015, BLIS [VZvdG15] took a look back at these principles and generalized them to provide a library that emphasizes even more portability and reusability. Here we want to give a first intuition of what makes this implementation strategy both efficient and simple.

Hierarchical tiling: an overview of the GotoBLAS strategy

As we said, BLAS (Basic Linear Algebra Subprogram) is a specification for a variety of linear algebra low-level routines. As these routines were reimplemented multiple times to take advantage of new architecture designs, this common specification made sure that everyone could rely on the same signatures and still get the benefit of an expert-made implementation.

One key point of GotoBLAS implementations is that they rely on a strict separation between the low-level and the higher-level parts of the code. The low-level part is called a *microkernel*. It consists in a fixed-size small subpart of a computation written directly in assembly language. Its design is intended to make the best possible use of the architecture features, such as vector instructions, Instruction level parallelism, or prefetch. We will get back to this design later in Section 2.1.1.

The higher-level part is written in C and makes calls to the microkernels. In GotoBLAS design, it also ensures that data are shaped according to the microkernel requirements. Indeed, a microkernel operates on a fixed-size, contiguous set of subarrays whose layout should match the iteration pattern. That is, accesses inside the microkernel have to be done in a contiguous fashion.

This requirement in turn makes it necessary for the outer-level code to do some reshaping of the data, which consists in reordering and compacting array elements in smaller buffers. This is called *packing* in literature.

The overall scheme is called hierarchical tiling: it consists essentially in making sure that there is a tile that fits in the cache at every level to exploit data reuse.

In GotoBLAS, packing is done at each level of the cache hierarchy. This means that at each point in the loop nest where the footprint overflows a given cache level, there is a copy/reshuffling that makes sure accesses are done linearly.

BLIS [VZvdG15] is another implementation of BLAS, designed to maximize genericity and code reuse. It does so by providing a set of routines focusing on a kind of polymorphism. Indeed, a typed API allows the user fine-grained control over the dimensionality of computation (pointwise operations on vectors, pointwise operations on matrices, 2d operations...) and the types of matrice elements

```
1  for (i = 0; i < I; i++) {
2    for (j = 0; j < J; j++) {
3      for (k = 0; k < K; k++){
4        C[i,j] += A[i,k] * B[k,j];
5      }
6    }
7  }
```

Figure 2.1: Naive matrix multiplication

(single or double precision floating point numbers, 32 or 64-bit integers...), and the original layout. This flexibility is not attained by multiple implementations but by adapting dynamically around a bunch of *specialized microkernels*, which is identified as the main performance bottleneck.

Microkernel design

In this paragraph, we describe how GotoBLAS has set the design of its microkernel and how it is directly guided by micro-architectural considerations.

At the lowest level, performance essentially depends on two parameters : use the full power of vector instruction especially, in our case, single fused multiply-add, and make sure that the core pipeline is full. The second point boils down to two contradictory requirements :

- hide latency by adding enough parallel instructions in the basic block
- minimize spilling, that is, make sure that most data is register-resident

To illustrate that, we present a step-by-step observation to see how to rewrite a naive implementation of a block matrice-multiplication into an efficient one. Recall the definition of a matrice-multiplication - we are using the Einstein's Notation here similarly to [RBA+13] :

$$C[i,j] = C[i,j] + A[i,k] * B[k,j]$$

If we were to implement it in a naive way such as in this code :

There would be several problems hindering the performance. Firstly, it is impossible to vectorize this code, as the inner loop is a reduction.

In this setting, vectorizing over a dimension imposes some constraints :

- this dimension should be parallel - no data dependencies across iteration over this dimension
- this dimension should be innermost in the layout of all tensors it accesses.

These constraints are not strictly necessary to apply vectorization from a semantic point of view. The first one is too tight: it should be enough to check

```
1  for (int i = 0; i < N - 32; i++) {
2    A[i + 18] = A[i] * B[i];
3    }
```

Figure 2.2: Example of vectorizable loop despite it not being parallel

that there is no dependency from one iteration to another in the same vector-size frame. For example, assuming a vector size of 16, the loop in Figure 2.2 can be vectorized, even if there is technically a dependency over dimension i.

However, such a dependency could have a performance impact, as we could potentially pay for the latency of a vector instruction if we choose to look for instruction-level parallelism over the same dimension - this is explained later. Moreover, we place ourselves in a setting where all loops are either parallel or reduction loops (simple dependency from an iteration to the previous one). Therefore, it makes sense to tighten our constraints, as such skewed dependencies that would allow vectorization of non-strictly parallel loops never occur.

The second constraint can be relaxed if we allow ourselves scatter-gather operations, which consist in building a vector from non-contiguous elements. However, this has a performance cost. When we want to vectorize over a dimension that is not innermost for all tensors accessed, we will rely on packing, which is explained later in Section 2.1.1, which builds a new intermediate tensor with the proper layout.

Given these constraints, to make a naive matrix multiplication efficient, the first major transformation would be to do a loop exchange to bring a vectorizable loop at the inner level. Here we assume a fully permutable computation space where all loop exchanges are semantically valid. Dimensions i or j are the only candidates for vectorization, as k is not a parallel dimension. Either could be vectorized if we make sure that they are the inner dimensions of their respective matrices. This can imply some reshaping of the data in the worst case, but for now, we will assume that j is the inner dimension of both C and B, and is therefore vectorizable. Figure 2.3 shows a vectorized version of the code after permutation.

This is better but still not enough: we still have a cross-iteration dependency over the k loop. Therefore we have to pay for the latency of the fused-multiply-add instruction and we do not take advantage of instruction-level parallelism. Therefore one needs to iterate again over a parallel dimension outer of the strip-mined loop. These iterations can be done over either i or j for the same reason as before. The control flow can also hinder the performance, as it makes it harder to fill the pipeline, even if branch-prediction can mitigate this to some degree on modern architectures. This is one the reasons for unrolling loops, the other being that a bigger basic block allows for more registers to be used, thus avoiding anti-dependencies. At low-level, register renaming exploits automatically the fact that there are more physical registers than the logical ones in order to break these anti-dependencies.

Again either i or j can be used. Let us assume we unroll over the i dimension.

```
1  for (i = 0; i < I; i++) {
2    for (j0 = 0; j0 < J; j0 += VECSIZE) {
3      for (k = 0; k < K; k++){
4        // broadcast(v) = [v, v, v,...]
5        a_ik = broadcast(A[i,k]);
6        // vector loads
7        c_ij = C[i, <j:j+VECSIZE>];
8        b_kj = B[k, <j:j+VECSIZE>];
9        // vector multiply-add
10       c_ij += a_ik * b_kj;
11       // vector store
12       C[i, <j:j+VECSIZE>] = c_ij;
13     }
14   }
15 }
```

Figure 2.3: Vectorization on j

This yields the code found in Figure 2.4.

Another subtlety hits here. The line $C[i,j] += A[i,k] * B[k,j]$; hides a series of instructions :

- a read at the address &C[i, j] into a register
- two reads at address &A[i, k] and &B[k, j]
- a multiplication followed by an addition
- a write at the address &C[i, j]

This means that the same memory address (&C[i, j]) will be loaded into a register and then stored back at each iteration of the surrounding k loop. We can avoid this by performing a transformation called scalar promotion, which consists in transforming a slice of the array into a group of variables that can be stored in registers, thus avoiding memory accesses. In this specific case, we use vector registers to do so. This is shown in Figure 2.5.

This basic block was unrolled by a factor of two for the sake of readability but in practice, the unroll factor is usually higher. The value of the unroll factor depends on both the number n_r of vector registers and the latency l of a multiply-add instruction.

GotoBLAS makes the following design choices about this microkernel: hide latency and avoid spilling entirely. The inner basic block should be unrolled enough to hide the latency, that is, after having issued an instruction *inst* enough instructions that do not depend on *inst* should be issued so that the next instruction that needs the result of *inst* will have its dependencies immediately available. But it should also be small enough so that there is no need for spilling.

This process is described in more detail in [LISQO16]. From these requirements we can derive a formal constraint : Let l be the latency of a float multiply-add instruction, n_r be the number of vector registers available. Let u_i the unroll

```
for (i = 0;i < I; i += 2) {
  for (j0 = 0; j0 < J; j0 += VECSIZE) {
    for (k = 0; k < K; k++) {
      c_ij = C[i, <j:j+VECSIZE>];
      b_kj = B[k, <j:j+VECSIZE>];
      // broadcast(v) = [v, v, v,...]
      a_ik = broadcast(A[i,k]);
      c_ij += a_ik * b_kj;
      C[i, <j:j+VECSIZE>] = c_ij ;

      c_i1j = C[i + 1, <j:j+VECSIZE>];
      a_i1k = broadcast(A[i + 1,k]);
      c_i1j += a_i1k * b_kj;
      C[i + 1, <j:j+VECSIZE>] = c_i1j;
    }
  }
}
```

Figure 2.4: Unrolling on i

```
for (i = 0; i < I; i += 2) {
  for (j = 0; j < J; j+=VECSIZE) {
    c_00 = C[i,<j:j + VECSIZE>];
    c_10 = C[i + 1,<j:j + VECSIZE>];
    for (k = 0; k < K; k++) {
      a_0k = broadcast(A[i, k]);
      b_kj = B[k, <j:j + VECSIZE>];
      c_00 += a_ik * b_kj;
      a_1k = broadcast(A[i +1, k]);
      c_10 += a_1k * b_kj;
    }
    C[i     , <j:j+VECSIZE>] = c_00;
    C[i + 1, <j:j+VECSIZE>] = c_10;
  }
}
```

Figure 2.5: Scalar promotion

factor on dimension i, and u_j the unroll factor on dimension j. We assume that the loop is vectorized on j, which implies that the actual footprint of the basic block over dimension j is $u_j * VECSIZE$. We get a pair of inequalities :

$$\begin{cases} u_i * u_j + u_j + 1 \leq n_r \\ u_i * u_j \geq l \end{cases} \tag{2.1}$$

Of course, matrix multiplication is symmetric over matrix A and B and dimensions i and j. Therefore we could choose to vectorize dimension i instead, which would invert u_i and u_j in this system. Any choice of factors that verify these constraints is a good candidate for a microkernel. There is no guarantee that this system of inequations has at least a solution, but in practice, all architectures we work on have some. The performance penalty incurred for breaking these constraints is also architecture dependent. For example, the cost of spilling can vary from one machine to another. This will be discussed and evaluated in more detail in Chapter 4.

Writing microkernel: a language dilemma

When writing such a low-level piece of code, the question of the most pertinent language to use arises. Microkernels were historically written in assembly language and still are in BLIS. This is in part the reason why they were considered expert-level subjects. Assembly was chosen to put a maximum of control into the hands of the developers. This is reasonable for such sensitive code where anything can hamper performance. However, this choice could be discussed with regard to the capabilities of modern hardware which has many more facilities to transparently hide bottlenecks. For example, while developers and compilers targetting older architectures used to have the burden of accounting for memory access latency and were in charge of scheduling the program accordingly - so that memory accesses are done early enough that they are ready as soon as they are needed - the out-of-order execution implemented by Intel architectures and others can do this scheduling automatically. Hardware prefetching is another strategy that helps amortize latency by speculatively bringing data in the cache before they are explicitly needed, to make them available as soon as possible - assuming the prediction was right and they are indeed accessed soon after. That, along with other similar hardware optimizations, can question the need of programming in assembly, as this choice has impacts such as increasing development cost, hurting portability, and imposing on developer the burden of doing manually tasks otherwise well-handled by compilers such as register allocation and scalar evolution. Maintaining correctness is also a huge challenge.

In contrast, we could choose to write our microkernel directly in C, thus relying on the compiler to do the transformations needed to exploit at least vector instructions. This proved insufficient in practice: the compiler was not able to vectorize the code the way we want it to. This solution also suffers from a lack of portability across compilers and different versions of the same compilers. An alternative is to write most of the code in C with the additional help of

architecture-specific *intrinsics*. Intrinsics are special C functions that map directly to an assembly instruction. This allows forcing the vectorization scheme that yield performance while offloading other tasks such as register allocation or handling induction variables to the compiler. We will discuss the question of the impact of the compiler in more detail in Section 4.6. In conclusion, while this tradeoff was the best we found it is not completely foolproof. It happened that in cases where basic blocks were particularly long (above tens of thousands of lines of code), compilers failed to yield the expected performance, sometimes because of poor register allocation strategy. Nevertheless, it has proven reliable as long as we limit ourselves to smaller generated codes.

Outer-level Strategy : Packing and streaming

Implementing a suitable microkernel is not enough. While it accounts for core-level considerations such as hardware-pipelining (by exposing instruction level parallelism) and register locality, it does not take care of the cache usage. Memory accesses latency can vary from a factor from one to hundreds depending on what level of the hierarchy the address we are accessing is located. Available bandwidth also depends on this level of access. As a reminder, cache policy is the way a given architecture will choose which cache line it is going to evict when new access is made. There are different types of strategies in this regard, the simplest and best-known one being *least recently used* (LRU) where the cacheline that remained untouched for the longest time is chosen to be evicted. Cache policy is often considered an industrial secret. Moreover, modern architectures incorporate many features such as hardware prefetching, complex cache replacement policies, and others. As a result, predicting cache behavior for a given program is a challenging task. As we will see in Section 5.4.2, it is actually challenging even when simplifying the cache model to the extreme (for example considering we have a perfect control over which cache line will be evicted).

Guiding principles Instead of relying on a specific cache model, BLIS simply assumes that caches work best when they are confronted with the simplest memory access patterns. Their strategy is to make sure the accesses are done in the most regular way possible. In other words, accesses done at one given level of cache should be done as much as possible in a contiguous way. The bet is that cache policies, however complex they are, are more likely to behave smoothly with such nice accesses. This is also an elegant way to retrieve some free portability: while a cache model is most likely tied to a given architecture and can become irrelevant when applied to a new one, this strategy makes as few assumptions as possible and thus can be applied largely.

In practice BLIS achieves this goal in two ways. The first one, called *streaming*, consists in, for every level of cache, from the inner level to outer :

1. choosing a tensor

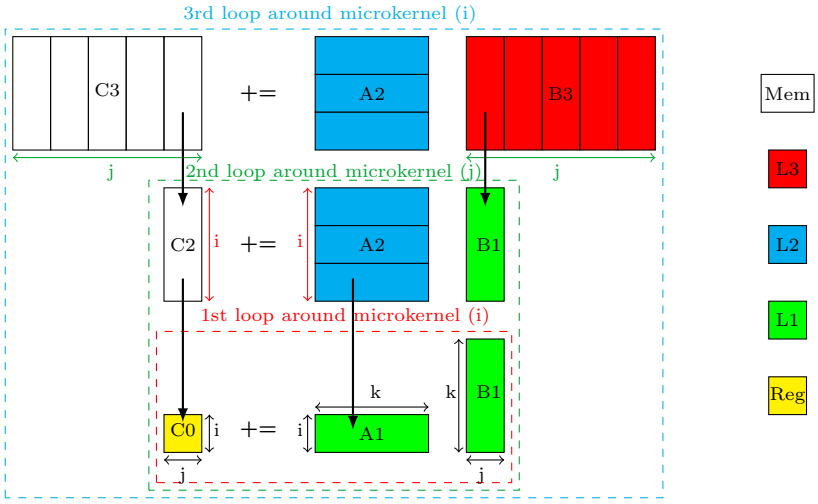


Figure 2.6: GotoBLAS tiling strategy

2. tile the code in such a way that the *footprint* of this tensor is close but smaller than the cache capacity of the current level

3. maximize reuse over this tensor and only this one

This means that at every level of cache, accesses to all tensor but one will miss. This is illustrated in Figure 2.6 for the first two levels of cache. It is worth noting that the work of Olivry and al. [OIT+21] has proven later that it is the right way to optimize the ratio of computation over communication - communication here means the volume of data that is retrieved towards the cache - we will see that in more details in Section 5.4.2.

The other strategy used by BLIS is, again, *packing*. By reshuffling the layout to make it match the iteration pattern, one makes sure that accesses done to a tensor at a given level are contiguous, which makes them hopefully prefetch-friendly and minimizes cache conflicts.

BLIS : strengths and limitations

Overall, what makes the BLIS approach stands out is that it gives a relatively simple and systematic way to build a high-performance linear algebra kernel. Besides, the separation of the low-level assembly routines and the enclosing code allows for a great composability: the low-level work can be done only once and all other applications can build on it with a high-level API. However, this flexibility comes with a price: *it is the responsibility of the outer level of the code to adapt its layout to the requirements of the microkernel.* As we said before, this implies data shuffling which can be somewhat awkward, especially since the microkernel size and parameters are not an easy target for transposition routines [SSB17]. This problem is exacerbated when some of the problem dimension sizes

are small, which is often the case when we deal with convolution in real-world Convolutional Neural Networks. Therefore, packing can have a cost that could be avoided otherwise and whose benefits is unclear in terms of performance. Besides, as the microkernel has a fixed size, it does not divide perfectly every possible problem size and thus the library has to do some residual work that can not be handled by the microkernel. Depending on the implementation of the partial tile and the relative size of the problem size and the microkernel, this can also hamper the performance when the partial tile execution becomes dominant.

2.1.2 Tensorflow

Tensorflow [ABC+16] is one of the main computation libraries designed for machine learning networks. It exposes a fixed API of operators that let users define a dataflow graph. Contrary to other deep learning libraries such as Caffe [JSD+14], dependencies across nodes in this graph can be dynamic and depend on execution time values. As TensorFlow is embedded in several languages such as Python, JavaScript, or C++, In terms of the spectrum of implementation we presented in Section 1, this can be placed somewhere between a library and an embedded domain-specific language. The execution model of TensorFlow relies on calls to specialized linear algebra libraries such as Eigen. This limits both its flexibility and its ability to reach top-level performance, as it can not be specialized to specific problem sizes.

2.1.3 Halide

Halide is a Domain Specific language embedded in C intended to help programmers design image-processing applications. Its main contribution was to decouple the algorithm (that is, the semantic specification) from the schedule - which would correspond to the operational semantic. Indeed, Halide allows users to provide a semantic skeleton that can be later completed by a *schedule* - which can be expressed as a polyhedral mapping from the iteration space to a totally-ordered set. This in turn allows a good encapsulation of the optimized part of the code - users of the library usually only care about the semantics and not about the implementation. [RBA+13]

2.1.4 Polyhedral tools

Polyhedral compilers such as Diesel [ERR+18], Polly [GGL12], Pluto [BHRS08], PPCG [VCJC+13], Tensor Comprehensions, [VZT+19], Tiramisu [BRR+19] are able to automatically generate multi-level tiled code for affine loop nests. Diesel is a DSL for neural networks. Networks are represented internally as a directed acyclic graph and all operators are optimized together. It can also take problem sizes into account when they are known statically and optionally apply auto-tuning techniques. Tiramisu also exposes a custom DSL and optimizes both

dense and sparse neural networks and uses an internal polyhedral representation to do so. Contrary to Diesel, it includes a scheduling language that let users specify a particular implementation. Polly is a polyhedral optimizer integrated into LLVM, it can thus be applied to any setting that relies on LLVM. Pluto is a source-to-source compiler that performs automatic tiling and parallelization. PPCG takes a sequential C code as input and generates an equivalent parallel code for CUDA.

These tools deal with a much broader class of programs than us, as we limit ourselves to a small subset of what the polyhedral framework can express. As a result, their performance is usually several factors below what can be achieved with more specialized tools, as it is shown in [Bon20]. This is because some of the lower-level optimization needed are not easily expressed in this framework, even if they significantly outperform general-purpose compilers on naive code.

2.1.5 MLIR

MLIR (Multi-Level IR) [LAB+21] is a recent effort to develop an intermediate representation that let users define *dialects* specialized for a given application, such as linear algebra. These dialects aim to bridge the gap between lower-level IR such as LLVM IR and high-level languages. They also strive to be extensible and reusable. Users can define optimization passes at dialect levels that allow to implement a GEMM (general Matrix Mulitiplication) with the same principles as GotoBLAS, and attain performance close to BLIS level. MLIR tries to leverage some notions from the polyhedral framework to express program transformation easily.

2.1.6 BOAST : Source-Source Optimization for optimizing loop structures

BOAST (Bringing Optimization through Automatic Source-to-Source Transformation) [VPG+18] is a tool developed by Videau et al. that, as hinted by its name, performs transformation at the source level to optimize loop-intensive programs. It is embedded in the Ruby programming language and as such qualifies as an embedded domain-specific language. It is a metaprogramming tool that allows users to specify their compute kernel and the way they want it to be optimized in a high-level language. As in AutoTVM, the semantics of the kernel and the optimization are separated. BOAST supports many targets such as OpenCL, CUDA, C or FORTRAN.

2.1.7 TVM

TVM [CMJ+18] is a deep learning compiler that exposes a DSL to define custom operators and optimize them with an autotuning framework. It essentially gives up on finding a relevant performance analytical model. The reasoning is that given the size of the optimization space, that depends both on the semantics of the operation we are trying to optimize and microarchitectural choices, and the

fact that we can usually afford a huge training time if needed (these operations are going to run thousands if not millions of times), the problem is suited for machine learning techniques. Therefore, in the fashion of Halide, TVM offers a framework that provides a way to define a custom operator with a special semantic, completed with some primitives that help define a search strategy.

This choice amounts to moving the expertise from the compiler field to the machine learning field. However, this setup still requires some amount of compiler expertise, as defining relevant features is a quite demanding task.

As a platform, TVM provides good mechanisms that allow users to experiment with different strategies.

One of the main limitations of TVM is that they do not build on previous knowledge of optimized kernels. That means that a concept such as a microkernel has to be rediscovered by the search strategy. This in turn implies that many candidates will have to be explored before this knowledge can be rediscovered. Moreover, some degree of expertise at least is still needed to define the features that should be explored. This can cause troubles because one can be tempted to somewhat hardcode aspects that are known beforehand to be useful or to prevent some patterns known to degrade performance.

There is an improvement of TVM called AutoTVM (or Ansor) that we failed to consider properly at first. It somewhat moves toward even more automation as some choices that were hardcoded in TVM are now part of the search space, such as what we call the permutation of dimensions (defined later in Section 5.4. When used properly the results were quite impressive and improved greatly over the basic version. Analyzing these results earlier would have allowed us to improve further our optimization flow but I have to let this for future work. We will detail this part later in Section 5.4.

2.1.8 Data-movement modelling : Mopt and Ioopt

As cache behavior is identified as one of the main parameters that impact performance at outer levels, it makes sense to use a model that predicts this behavior. [DZ03] is a good example of such a model: the reuse distance metrics it tries to estimate is an approximation of the number of accesses that separate two consecutive accesses to the same data.

Mopt [LSV+19] and Ioopt [OIT+21] are two other similar models. Given a schedule and a cache size, they try to predict, assuming some simplifying hypothesis on cache policy, which amount of data will be loaded towards this cache. In practice, Ioopt can provide both a lower and a higher bound on this metric. These bounds can be tight on favorable cases such as ours. As these models provide a symbolic expression they can be fed into a numerical solver that will attempt to minimize them. This assumes of course that cache effects are the bottleneck of the application, which is false if one does not take care of things such as instruction-level parallelism and vectorization. In consequence, these models can only be used in conjunction with other techniques, such as relying on a microkernel. One of our implementation attempts relies on Ioopt

to choose a tiling scheme, and as such, we describe it in more detail in Section 5.4.

On the other hand, Mopt has a corresponding implementation that exploits its model and therefore qualifies as a competitor.

2.2 Operators

Now we are going to define the operators we are striving to optimize. Apart from matrix multiplication we omit for brevity, we will define two of them: convolutions, which we spent most of our time optimizing, and tensor contraction. We will also see how these three operators (matrix multiplication, convolution, and tensor contraction) are actually close in terms of semantic and operational definition. This observation has been done in [ZFL18a] and has a deep consequence on the way we optimize convolution: we follow the principles set on matrix multiplication before.

2.2.1 Tensor Contraction

An interesting operation we will investigate in this work is the tensor contraction, which is a generalization of a matrix multiplication of higher dimensionality. For reasons that also apply in the case of convolution, it can be optimized with the same guiding principles as matrix multiplication. Indeed, tensor contraction is a generalization of matrix multiplication, as we will show with an example.

Here is an example of tensor contraction, again with Einstein's notation :

$$Out[l1, r1, l2] += Left[l2, k1, k2, l1] \times Right[r1, k2, k1]$$

To show the analogy with matrix multiplication, we can apply the following transpositions (permutation of indices) :

$$\begin{cases} Out[l_1, r_1, l_2] \Rightarrow \widehat{Out}[l_1, l_2, r_1] \\ Left[l_2, k_1, k_2, l_1] \Rightarrow \widehat{Left}[l_1, l_2, k_1, k_2] \\ Right[r_1, k_2, k_1] \Rightarrow \widehat{Right}[k_1, k_2, r_1] \end{cases} \tag{2.2}$$

followed by grouping a few dimensions together :

$$\begin{cases} l_1, l_2 \Rightarrow l_1 l_2 \\ k_1, k_2 \Rightarrow k_1 k_2 \end{cases} \tag{2.3}$$

This yields the following definition :

$$\widehat{Out}[l_1 l_2, r_1] += \widehat{Left}[l_1 l_2, k_1 k_2] \times \widehat{Right}[k_1 k_2, r_1]$$

Which is matrix multiplication. Therefore a tensor contraction is a matrix multiplication modulo some reshaping. It can be implemented by a transposition followed by a call to an optimized matrix multiplication library. This

Figure 2.7: Pseudo-C code for a 2D-convolution

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

implementation can be improved by trying to do this transposition "on the fly" during the computation.

2.2.2 Convolution

A *convolution* is an operation commonly used by the deep learning community. Indeed, there is a whole class of deep learning networks called *Convolutional Neural Network* (that we will call CNN from now on) that relies on this specific operation. It is a key part of all modern image recognition tools and is very compute-intensive. As a consequence, they are a major bottleneck of a lot of applications.

In Einstein's notation, a convolution operation is defined as follows :

$$O[b, h, w, k] += I[b, h + r, w + s, c] \times K[r, s, c, k]$$

h and w are the height and width of the output image, r and s the dimensions of the sliding window, c the number of input channels, k the number of output channels, and b the size of the batch (number of images on which the network is applied in parallel). This notation is a compact way of defining tensor operations. All indices that only appear to the right of the equation are implicitly summed over. Therefore, this is equivalent to the loop nest in Figure 2.7.

Now we are going to dive into the specifics of optimizing a convolution on CPU.

Convolution is almost a matrix multiplication

It has been noted in literature that convolution can be viewed as *a particular matrix multiplication* after some dimension renaming and fusion. This is very similar to what we have shown in the previous section about tensor contraction. If we look at the definition of a matrix multiplication in Einstein's notation :

$$C[i, j] += A[i, k] * B[k, j]$$

And fuse the convolution dimensions in the following way :

$$\begin{cases} b, h, w \Rightarrow bhw \\ r, s, c \Rightarrow rsc \\ k \Rightarrow k \end{cases} \tag{2.4}$$

We see that our convolution becomes similar to a matrix multiplication :

$$O[bhw, k] += \hat{I}[bhw, rsc] * K[rsc, k]$$

$\hat{I}$ is the result of applying *im2col* on tensor I. It is defined as :

$$\hat{I}[b][h][r][w][s][c] = I[b][h + r][w + s][c]$$

This definition makes it obvious that many elements of $\hat{I}$ are duplicated. While this makes for a bigger footprint, it also greatly simplifies the pattern of accesses.

This reshaping is necessary for the following reasons. As the astute user would have noted, the access function of the input tensor of the convolution is slightly twisted. It turns out that I is not a 5-dimensions tensor, but a 3-dimensions tensor with a non-injective access to dimensions h and w - non-injective in the sense that for example $h = 0, r = 1$ and $h = 1, r = 0$ actually are the same slice of the tensor. This means that convolution is not strictly equivalent in terms of accesses to a matrix multiplication, and that standard matrix multiplication routines would not be semantically correct if they were applied naively. The consequence is that using a matrix multiplication routine implies a reshaping of the data in the general case - even if a smart dispatch could skip this step when appropriate for example when $R = S = 1$. This reshaping is described in detail in [CPS06].

As a result, some of the first implementations of convolution used to just do the necessary work on input data to make them suitable to a call to a matrix multiplication library, which leverages decades of research and engineering. This is explained in [CPS06].

This reshaping is artificial in the sense that it is justified only by the fact that highly-tuned matrix multiplication libraries pre-exist convolution optimization. Re-using matrix multiplication optimization principles but not code would allow yielding much more performance and this is what has been done in further work such as [ZFL18b].

Focusing on Inference

Our work is focused on CPU optimization. In a real-world context, CNN applications use CPU to do inference, whereas the training is done on GPU. For inference, the goal is to reduce the latency of a single application on one input image. This means that we will assume *a batch of size 1* from now on and that the dimension b will be omitted.

2.3 Modelization of performance

Building an optimized program requires some prediction of the way the architecture will behave upon running it. Optimizing compilers incorporate several performance models to guide choices. Some of the approaches we presented could benefit from such a model. Overall, there are several axes along which a model can be placed :

1. Quick evaluation - how long it takes to compute
2. Accuracy of the prediction
3. Explainability
4. Differentiability

The first one is a matter of *usability*: there is usually no point in having a model if running the model takes more time than running the program it is supposed to evaluate. The second one is how much the model correlates with the actual performance. The third one asks whether we are in a "black-box" approach or if the model tries to have clean, separable components that can be modified independently. The fourth one measures how much this model can be "reversed" if it is possible to use it to guide a search.

It is worth noting that a fully differentiable model can somewhat compensate for a slow execution if it allows to find more quickly an implementation that minimizes the metric given by the model.

The polyhedral world has proposed a few interesting techniques. A noticeable one would be the cost function used by the Pluto Heuristic implemented by Bondhugula and al.[BHRS08]. In this work, a data locality metric is used as a proxy for actual cache performance, in the same spirit as [DZ03] or [SSF+12]. Given a tile size, Pluto aims to minimize the average number of iterations between two consecutive accesses to the same address. It does it by solving for every dimension in the problem an Integer Linear Problem that maximizes data locality (for a certain definition of data locality we will not detail). As a result, the objective function of Pluto is quite good over the explainability and differentiability (it is possible to analytically derive a solution that maximizes the objective). However, the tile sizes have to be chosen manually and it can not account for multiple objectives. Some other works such as this one extend it, but try to optimize both parallelism and data locality.

As we have seen, BLIS chooses to get rid of the use of a model entirely and counts solely on the fact that a smoother pattern of accesses can (hopefully) make the cache behavior nicer.

In general, which search strategy to use and in particular whether using or not a model cannot be decided in isolation, and the tradeoff depends a lot on external constraints. Do we allow ourselves to know the size of the problem and the architecture ahead of time or not? If we want to build a tool that can provide high-performance on problem sizes and architecture that are unknown beforehand, then machine learning tools relying on training are prohibited, and

any model can only be used if its cost is amortized in one run by the performance gain. On the contrary, when both problem sizes and architecture are known and available ahead of time, this opens a lot of opportunities, in term of autotuning and model alike. In this work, we mostly assume that both problem sizes and architecture are known ahead of time.

2.4 Conclusion

In this chapters, we presented several things. First the operators we are going to optimize. As we have seen, they are part of the same algorithm family. Having to express this family of related but slightly different specifications motivated us to build an embedded DSL code generation framework we describe in Chapter 3. We also described the state of the art, different kinds of optimization techniques, and different tools that make use of these techniques. This in turn will come into play when we describe our own design choices in our implementation. And of course, our competitors will serve as a baseline for our evaluation in Section 6.

Chapter 3

Code generation

To explore a wide range of code variants, having an ergonomic way to design and test new kernels is a requirement. In particular, we want to separate the semantics and schedules of the kernel from the implementation details. Controlling data layout and reshaping is also needed. Such functionalities can be found in other frameworks, such as TVM, as mentioned previously, or Halide [RBA+13]. While Halide has its own syntax and language, TVM is directly embedded in Python. Neither of those, however, provides a sufficient level of control for our needs. Indeed, both are lacking some kind of memory layout specification, and also do not let the user specify some details such as register tiling. We preferred an explicit code generation scheme to template-based or meta-programming approaches. This code generation framework was coded in the OCaml language and benefits from some its advanced typing features.

The *iteration space* is an integer vector, taken by the loop indices enclosing a given computational statement. *Tiling* [RT99, CM95] is a loop transformation that partitions the iteration space into sets, called *tiles* and executed atomically. We only consider programs with rectangular iteration spaces and rectangular tiling. This is explained in more detail in Section 2.1.1. 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.

Class of programs considered We use lowercase for problem dimensions (i, j, k) for matrice multiplication, 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.

In a similar fashion, as Halide [RBA+13], our code generation takes two different inputs: the first one is a semantic description of our computation graph. The second is an implementation scheme description that specifies how vectorization, unrolling, tiling, and other implementation choices are done.

3.1 Computation graph

This part of the specification describes the semantics of the program we want to implement. As we said, we assume a rectangular iteration space. We first declare the dimensions we are going to use, then the tensors with their associated dimensions. Then a few primitives are provided such as *Load*, *Store*, $\times$ or $+$ which allow specifying a *graph of computation*. Input nodes are *Load* or constants, output nodes are *Store* and intermediate nodes are arithmetic operations. This specifies a basic block over which all dimensions are iterated implicitly.

Example We will now show the canonical example of matrix multiplications. We first declare **three dimensions** i, j, and k and **three tensors** A, B, and C with respectively indexing dimensions (i, k), (k, j), and (i, j):

```
let dim_gen = Dim.fresh_gen ()
let i_dim = dim_gen ~name:"i" ()
let j_dim = dim_gen ~name:"j" ()
let k_dim = dim_gen ~name:"k" ()
let a = Tensor.make ~name:"A" [i_dim; k_dim]
let b = Tensor.make ~name:"B" [k_dim; j_dim]
let c = Tensor.make ~name:"C" [i_dim; j_dim]
```

We define our basic block as follows:

```
let c_ij = Load (c, [index i_dim; index j_dim])
and a_ik = Load (a, [index i_dim; index k_dim])
and b_kj = Load (b, [index k_dim; index j_dim]) in
let contract = Add (c_ij, Mul (a_ik, b_kj)) in
Store (contract, c, [index i_dim; index j_dim])
```

This reads as:

1. . Load $C[i, j]$ into c_{ij};
2. . Load $A[i, k]$ into a_{ik};
3. . Load $B[k, j]$ into b_{kj};
4. . Compute $c_{ij} + a_{ik} * b_{kj}$;

5. . Store the result into $C[i, j]$.

By default, accesses to tensors are linearized. For example, $Load(c, [indexi; indexj])$ is translated at code generation time into : $C[i * J + j$, with J the size of dimension j. To support the specific access pattern of convolution, an additional ad-hoc constructor is added for tensors. We can "join" two dimensions together inside a tensor :

```
let input = Tensor.make ~name:"Input"
                        [Join(w, r), Join(h, s), c]
```

Then an access to tensor *Input* such as :

```
let input_ld = Load(input,
                    [index w, index r,
                     index h, index s, index c]
                   )
```

will be translated as $input[(w + r) * C * (H + S - 1) + (h + s) * C + c]$.

3.2 Atoms

Our code generator is driven by a so-called *optimization scheme*. An optimization scheme is a list of *atoms* that describe the layered structure of the generated code, *from the outermost loop inwards*. We first present briefly all of these atoms and their semantics. The next section will illustrate their use by example.

- R_d inserts the outer loop along dimension d. This loop will iterate over the outer-level tiles along d. the problem size D should be divided by the sizes of these tiles. Besides, R_d may appear at most once for a given dimension d.
- 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 $\mathsf{V}_\bullet$.
- $\mathsf{T}_{\alpha,d}$ 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.
- $\mathsf{U}\lambda_d$ inserts a parametrized unroll: there will be multiple instantiations of this unroll with different values. This atom assumes the use of $\lambda_{\mathsf{seq}_d}.[\ell]$ at an outer position.
- $\lambda_{\mathsf{seq}_d}.[\ell]$, 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 atom introducing these tiles. This specifier generates non-perfectly nested tiles, composing

microkernels whose sizes do not individually divide the size of a given dimension. For example, splitting a dimension y of size $Y = 34$ into two non-equal parts 22 and 12 with $\ell = [(2, 11), (1, 12)]$ fulfills the divisibility constraint (no partial tiles) while involving high-performance microkernels of size 11 and 12 along y.

- $\mathsf{Texct}_{\alpha,d}$ inserts a tile loop that has a footprint of exactly α, regardless of multiplicity. It implies that the inner tile loops on dimension d have a parametrizable size, which is enabled either by the $\mathsf{Tvar}_{\alpha,d}$ or the $\mathsf{ExternalCall}_{name,\alpha,d}$ atoms.
- $\mathsf{ScalP}[d_0, d_1, ...]$ takes all tensor accesses inside the current tile that do not include any dimension in $[d_0, d_1, ...]$, and move these accesses out of the loop.
- Pack_A introduces a temporary buffer that corresponds to the subset of A that is accessed in the inner levels of the loop nest at this level of the scheme.
- $\mathsf{PackT}_A\,[ell]$ is similar to Pack_A but in addition performs a permutation of dimensions over the buffer specified by ell.
- $\mathsf{Tvar}_{\alpha,d}$ inserts a tile loop that has a variable bound, which in turn allows using $\mathsf{Texct}_{\_,\_}$.
- $\mathsf{ExternalCall}_{name,\alpha,d}$ inserts a call to an external function of name $name$, which is flexible on dimension d, and this call is done with a size α. This is useful when we want to make use of a handcrafted microkernel.
- $\mathsf{Tpar}_{\alpha,d}$ has the same semantic as $\mathsf{T}_{\alpha,d}$, but is executed in parallel
- $\mathsf{Tfused}_{[(d1,i1),(d2,i2)...]}$ iterates on all dimensions $[d1, d2...]$ in a unique loop and executes it in parallel.

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 [VZvdG15] microkernel for single precision floating point number 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 ($i = 6, j = 16 =$ `2 vectors`$, k = n_k$) 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) [VZvdG15]. 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.

```
1  float scal_0 ,scal_1 ,scal_2 ,scal_3 ,scal_4;
2  for (k = 0; k < K; k += 1){
3    for (j = 0; j < J; j += 1){
4      for (i = 0; i < I; i += 1){
5        scal_1 = C[J * i + j];
6        scal_3 = A[K * i + k];
7        scal_4 = B[J * k + j];
8        scal_2 = scal_3 * scal_4;
9        scal_0 = scal_1 + scal_2;
10       C[J * i + j] = scal_0;
11     }
12   }
13 }
```

Figure 3.1: $[\mathsf{R}_k, \mathsf{R}_j, \mathsf{R}_i]$

```
1  for (k = 0; k < K; k += 1){
2    for (j = 0; j < J; j += VECSIZE){
3      for (i = 0; i < I; i += 1){
4        scal_0 = A[K * i + k];
5        vec_1 = _mm256_set1_ps(scal_0);
6        vec_2 = _mm256_load_ps(&B[J * k + j]);
7        vec_3 = _mm256_load_ps(&C[J * i + j]);
8        vec_0 = _mm256_fmadd_ps(vec_1, vec_2, vec_3);
9        _mm256_store_ps(&C[J * i + j], vec_0);
10     }
11   }
12 }
```

Figure 3.2: $[\mathsf{R}_k, \mathsf{R}_j, \mathsf{R}_i, \mathsf{V}_j]$

3.3 Examples

In this section, we show a few examples to illustrate the use of our framework and what kind of code it produces. All schemes will be implementing a matrix multiplication.

To illustrate a possible **implementation** scheme, we start with the non-tiled version of the code that is defined as follows: $[\mathsf{R}_k; \mathsf{R}_j; \mathsf{R}_i;]$

This defines three loops that respectively iterate on dimensions `i`, `j`, and `k`. Those loops are listed from outer to inner. Here loop-type R stands for "remaining", that is, the loop traverses the full dimension. This produces the C code found in 3.1.

Our framework also provides support for vectorization along a dimension. This can be expressed using loop type V as in the following scheme:

This would produce the C code in Figure 3.2 made up of four loops where the inner one (fully unrolled) is vectorized along dimension j (observe the stride $VECSIZE$ on the remaining loop along j):

We can also unroll our loop along a dimension as in Figure 3.3.

We can tile dimensions, which creates subloops, as in Figure 3.4.

```
for (k = 0; k < K; k += 2){
  for (j = 0; j < J; j += 1){
    for (i = 0; i < I; i += 1){
      scal_1 = C[J * i + j];
      scal_3 = A[K * i + k];
      scal_4 = B[J * k + j];
      scal_2 = scal_3 * scal_4;
      scal_0 = scal_1 + scal_2;
      C[J * i + j] = scal_0;

      scal_7 = A[K * i + k + 1];
      scal_8 = B[J * (k + 1) + j];
      scal_6 = scal_7 * scal_8;
      scal_5 = scal_1 + scal_6;
      C[J * i + j] = scal_5;
    }
  }
}
```

Figure 3.3: $[\mathsf{R}_k, \mathsf{R}_j, \mathsf{R}_i, \mathsf{U}_{2,k}]$

```
for (k0 = 0; k0 < K; k0 += 32){
  for (j = 0; j < J; j += 1){
    for (i = 0; i < I; i += 1){
      // Tiling dim k by 32
      for (k = k0 ; k < k0 + 32; k += 1){
          scal_1 = C[J * i + j];
          scal_3 = A[K * i + k];
          scal_4 = B[J * k + j];
          scal_2 = scal_3 * scal_4;
          scal_0 = scal_1 + scal_2;
          C[J * i + j] = scal_0;
      }
    }
  }
}
```

Figure 3.4: $[\mathsf{R}_k, \mathsf{R}_j, \mathsf{R}_i, \mathsf{T}_{32,k}]$

```
for (k0 = 0; k0 < K; k0 += 16){
  for (i = 0; i < I; i += 1){
    //Pack B into B0
    for (kg0 = k0, kl0 = 0;
      kg0 < k0 + 16;
      kg0 += 1, kl0 += 1){
        for (jall = 0; jall < J; jall += VECSIZE){
          vec_0 = _mm256_load_ps(&B[J * kg0 + jall]);
          _mm256_store_ps(&B0[J * kl0 + jall], vec_0);
        }
    }
    //Tiling dim k by 16
    for (k = k0, kp_0 = 0;
      k < k0 + 16;
      k += 1, kp_0 += 1){
        for (j = 0, jall = 0; jall < J; j += 1, jall += 1){
          scal_1 = C[J * i + j];
          scal_3 = A[K * i + k];
          scal_4 = B0[J * kp_0 + jall];
          scal_2 = scal_3 * scal_4;
          scal_0 = scal_1 + scal_2;
          C[J * i + j] = scal_0;
        }
    }
  }
}
```

Figure 3.5: $[\mathsf{R}_k, \mathsf{R}_i, \mathsf{Pack}_B, \mathsf{T}_{16,k}, \mathsf{R}_j,]$

We can tell the generator to "pack" a tensor: we create a temporary buffer in which we copy all elements such that they are accessed contiguously in the inner loop nest. This is shown in Figure 3.5.

Scalar promotion is meant to move invariant code outside of a set of loops. $Scalpromd_i$ moves all accesses that are orthogonal to all d_i outside of this level of the loop nest. This is shown in Figure 3.6. As it could in most cases be automatically apply we will often omit it from future scheme descriptions.

We still need to show how the $\mathsf{U}_{\alpha,d}$ and $\lambda_{\mathsf{seq}_i}.[(12,6),(8,7)]$ work. Assume for example a matrix-multiplication of size $I \times J \times K = 128 \times 128 \times 64$. We will implement this operation by combining microkernels of size 6 and 7. The rationale of this choice is explained in more detail in Section 5.3. μkernel_gemm$_{6,16}$ is a shorthand for a basic block unrolled 6 times over dimension i and unrolled and vectorized 16 times over dimension j. 128 is not divisible by 6, but $128 = 12 \times 6 + 8 \times 7$, and efficient code can be obtained using the following scheme:

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

which leads to the loop structure shown in Figure 3.7.

We will explain how the $\mathsf{Texct}_{\alpha,d}$, $\mathsf{ExternalCall}_{name,\alpha,d}$ and $\mathsf{Tvar}_{\alpha,d}$ work in a specific Section 3.5. This gives a good overview of the possible use of the library. We will now give some insights into the implementation.

```
for (k0 = 0; k0 < K; k0 += 32){
  for (j = 0; j < J; j += 1){
    for (i = 0; i < I; i += 2){
      // accesses to C are the only one
      // that are independent of k
      c0 = C[J * i + j];
      c1 = C[J * (i + 1) + j];
      // Tiling dim k by 32
      for (k = k0; k < k0 + 32; k += 1){
          scal_1 = c0;
          scal_3 = A[K * i + k];
          scal_4 = B[J * k + j];
          scal_2 = scal_3 * scal_4;
          scal_0 = scal_1 + scal_2;
          c0 = scal_0;

          scal_9 = c1;
      scal_7 = A[K * (i + 1) + k];
          scal_8 = B[J * k + j];
          scal_6 = scal_7 * scal_8;
          scal_5 = scal_9 + scal_6;
          c1 = scal_5;
      }
      C[J * i + j] = c0;
      C[J * (i + 1) + j] = c1;
    }
  }
}
```

Figure 3.6: $[\mathsf{R}_k, \mathsf{R}_j, \mathsf{R}_i, \mathsf{ScalP}[k], \mathsf{T}_{32,k}, \mathsf{U}_{2,i}]$

```
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 3.7: Microkernel composition example.

3.4 Code generation algorithm

This section will briefly describe the generation algorithm. The principle is simple: we have a graph-like intermediate low-level representation of a loop nest. we scan the scheme from inner to outer, building our loop nest graph progressively. We maintain a structure that holds some invariants that have to be checked at each level, and also various information that is updated with each new atom.

In a nutshell, we want to ensure the following properties :

- Vectorization is allowed only once, at the very end of a scheme
- If a dimension is vectorized, then it must be the innermost dimension for all tensors it accesses
- Unroll atoms can appear in the last position or before a Vectorisation atom but should be placed after any other atoms

To build the code at each level, we also pass the following information from one level to another :

- for each dimension, the footprint at a given level
- for each dimension, the current index to use (for example for dimension i, this would be i, i_0, i_1...).
- for each tensor, the size and name of all intermediate buffers used for packing.

3.4.1 Vectorization and Unroll - Generating a basic block

Now we describe the low-level structure that represents the code structure. At the innermost level, we have a structure representing a basic block as a directed graph of instructions. The nodes of the graph are instructions, while the edges are data dependencies.

For example, in Figure 3.8 we give the representation of the basic block of a matrice multiplication.

This structure is a *Directed Acyclic Graph* which grows when we unroll it twice on dimension j, as we can see in Figure 3.9.

Vectorization is represented as a special basic node as seen in Figure 3.10. Broadcast is an operation that initializes all elements of a vector with a scalar value.

Now we describe the rules that guide the code generation at the inner level :

- **Vectorization V_d:** Considering the definition of the computation described in Section 3.1, 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.

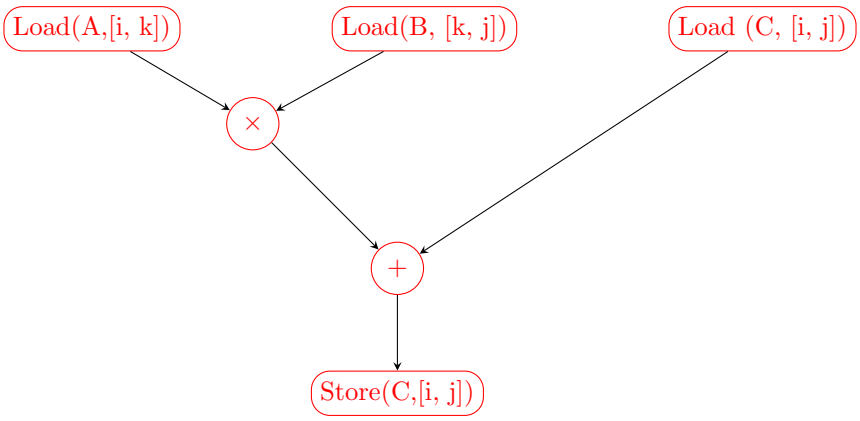


Figure 3.8: DAG representation of the Basic block of a matrix-multiplication

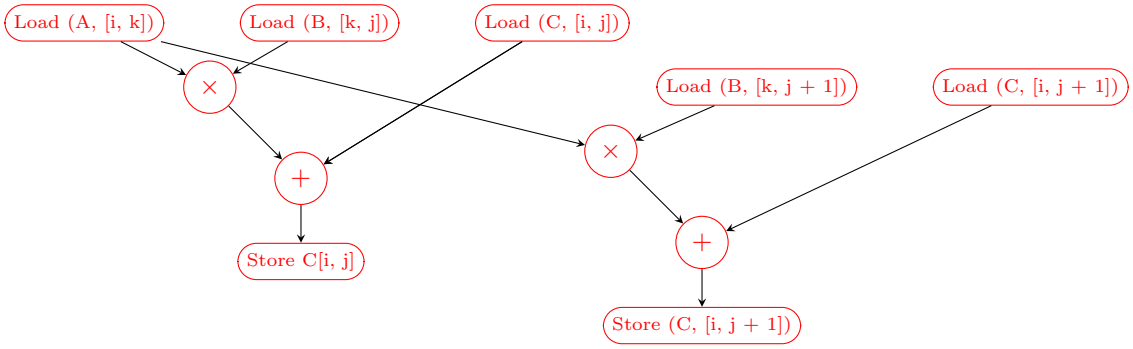


Figure 3.9: Matrix multiplication unrolled twice on dimension j

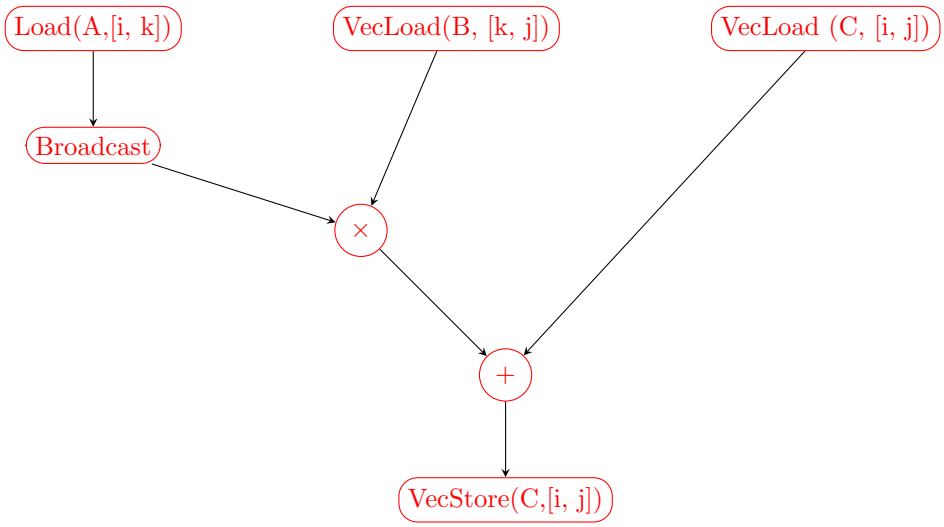


Figure 3.10: Matrix multiplication vectorized on dimension j

- $\mathsf{Op}\,(x, y)$ 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 either both true or both false

The C code uses Intel intrinsics to manipulate vectors. *Intrisics* are special C functions that are mapped directly to an assembly instruction, such as

```
__m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c)
```

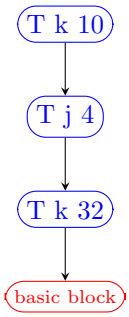
This operation does an vector-multiplication of b and c before accumulating the result nto a.

- **Unroll $\mathbf{U}_{k,d}$:** Unroll the computation over the d dimension k times by traversing the instruction DAG and duplicating any instruction that accesses dimension d, along with any instruction that depends on an instruction that accesses d (recursively)

3.4.2 Tiling loops above the basic block

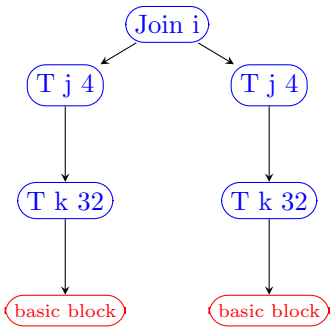
At the outer level, we represent the loop nest as a tree whose leaves are basic blocks as we just described and whose nodes are loops. These nodes are parametrized by a dimension, an index on this dimension, and a size. See an example in Figure 3.11 where the size of the basic block has been set to i = 6 and j = 16 .

This tree looks like a simple list (every node has one single child), and it is until one brings in the lambda atom. In this case, there is an additional type of node that joins two loop nests. We can see an example in Figure 3.12.




```
1  for (int k0 = 0; k0 < 10 * 32; k0 += 32) {
2    for (int j = 0; j < 4 * 16 ; j += 16) {
3      for (int k = k0; k < k0 + 32 ; k += 1) {
4      <basic block : i=6; j=16; k=1>
5      }
6    }
7  }
```

Figure 3.11: Loop nest Graph. DAG on the left and corresponding code on the right




```
1  for (int i0 = 0; i0 < 42; i0 += 6) {
2    for (int j = 0; j < 4 * 32 ; j += 32) {
3      for (int k = 0; k < 32 ; k += 1) {
4      <basic block : i=6; j=32; k=1>
5      }
6    }
7  }
8  for (int i0 = 42; i0 < 106; i0 += 8) {
9    for (int j = 0; j < 4 * 32 ; j += 32) {
10     for (int k = 0; k < 32 ; k += 1) {
11     <basic block : i=8; j=32; k=1>
12     }
13   }
14 }
```

Figure 3.12: Loop nest Graph with lambda

3.4.3 Handling unknown parameters during code generation: the Lambda atom

Until now all schemes we have seen represent a perfectly nested loop. This fits nicely with our representation as a list of atoms, each one of these atoms representing more or less a given level in the loop nest. However these basic atoms impose a restriction on the shape of the code: for a given dimension d, each size of atoms iterating on this dimension should be a divisor of the size of d. For example, in the scheme :

$$[\mathsf{T}_{\alpha_2,j}, \mathsf{T}_{\gamma_1,k}\mathsf{T}_{\alpha_1,j}, \mathsf{T}_{\beta_1,i}, \mathsf{T}_{\gamma_0,k}, \mathsf{U}_{\beta_0,j}, \mathsf{U}_{\alpha_0,j}, \mathsf{V}_j]$$

, we have the following constraints on α_i, β_i and γ_i :

$$\begin{cases} \alpha_2 \times \alpha_1 \times \alpha_0 \times vecsize = J \\ \beta_1 \times \beta_0 = I \\ \gamma_1 \times \gamma_0 = K \end{cases} \tag{3.1}$$

where I, J and K are the sizes of dimensions i, j and k. This can be a problem if one of the dimensions has a size that is not easily divisible (for example if I is a big prime).

We handle this problem by the introduction of atoms $\mathsf{U}\lambda_d$ and $\lambda_{\mathsf{seq}_d}.[ell]$. We call this construction Lambda for reasons we will explain later.

In a nutshell, we want to express a "beyond perfect" loop nest, which is a set of loops that are not related by a strict inclusion relation, but still retain a "correct by construction" semantic for our specification. At some level in the code structure, there will be two or more loops sequenced with each other. This introduces ambiguity to what "inner to outer" even means at this point. We illustrate that with an example. First, say that we want to have two levels of tiling of size α_0 and α_1 on dimension i that is of size 43. If we keep the constraint of divisibility explained above, we have the constraint that $\alpha_1 \times \alpha_0 = 43$, that is :

$$\begin{cases} \alpha_1 = 43 \\ \alpha_0 = 1 \end{cases} \tag{3.2}$$

The solution we propose is to express 43 as $2 \times 11 + 3 \times 7$. Here is the corresponding scheme :

$$[\lambda_{\mathsf{seq}_i}.[(2,11);(3,7)]\,;\mathsf{T}_{32,k};\mathsf{U}\lambda_i]$$

which generates the code in Figure 3.13. The point is that $\mathsf{U}\lambda_i$ introduces a placeholder for the factor of unrolling on dimension i. This placeholder is then instantiated twice with the atom $\lambda_{\mathsf{seq}_i}.[(2,11);(3,7)]$, first with value 11 (which yields a basic block unrolled by a factor of 11) and then 7. The first instantiation is iterated two times, and the second three times. This process of abstraction/instantiation can be thought of as introducing a lambda function that takes as argument the unroll factor and then returns a loop nest with the placeholder filled with the argument, hence the name.

```
for (int i = 0; i < 22; i+=11)
  for (int k = 0; k < 32; k++)
    <basic block: i=11; j=1; k=1>
for (int i = 22; i < 43; i+=7)
  for (int k = 0; k < 32; k++)
    <basic block: i=7; j=1; k=1>
```

Figure 3.13: First example of a combination of loops

```
for (int i = 0; i < 7 * 4; i+=4)
  for (int j = 0; j < 14 * 8; j+=8)
   <basic block: i=4; j=8; k=1>
for (int i = 0; i < 7 * 4; i+=4)
  for (int j = 14 * 8; j < 12 * 8 + 12 * 6 ; j+=6)
   <basic block: i=4; j=6; k=1>
for (int i =  7 * 4; i < 7 * 4 + 3 * 5; i+=5)
  for (int j = 0; j < 14 * 8; j+=8)
  <basic block: i=5; j=8; k=1>
for (int i =  7 * 4; i < 7 * 4 + 3 * 5; i+=5)
  for (int j = 14 * 8; j < 12 * 8 + 12 * 6 ; j+=6)
  <basic block: i=5; j=6; k=1>
```

Figure 3.14: Multiple lambdas

Using this abstraction allows specifying lambdas on multiple dimensions while retaining correctness easily. Let's take the following example: we want to implement a matrix multiplication with $i = 43$ and $j = 180$. For some reason we want to split both dimension i and j. If we want to split on both dimensions i and j in such a way that i is iterated first 7 times with an unrolled factor of 4, then 3 times with an unroll factor of 5, and j is iterated 14 times with an unrolled factor of 8 then 12 times with an unrolled factor of 6, we have to implement the cartesian product of $i \in [(7,4),(3,5)]$ and $j \in [(12,8),(14,6)]$.

The code in Figure 3.14 can be represented by the following scheme :

$$[\lambda_{\mathsf{seq}_i}.[(7,4),(3,5)]\,;\lambda_{\mathsf{seq}_j}.[(14,8),(12,6)]\,;\mathsf{U}\lambda_i;\mathsf{U}\lambda_j]$$

The point is that we avoid the redundancy of writing the 4 different loops by hand, which would be both tedious and error-prone. In terms of implementation, the main problem is that in our inner-to-outer scheme, just after the use of $\mathsf{U}\lambda_d$ we do not have the unroll parameter yet. Thus we need a way to go on our generation scheme despite a missing parameter.

We could have handled this problem by implementing multiple passes on the scheme instead of a single inner-to-outer one. But we chose an alternative path: we leveraged the fact that our implementation is done in a functional programming language, which by definition offers great support for function definition, application, and composition. So whenever an atom that requires a parameter is encountered, we introduce a function that takes this parameter

and returns the code structure associated. This is conceptually equivalent to introducing a free "unroll variable" that will be instantiated later.

When we want to introduce a $\mathsf{U}\lambda_d$ on dimension d, the size of the unroll is not known yet and will be instantiated later multiple times. So we take the current state of the code generation graph, which can be either in a "concrete" state (all unrolling factors are known) or in a "pending" state: we are still waiting for one or more unrolling factors. In the "concrete" case, we replace the concrete graph with a new function f' that takes the unrolling factor as parameter and returns the graph unrolled of this factor. In the "pending" case, we compose the previous function f with a new function f' similar to the one we just described.

Then these "pending" states will be eliminated by application of a $\lambda_{\mathsf{seq}_d}.[[...]]$ atom. As the composition we described can yield a nested structure, in this case, we find recursively at which point the unrolling factor with the corresponding dimension d was introduced, to then apply the corresponding function to all parameters in the list. This yields a list of loop nests $[ln1, ln2..]$.

Then a helper function *sequence_loops* is used. It takes a list of pair of int and loop nests and returns the loop nest that consists of sequencing each of these loop nests surrounded by a loop whose size is the given integer.

For example the code corresponding to :

```
sequence_loops( [(2, ln1), (8, ln2), (9, ln3)])
```

would be :

```
1 for (int i = 0; i < 2; i++)
2   <ln1>
3 for (int i = 2; i < 10; i++)
4   <ln2>
5 for (int i = 10; i < 19; i++)
6   <ln3>
```

This introduction/elimination scheme may seem complex, but it allows us to handle nicely cases where we miss a parameter to continue our generation in a general way. It avoids back-and-forth passes to find all required parameters and guarantees by construction that transformations will be applied in the right order. Also, this scheme is general enough to compose different cases where some information is not available now.

3.5 Partial Tiles

Until now, all atoms we described are constrained to respect a divisibility hypothesis, escape from the small escape hatch enabled by $\mathsf{U}\lambda_d$ and $\lambda_{\mathsf{seq}_d}.[ell]$. We will show later that this setting is enough to get all the needed performance. However, assuming we know exactly (for example with the help of a model) which sizes we want at every level of our loop nest, while Lambda alone would suffice to generate a code corresponding to a scheme given ahead of time, the split would make code size increases exponentially.

Other techniques such as padding exist to fulfill divisibility constraints and to enforce perfectly nested loops. At the lower level, it is possible to write a

kernel that can handle multiple sizes of submatrices. For example, we can build a microkernel for matrice multiplication that can handle any size for i in the range $[1, 16]$, and with j and k fixed. Such a technique is described later in Section 5.3.1. Here we focus on the generation of the call to this microkernel, assuming that we have one available.

$\mathsf{ExternalCall}_{name,\alpha,d}$ is the atom for calling an external code. We omitted a few parameters for brevity that are necessary to make the call valid. In addition to the ones we mentioned earlier (name of the function, flexible dimension, call size) the actual atom takes the following arguments :

- Pointers to each tensor in use
- A stride value for each tensor/dimension pair

Providing flexibility without compromising too much on performance at the lowest level is the most difficult task. Indeed, as explained in Section 2.1.1 we need to unroll the inner basic block to remove a part of the control flow. We will explain in detail in Section 5.3.1 how we can achieve to do that, but as far as we know we have to rely on architecture-specific instructions such as masked vector loads and stores that are not available on all platforms. On the outer level, there are simpler and more portable solutions though.

Indeed, when loops are expressed as traditional C for loops and not unrolled, it suffices to use a variable expression as a loop bound instead of a constant and to make this variable take the appropriate value at runtime. We recall the three kinds of atoms that handle that :

- $\mathsf{Texct}_{\alpha,d}$ inserts a tile loop that has a footprint of exactly α, regardless of multiplicity. It implies that the inner tile loops on dimension d have a parametrizable size, which is enabled either by the $\mathsf{Tvar}_{\alpha,d}$ or the $\mathsf{ExternalCall}_{name,\alpha,d}$ atoms.
- $\mathsf{Tvar}_{\alpha,d}$ inserts a tile loop that has a variable bound, which in turn allows using $\mathsf{Texct}_{\_,\_}$.
- $\mathsf{ExternalCall}_{name,\alpha,d}$ inserts a call to an external function of name *name*, which is flexible on dimension d, and this call is done with a size α. This is useful when we want to make use of a handcrafted microkernel.

$\mathsf{Tvar}_{\alpha,d}$ introduces a loop that uses an expression instead of a constant as a bound, and $\mathsf{Texct}_{\alpha,d}$ sets dynamically this expression to the right value. 3.15 shows a use example.

In addition, there is a more general case $\mathsf{Texctg}_{\alpha,d}$ defined as follows:

- $\mathsf{Texctg}_{\alpha,d}$ introduces a loop that both uses a variable as bound and defines a variable for the inner loops, see Figure 3.16.

These atoms are not independent of each other but are related by a few properties. Namely, introducing a $\mathsf{Texct}_{\alpha,d}$ at some level makes it mandatory for the next atom on the same dimension to be either a $\mathsf{Tvar}_{\alpha,d}$ or a $\mathsf{Texctg}_{\alpha,d}$

```
1  for (int i0 = 0; i0 < 120; i0 += 32) {
2    for (int j = 0; j < 64; j++) {
3      int var_i = MIN(32, 120 - i0);
4      for (int i = i0; i < var_i; i++) {
5        for (int k = 0; k < 32; k++) {
6            C[i, j] += A[i, k] * B[k, j];
7        }
8      }
9    }
10 }
```

Figure 3.15: $[\mathsf{Texct}_{120,i}, \mathsf{T}_{64,j}, \mathsf{Tvar}_{32,i}, \mathsf{T}_{32,k}]$

```
1  for (int i1 = 0; i1 < 260; i1 += 120) {
2    int var_i0 = MIN(120, 260 - i1);
3    for (int j = 0; j < 64; j++) {
4      for (int i0 = i1; i0 < i1 + var_i0; i0+=32) {
5        int var_i = MIN(32, 120 - (i0 - i1));
6        for (int i = i0; i < var_i; i++) {
7          for (int k = 0; k < 32; k++) {
8              C[i, j] += A[i, k] * B[k, j];
9          }
10       }
11     }
12   }
13 }
```

Figure 3.16: $[\mathsf{Texct}_{260,i}, \mathsf{T}_{64,j}, \mathsf{Texctg}_{120,i}, \mathsf{Tvar}_{32,i}, \mathsf{T}_{32,k}]$

. For the same reason, a $\mathsf{Texctg}_{\alpha,d}$ also imposes the next atom on the same dimension to be either a $\mathsf{Tvar}_{\alpha,d}$ or a $\mathsf{Texctg}_{\alpha,d}$.

Additionally, a few rules make sure that these atoms interact correctly with the other ones. By default, we impose divisibility between atoms $\mathsf{U}_{\alpha,d}$ and $\mathsf{T}_{\alpha,d}$.

Therefore, using either of these two atoms does not preclude the use of a partial tile, but it does impose that the granularity of a tile becomes the size of the tile defined by the inner atoms. For example in : $[\mathsf{Texct}_{34,i}, \mathsf{Tvar}_{13,i}, \mathsf{T}_{2,i}, \mathsf{U}_{3,i}]$ The suffix $[\mathsf{T}_{2,i}, \mathsf{U}_{3,i}]$ imposes that any tile over it be a multiple of 6. Therefore this scheme is not legal since neither 13 nor 34 are multiple of 6. However, $[\mathsf{Texct}_{48,i}, \mathsf{Tvar}_{18,i}, \mathsf{T}_{2,i}, \mathsf{U}_{3,i}]$ is legal, even if 48 is not a multiple of 18, because we respect the inner divisibility constraint.

3.6 Packing

Packing is a strategy that consists in doing a reshuffling of data on the fly that can in some situations improve memory accesses performance. This improvement can come in a few different ways. First, accessing smaller, more compact buffers make it easier to benefit from spatial locality. Second, accessing data in a linear fashion can work better with features such as hardware prefetching.

The idea behind packing is that whenever a packing atom on a tensor T is encountered, we create a buffer of the size of the footprint of the part of the inner partial scheme on said tensor T.

We have implemented two different atoms that enable this strategy :

- Pack_T where T is a tensor
- $\mathsf{PackT}_T\,[dl]$ where T is also a tensor dl is a list of dimensions that specifies the layout of the new intermediate tensor we want to build. dl should contain exactly the dimensions of T, but can be in any order (the rightmost dimension in the list is the innermost one in the target tensor).

The first one is the simplest. It consists only in compacting the data in a new buffer without any further reshuffling. That is, the packed tensor is a tile (or a footprint) of the original tensor.

The second one adds the possibility of reordering the dimensions inside the packed tensor. This can be useful for example if the dimension we want to vectorize is not the innermost one in the source tensor. In this case, we call a handwritten shuffling routine inspired by [SSB17] which relies on vectorized shuffle instructions.

In terms of the code generation algorithm, there are two things to take into account:

1. replacing every access inside the inner partial scheme with the right tensor and indexes
2. generating the code corresponding to the copy of the original tensor into the temporary buffer (and the symmetric copy after computation is done if we packed an output tensor)

```
1  for (int k0 = 0; k0 < 512; k0 += 128)
2    for (int j0 = 0; j0 < 256; j0 += 32)
3      for (int i = 0; i<128; i+=4){
4        for (int kp = 0; kp < 128; kp += 16)
5          for (int ip = 0; ip < 4; ip+=1)
6            vec_load(&A0[ip * 128 + kp],
7                 &A[(i + ip) * 512 + (k0 + kp)]);
8        for (int k = k0; k < k0 + 128; k++) {
9        //All accesses to A replaced by accesses to A0
10         <basic block : 4 x i, 32 x j, 1 x k >
11       }
12     }
```

Figure 3.17: $[\mathsf{T}_{8,j}, \mathsf{T}_{32,i}, \mathsf{Pack}_A, \mathsf{T}_{256,k}, \mathsf{U}_{4,i}, \mathsf{U}_{2,j}, \mathsf{V}_j]$

The first one is a simple substitution inside the instruction DAG. The second one is close to the process described in [SSB17].

We will now develop an example that shows how this is done in a concrete case. Imagine we are doing a matrix multiplication and we have the following scheme : $[\mathsf{T}_{8,j}, \mathsf{T}_{32,i}, \mathsf{Pack}_A, \mathsf{T}_{128,k}, \mathsf{U}_{4,i}, \mathsf{U}_{2,j}, \mathsf{V}_j]$

We assume that A is accessed in the following way $A[i,k]$, that is, with k as the innermost dimension. The important part is to determine the footprint of the partial tile that is inside the Pack_A atom: $[\mathsf{T}_{256,k}, \mathsf{U}_{4,i}, \mathsf{U}_{2,j}, \mathsf{V}_j]$

Assuming a vector size of 16 single precision floating point numbers (as is the case on AVX512), we can deduce that the footprint of this partial tile has the following size:

$$\begin{cases} i : 4 \\ j : 32 \\ k : 128 \end{cases}$$

Thus as A is accessed by dimensions i and k we want to define a temporary tensor of size $4 \times 32 = 128$. We will call this new buffer $A0$.

Then we need to generate the code that copies the necessary data from the original tensor into this buffer. We generate a vectorized copy. This is legal when both the original and target tensors' inner dimensions have a footprint (for the partial tile we consider) divisible by the size of a vector.

The final code is shown in Figure 3.17.

When we have to reorder the layout, a simple vectorized copy is not enough anymore and shuffling has to be done. Done in a naive way, this reordering would have a prohibitive cost. Fortunately, modern architectures provide hardware shuffling instructions that perform different kinds of recombination of vectors. Thanks to these instructions, it is possible to implement a `vector size` $\times$ `vector size` matrix transposition in an efficient way. Springer and al. [SSB17] explains how to do this, and how to build around it an efficient reshuffling of any tensor under the condition that the inner dimensions of both the source tensor and the target tensor are multiples of the size of a

```
for (int k0 = 0; k0 < 512; k0+=128){
  for (int j = 0; j<256; j0+=32){
    for (int i = 0; i<128; i+=4){
      for (int kp = 0; kp<128; kp+=16){
        for (int jp = 0; jp<32; ip+=16){
          shuffle_16x16(
            // Target address
            &B0[kp * 32 + jp],
            // stride of k in B0
            32,
            &B[(j + jp) * 256 + (k0 + kp)],
            // stride of j in B
            256
          );
        }
      }
    for (int k = k0; k < k0 + 128; k++) {
        //All accesses to B replaced by accesses to B0
        // and indexes replaced accordingly
        <basic block : 4 x i, 32 x j, 1 x k >
      }
    }
  }
}
```

Figure 3.18: $[\mathsf{T}_{8,j}, \mathsf{T}_{32,i}, \mathsf{Pack}_B[k,j], \mathsf{T}_{128,k}, \mathsf{U}_{4,i}, \mathsf{U}_{2,j}, \mathsf{V}_j]$

vector. Therefore, we followed these principles to build a handwritten routine for `vector size` $\times$ `vector size`, which is then called in a loop nest that does the transposition. Figure 3.19, done by Valentin Trophime - who worked on the evaluation of packing in his internship - illustrates how an 8x8 transposition works.

As an example, let us take the matrix multiplication but assume that tensor $B[j,k]$ is stored with k as the inner dimension. We want to switch k and j to allow vectorization on j. This yields the code shown in Figure 3.18. The kernel shuffle_16x16 works in a similar way as the kernel illustrated in Figure 3.19

3.7 Work in progress : Parallelism

Until a few months ago we relied on the TVM backend to provide parallel execution. Recently we added primitives to support that natively. This is still a work in progress, but it allows us to make use of a multicore execution. This was joint work with Valentin Trophime, who was an intern at the time. Two additional atoms are provided :$\mathsf{Tpar}_{\alpha,d}$ and $\mathsf{Tfused}_{[(d1,i1),(d2,i2)...]}$.

Recall that $\mathsf{Tpar}_{\alpha,d}$ has the same semantic as $\mathsf{T}_{\alpha,d}$, except that it is executed in parallel : it executes α iterations on dimension d. $\mathsf{Tfused}_{[(d_1,\alpha_1),(d_2,\alpha_2)...]}$ is a loop that iterates on many dimension at the same time. $Tpar\alpha d$ is equivalent semantically to $\mathsf{Tfused}_{[[(d,\alpha)]]}$.

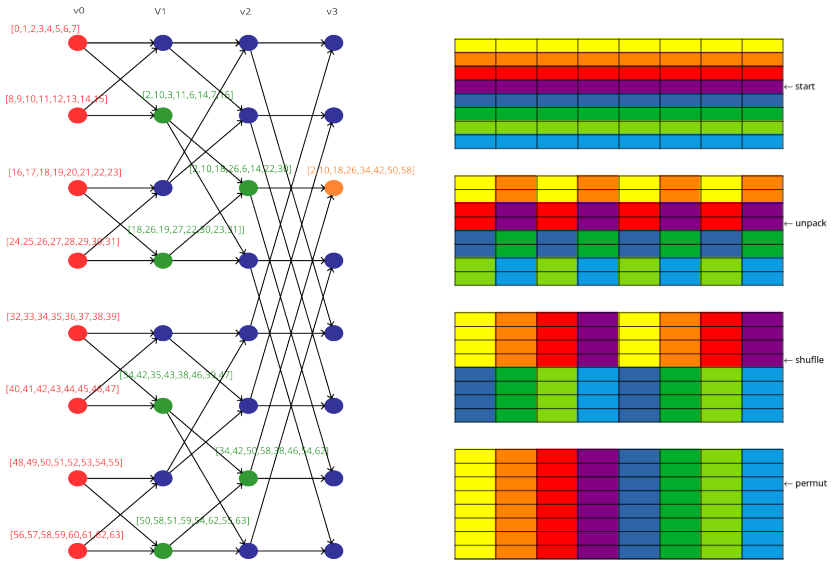


Figure 3.19: Example of transposition

$\mathsf{Tfused}_{[\_]\_}$ is sometimes necessary when no dimension is big enough to offer a sufficient amount of parallelism alone.

These extensions are implemented and semantically correct, but were never leveraged in any space exploration. This is left for future work.

3.8 Conclusion

In this section, we developed a very small Embedded Domain Specific Language that allows us to define an implementation for a given semantic specification in our design space. In the next Section 4 we will detail the experimental platform this code is running on, the way we evaluate its performance, and check its semantic correctness.

Chapter 4

Experimental platform

In this chapter, we describe the experimental setup needed for our work. During the implementation, several challenges were raised. At some point in our experimentations, we faced some instability in our measurement that led us to implement a more flexible and hopefully robust framework to collect performance indicators on our bench suites. This framework proved useful in many ways that we are going to present in this section.

4.1 General Characteristics of the platform

The point of this platform is to debug the code generation, launch easily different benches with varying arguments, and retrieve indicators such as timing (in cycles) but also possibly other counters. There are many parameters one may want to change across runs. Compilers, compiler options, which benches to run, whether to generate a new code or not (we may need to run a handcrafted code for debugging purposes), number of repetitions, and which semantic checks to perform.

4.2 Interface

As we saw in this chapter, apart from the code generation we need to test a variety of diverse configurations, testing different benches (matrix multiplication, convolutions, or tensor contractions) that use different arguments, with different hardware counters, and different semantic checks. In the case of convolution and tensor contractions, we needed to have flexibility on the layout of the tensors too. As we were regularly switching some or all of these parameters, having to change the code manually each time soon proved error-prone and time-consuming. Thus we have developed a framework that offers an API that allows switching these parameters seamlessly. OCaml's powerful type system allows us making sure statically that every call is consistent - for example,

```
type _ bench = Blis: mm bench
             | Mkl:  mm bench
             | XSMM:  mm bench
             | TTiLe_mm:  mm bench
             | TTiLe_conv: conv bench
             | OneDNN:  conv bench
             | TTiLe_tc : tc bench
```

Figure 4.1: type for possible benches

that we never pass the arguments of a convolution to a matrix multiplication function.

The interface of the execution function takes the bench we want to execute (which can be a call to our code generation framework for either convolution or matrix multiplication or a call to an external library or several of these choices), the parameters of the bench (the sizes of the input and output on every dimension, the layout of the tensors, the counters we want to evaluate, the error checking method and some others.

We encoded into the type system some constraints that make sure that the layouts, the parameters, and the benches are consistent (as the structure that represents parameters is different for a convolution, a matrix multiplication, or a tensor contraction - the number of dimensions differ, for example), which allows to check it statically. The trick is to use Generalized Algebraic DataType, a feature that can be found in languages such as OCaml or Haskell, that we will not detail here. More explanations can be found in the OCaml manual [LDF+13].

The nice trick is that this interface enforces statically the consistency of our call, without having to duplicate the code (for example by separating each problem: matrix multiplication, convolution, tensor contraction, and having a distinct interface for each). For example, all benches are statically associated with the kernel they operate on (mm is short for matrix multiplication, conv for convolution, and tc for tensor contraction).

We also have specific types for the dimensions and tensor of each kind of tensor computation that we declare separately, see Figure 4.2. This, in turn, allows us having a distinct type for each kind of scheme (scheme for matrix multiplication, convolution, or tensor contraction) that we do not show here for brevity.

Finally, we have a type for the arguments of each benchmark (namely the sizes of the problem), that we do not show here. That way, we can make sure our *exec* function is not called with mixed arguments (such as a bench for matrix multiplication with a scheme for convolution).

```
val exec: 'a bench -> 'a args -> 'a scheme -> 'a results
```

If we mess up, we end up with a type error instead of a runtime check failure.

```
(* matrice multiplication *)
type dim = I | J | K
type tensor = A | B | C

(* convolution (in another file) *)
type dim = K | C | W | H | R | S
type tensor = Input | Output | Params

(* Tensor contraction (again, in a separate file) *)
type dim = LeftDim of int | RightDim of int | RedDim of int
type tensor = Out | Left | Right

(* This lets us write [T (3, I); U (5, K); V J]
 * and [T (3, C); U (5, H), V K]
 * But rule out [T (3, I); U (5, H)]
 * because it mixes mm dimensions with convolution dimensions
 *)
```

Figure 4.2: types of dimension and tensor for each tensor computation

4.3 Compiler impact

We already discussed our choice of generating C code instead of assembly code in Section 2.1.1. There are two main reasons for that: portability, and leveraging the optimization the compiler can do by itself. The flip side is that we give up some level of control over the final code. This lead to some fragilities, especially on codes that differ too much from typical handwritten code - which can happen when we rely on code generation. For example, we observed some bad behaviors from the register allocation algorithm.

To evaluate the impact of the compiler on our performance, we made sure to test different compilers and different levels of optimization. This in turn encouraged us to improve our code generation when we saw that we relied too much on a given optimization pass - and whether or not the compiler would succeed to apply it.

One example of that is the use of fuse-multiply-add. At first, we were generating code such as (in pseudo-code) :

```
1 // broadcast a value into a vector
2 a = _mm512_set1_ps(A[i, k]);
3 b = _mm512_load_ps(&B[k, j]);
4 c = _mm512_load_ps(&C[i, j]);
5
6 prod = _mm512_mul_ps(a, b);
7 sum = _mm512_add_ps(c, prod);
8 _mm512_store_ps(sum, &C[i,j]);
```

When compiled with -O2 this code was inefficient as it was using two vector

instructions - mul and add - without taking advantage of the existing fused-multiply-add operation that combines both into a single instruction. Compiling with O3 allowed the compiler to detect this deficiency and generate adequate instructions. This makes a huge difference in performance between the code compiled with O2 and O3. As a consequence, we change our code generation to detect these patterns ourselves and generate directly the fused-multiply-add instruction. This change alone allowed the two versions (O2 and O3) to stand on an equal footing.

Another point was the hoisting of memory accesses inside the basic block. Loop-invariant code motion is the operation of factorizing operations that are done repeatedly inside a loop to do it only once. In our case, the accesses on the output tensor (which is called C in the case of matrix multiplication) can be systematically hoisted out of the innermost loop.

For example, see these (semantically equivalent) codes :

```
1  // No hoisting, each iteration of this loop
2  // makes the same accesses to C
3  for (int k= 0; k < 256; k++) {
4    a = _mm512_set1_ps(A[i, k]);
5    b = _mm512_load_ps(&B[k, j]);
6    c = _mm512_load_ps(C[i, j]);
7
8    // sum = a * b + c (vectorized)
9    sum = _mm512_fmadd_ps(a, b, c);
10   _mm512_store_ps(sum, &C[i,j]);
11   }
12
13 // Loop-invariant code motion - accesses to C
14 // are factorized out of the loop
15 _m512 sum  = _mm512_load_ps(C[i, j]);
16 for (int k = 0; k < 256; k++) {
17   a = _mm512_set1_ps(A[i, k]);
18   b = _mm512_load_ps(&B[k, j]);
19
20   // sum = a * b + c (vectorized)
21   sum = _mm512_fmadd_ps(a, b, sum);
22   }
23 _mm512_store_ps(sum, &C[i,j]);
```

This optimization proved to be done only in an unreliable manner by compilers (be it icc, gcc, or clang). This was made obvious by experiments done on the microkernels selection (see Section 5.2). Some microkernels were performing surprisingly badly and by scrutinizing assembly we discovered that the compiler failed to apply this optimization properly in some cases. As a result, we implemented it in our code generation directly. While some microkernels were unaffected (the ones where the optimization has been correctly applied) some others were improved by a factor of 2, from 45% of peak performance to 90% or more.

4.4 Performance counters

In this section, we will describe briefly what tools are available for identifying program bottlenecks. To do that, hardware vendors provide counters that allow monitoring of some part of the CPU behavior at runtime. The most basic of these counters is the cycle counter. However there are many others that monitor cache misses, pipeline stalls, branch prediction misses, etc.

These counters are architecture-dependent. Some architectures provide many more than others, and their exact semantics are not always clear.

There is a library that attempts to provide a unified frontend for at least some common counters on many architectures. This library is called PAPI - for Performance Application Programming Interface [MBDH99]. It is initially intended for C++ but also provides a C API - at the expense of ergonomy, to some degree. As many counters are not available on all architecture, PAPI can only offer a best-effort on portability and still offers a mechanism that checks at runtime if a given counter is available or not. PAPI uses a system of events you have to register. We generate the static array of events we are going to register from the arguments passed to *exec* (see above in Section 4.2).

The counter we used the most was the cycles counter, which in turn let us define a unit of performance that is "the percentage of the peak performance". This corresponds to the ratio of the theoretical minimum number of cycles needed for a given computation over the measured number of cycles. Other counters used are measuring cache misses at different levels, the number of stalls in the pipeline, and micro-operations on a given CPU port. These counters can be accessed simultaneously, with some limitations. For example, only a limited number of hardware counters - 5 in our case - can be used at the same time.

Once these counters were set, it allowed us to dive into the rationale of the performance. Unfortunately, we did not get much success. That is, there was no obvious correlation between cycles and any other counter, apart from, to various degree, cache misses - and even there, the correlation was far from linear. Nevertheless, a large number of counters were tested, which would have been impossible in practice without this framework. As such, the platform was a fruitful investment.

4.5 Semantic Checks

To test the validity of our generated code, performing some kind of semantic check over the generated code is crucial. There were a few unexpected technical challenges in implementing this.

The basic principle is quite simple: we compare the results obtained by either our generated code or our competitors with the results obtained by running a naive implementation of the benchmark we want to test.

The first one dealt with Floating Point arithmetic (lack of) associativity. Floating-point addition is not associative due to its rounding semantic. This should not be a problem for us because our generation schemes never reorder

```
1  float C_inter[I][J][4];
2  // This loop can be done in parallel
3  for (int k0 = 0; k0 < 4; k0++) {
4    for (int j = 0; j < K; j++){
5      for (int i = 0; i < I; i++){
6        float c = 0.;
7        for (int k = k0 * K / 4; k < (k0 + 1) * K / 4; k++) {
8          c += A[i][k] * B[k][j];
9        }
10       C_inter[i][j][k0] = c;
11     }
12   }
13 }
14 for (int j = 0; j < K; j++){
15   for (int i = 0; i < I; i++){
16       C[i][j] = 0.;
17       for (int k0 = 0; k0 < 4; k0++) {
18         C[i][j] += C_inter[i][j][k0];
19     }
20   }
21 }
```

Figure 4.3: Matrix-Multiplication with block reduction

dependent operations relative to each other in convolution or matrix multiplication. Therefore the transformations we apply such as tiling or unrolling do not change the semantics in comparison to a naive implementation. This in turn means that a bit-by-bit comparison to the naive implementation output is a valid semantic check.

The trouble is that our competitors do change the order of dependent operations.

The main reason for that is parallelism. BLIS allows itself to parallelize over a reduction axis. This implies that it will perform a tree reduction that as a side effect change the order of the operations. We illustrate this transformation with an example of a block reduction in Figure 4.3.

As a result, a bit-to-bit comparison of our results with BLIS ones will result in a false-positive - we will report a different result although both computations did morally the same thing. This is a common issue in floating point arithmetic and the usual advice is to check results with an error margin. We use the following relative error check :

$$\frac{|\ observed - expected\ |}{max(|\ observed\ |, |\ expected\ |)} \leq \epsilon$$

In the edge-case where $observed = expected = 0$. we return a positive check result (and thus avoid the division by zero).

There is still a difficulty. By definition, a naive implementation of a convolution is extremely slow. We are about one or two orders of magnitude slower than an optimized version. As a result, comparing systematically against a basic, naive implementation is prohibitive. The solution we choose was to test only

a sample of the result. For each run, we select randomly a number N (typically 1000) of coordinates in the output tensor and compute these points naively. That way, the error checking cost is constant instead of increasing linearly with the problem size.

4.6 Performance reproducibility and stability

Making sure a performance measure is reproducible is a challenge. There is several ways the environment (in a broad sense) can disturb an execution.

- other processes can interfere with a run. We prevent that by running our experiments on a dedicated server thanks to the Grid5000 network [BCAC+13]. This method also has the benefit to run directly on hardware and not via a virtual machine.
- Operating systems can decide to interrupt and migrate a process to another core at any moment. Pinning the process to a particular core solves this problem (this can not be done with a virtual machine though).
- To minimize energy consumption, the CPU can change its frequency behind our back. This can be disabled via the OS
- Disable turbo mode - put 0 into `/sys/devices/system/cpu/intel_pstate/no_turbo`.
- Caches have an impact too, depending on the initial state of your data - do they reside in cache, and if so, at which level.
- Lastly, one possible source of noise is the memory allocation and the address alignment.

The cache problem was a bit more troublesome. There would be two ways of making our benchmark (hopefully) reproducible: either make sure that all our data are already present in the cache before we start the computation (in which case we say the cache is hot), or that all our data are out of the cache and will be brought in at first access (the cache is cold). It is unclear which one is "fairer" or more realistic, and we observed that at least in some cases benches behave differently - we have identified some cases where a bench is better than another with a cold cache, and the opposite with a hot cache. Besides, clearing the cache is not entirely obvious and always depends on the cache policy used. In our case, we have to bring our data into the cache when we initialize them, the point if we want a cold cache is to get them out after. We use a special function (see Figure 4.4) that allocates and repeatedly accesses a buffer that we make sure is much bigger than the L3 cache. To avoid dead-code elimination by the compiler, a dummy computation and IO is performed (the variable *res* in the code).

For memory allocation, two problems coexist: the fact that alignment issues can sometimes affect access latency and the possibility of a page fault. Memory alignment can be controlled via aligned_alloc whose definition follows.

```c
void flush_cache() {
    float tmp[8] = {0.};
    float res = 0;
    for (int i = 0; i< NUM_ITER; i++){
        float *dirty = (float *)malloc(BIG_SIZE * sizeof(float));
#pragma omp parallel for
        for (int dirt = 0; dirt < BIG_SIZE; dirt++){
            dirty[dirt] = dirt%100;
            tmp[dirt%8] += dirty[dirt];
        }
        for(int ii =0; ii <8;ii++){res+= tmp[ii];}
        free(dirty);

    }
  FILE* fd = fopen("/dev/null", "w");
  fprintf(fd, "%f\n", res);
  fclose(fd);
}
```

Figure 4.4: Flushing cache function

```c
#include <stdlib.h>
void *aligned_alloc(size_t alignment, size_t size);
```

We align our buffers on 1024 bytes. This is quite arbitrary, but it should be high enough to be nice to the hardware, although we can never be sure. Another possibility would be to align them on the size of hardware pages.

The way to avoid page fault is to make use of huge pages, which have to be allocated manually via a call to mmap. These pages are up to 1GB large, which is big enough for our execution by a huge margin. Note that this parameter should come into play only when we do allocations during the bench. In practice this should be the case in only one situation: when we pack our data in one way or another and therefore we have to allocate temporary buffers. As we discuss in other parts of this document we finally let packing out of our search space. This means that this specific part should not interact at all with our actual experiments. Therefore, it is mentioned here but was not used in practice.

Nevertheless, there was still an instability that we spent time trying to track down. The weird part was that this instability was not strictly noise. At some point, we observed a startup phase that we did not manage to explain. The first hundreds of 5000 runs were consistently slower than the next ones.

4.7 Conclusion

In this Chapter, we described the experimental platform we run on and some of the technical difficulties we had to face. Those include scalable error checking, flexible counters monitoring, fair comparison with other tools, and taming some instabilities induced by different factors, such as compiler optimization and hardware considerations. The interface was designed to provide some compile-

time semantic guarantees that were useful for productivity. This allows us to explore and evaluate efficiently many different strategies. In the next Section 5, we will explain how we define the space of possible strategies and the way to explore it.

Chapter 5

Space Exploration

In Chapter 3, we showed a framework that allows us to generate code for an optimized kernel. Then in Chapter 4 we demonstrate how we can use this code generation, make sure it's correct and measure its performance (regarding time or other metrics). In this Chapter we are going to discuss how we look for an efficient implementation in a humongous space. In essence, we identify several key points and split the work into different aspects. In Section 5.1 we first try to define what we call a good search space and to list the properties we expect from such a space. Then, in Section 5.2 we justify our choice of restricting the space to build upon a curated set of microkernels. The next Section 5.3 discusses our strategies and hypothesis about *divisibility*, the reasons why this is important, and how we deal with it. Section 5.4 explains the different strategies we tried to choose the outer level tiling values.

5.1 Quality of a search space

In Section 3 we detailed what kind of code we can generate within our framework. We will show that these atoms are enough to provide a competitive implementation of a convolution/matrix multiplication/tensor contraction. However, we still have to find a suitable implementation among the humongous number of possible codes that our framework allows.

This process of selection is split into two steps :

1. Prune the space by setting some constraints that maximize the proportion of good candidates

2. Select some candidate inside this space by a search strategy.

We will see that a smart restriction on the shape of candidates can make the search trivial in certain cases: a simple random search yields top results so quickly that it outperforms any clever machine-learning tool that needs a training phase.

$$\begin{cases} u_i \times u_j + u_j \leq n_r \\ u_i \times u_j \geq l \end{cases} \tag{5.1}$$

Figure 5.1: Microkernel unroll equation

This leads to a definition of what we call a "good" search space: a set of candidates over which we can draw randomly an implementation and have a good chance of getting a good performance. This is purely a matter of the "density" of good candidates rather than a size criterion: a smaller search space is not necessarily better, even if we know it contains at least one good candidate.

This in turn led us to spend some time characterizing our search space upon a few different selection hypotheses. We perform what we call an "ablation study" on these spaces: we try a hypothesis by comparing two spaces "all things otherwise being equal". In practice, for each configuration we have we select randomly a quite high number of candidates (a thousand, for example) and consider it a representative sample of the whole space. We can then plot the distribution of performance over this space, and also select a few metrics such as percentile score. The kth percentile of a space is the value for which k% of the space is below it, and (100 -k)% over it. We can associate this "score" with the expected performance of a random search, given a number of trials. At the end of the day, there is not a single metric that can by itself be taken to compare two spaces against each other, we need both a distribution and a budget of trials to discriminate between two configurations. A space can fair better than another on average with a 10-trials budget search but fall behind with a 50-trials budget.

5.2 Microkernel selection

Recall that in Section 2.1.1 we define what a microkernel is and the constraints we expect a good microkernel to match. We recall the equations here (see above for details) in the case of matrix multiplication in Figure 5.1. We assume a layout where j is the inner dimension of B and C and is vectorized. u_j is the (vectorized) unroll factor on j of the inner basic block, u_i is the unroll factor on i. n_r is the number of vector registers and l is the latency of a vector float multiply-add instruction.

Those constraints make sure that the microkernel can be implemented without any spilling. In practice, modern architectures are tolerant to a degree of register spilling. In consequence, there can be microkernels that do not fit these requirements and still fare well when evaluated on hardware.

The first equation is supposed to guarantee that all accesses to the output tensor/matrix fit in registers, but violating this constraint is not necessarily prohibitive. This can be explained by hardware features such as load-store queue, or instruction level parallelism that allows hiding cache access latency.

This justifies exploring microkernels that are close to the boundaries of this space.

As we saw in Section 2.2.2, convolution and matrix multiplication are fundamentally the same operations minus some details. Therefore it makes sense to look for a good microkernel for convolution too.

Here we put convolution dimensions on the left and their corresponding matrix multiplication dimensions on the right :

$$\begin{cases} h, w \iff i \\ k \iff j \\ r, s, c \iff k \end{cases} \tag{5.2}$$

Now if we want to adapt the notion of microkernel to convolution, then it means that we want to vectorize on k and unroll our basic block on dimensions h, w and k (again, see in Section 2.1.1 why we vectorize on k and unroll on i and j in the case of matrix multiplication).

However, this would miss an important difference between matrix multiplication and convolution, which is that there are two dimensions (r and s) that offer reuse. For example, the access to I is the same for $h = 0, r = 1$ and $h = 1, r = 0$.

$$O[b, h, w, k] += I[b, h + r, w + s, c] \times K[r, s, c, k]$$

This implies that unrolling on these dimensions can spare some registers and favor reuse inside the microkernel, which is intuitively beneficial. As a result, there are five dimensions over which we can unroll our basic block, namely k, h, w, r and s.

For convolution, we assume that k is the vectorized dimension (and the inner dimension of K and O). If we adapt the equations (5.1) to convolution we get the equations shown in Eq. (5.3).

$$\begin{cases} u_h \times u_w \times u_k + u_r \times u_s \times u_k \leq n_r \\ u_h \times u_w \times u_k \leq l \end{cases} \tag{5.3}$$

The microkernel must have a loop along one of the reduction dimensions surrounding it to make sure accesses to tensor *Output* are resident in registers. This allows reusing partially accumulated reductions in the output array promoted to vector registers. The default dimension chosen to do that is c, but it is also possible to do it with r and s when c is too small.

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

$$[\mathsf{U}_{u_s,s}, \mathsf{U}_{u_r,r}, \mathsf{U}_{u_c,c}, \mathsf{U}_{u_w,w}, \mathsf{U}_{u_h,h}, \mathsf{U}_{u_k,k}, \mathsf{V}_k]$$

This microkernel can be unrolled over any dimensions, provided that they meet the following constraints :

1. $16 \leq u_w \times u_h \times u_k + u_r \times u_s \times u_c \times u_k \leq 36$

2. $14 \leq u_w \times u_h \times u_k \leq 28$ (constraint to prioritize the output tensor)

3. $1 \leq u_w, u_h, u_k, u_c \leq 16$

4. $(u_r, u_s) \in \{(1,1),(3,3),(5,5),(7,7)\}$ or $(u_r, u_s) \in \{1\} \times \{3,5,7\}$

$u_w \times u_h \times u_k$ is the number of vector registers used by *Output* and $u_r \times u_s \times u_c \times u_k$ is the number of vector registers used by *Parameters*. Constraint 1 makes sure that the total number of registers used is reasonably close to the number of available registers - if it is bigger we will need spilling, if it is lower then the architecture is underused. Constraint 2 prioritize accesses to *Output*. We do this choice because accesses to *Output* are both loads and stores, which is usually more expensive than doing only loads. Constraint 3 limits the maximum value of unroll. Constraint 4 exists because dimensions r and s do not usually take arbitrary values in practice, but generally fall into one of $\{1,3,5,7\}$.

The order of dimension in this unrolling scheme was chosen to put the reduction dimensions at the outer level, and once this requirement is met it mostly follows the layout we use for convolution. In theory, scheduling two instructions that depend on each other (which happens when unrolling over a reduction dimension) could trigger a latency penalty. In practice, both compiler and hardware scheduler in out-of-order architectures apply reordering to these instructions, so this is not likely to happen, and we did not observe any performance impact of using a different order in our generation scheme.

This makes a total of 3059 microkernels, and we remark that the microkernel $(k, c, w, h, r, s) = (2, 1, 12, 1, 1, 1)$ is one of them. This microkernel corresponds to the one used by BLIS in the context of matrix multiplication (via the mapping from matrix multiplication to convolution we described in Section 2.2.2).

Microkernel evaluation To evaluate performance, we repeat the resulting unrolled basic block many times along the c dimension ($\mathsf{T}_{512,c}$). The results for a slice of the space on AVX512 are shown in Figure 5.2 (on an Intel Xeon Gold 6130, frequency set to 2.1 GHz, Debian GNU/Linux, kernel version 4.19, and hardware counters monitored with PAPI v5.7.0).

5.3 Divisibility

One of our primary choices was to allow only perfect loop nests except for one dimension, which then would be handled with our lambda construct described in Section 3.4.3. We are going to spend the next section discussing this choice and what we have done to justify it.

We recall what we call our divisibility constraint. Given a dimension d that appears several times in a scheme with atoms of size α_1, α_2... The use of our basic atoms imposes that the product of all α divides the full size of d. It is equal to the product of all α if we do not use R_d. For example, in this scheme :

$$[\mathsf{T}_{\alpha_2,j}, \mathsf{T}_{\gamma_1,k}\mathsf{T}_{\alpha_1,j}, \mathsf{T}_{\beta_1,i}, \mathsf{T}_{\gamma_0,k}, \mathsf{U}_{\beta_0,j}, \mathsf{U}_{\alpha_0,j}, \mathsf{V}_j]$$

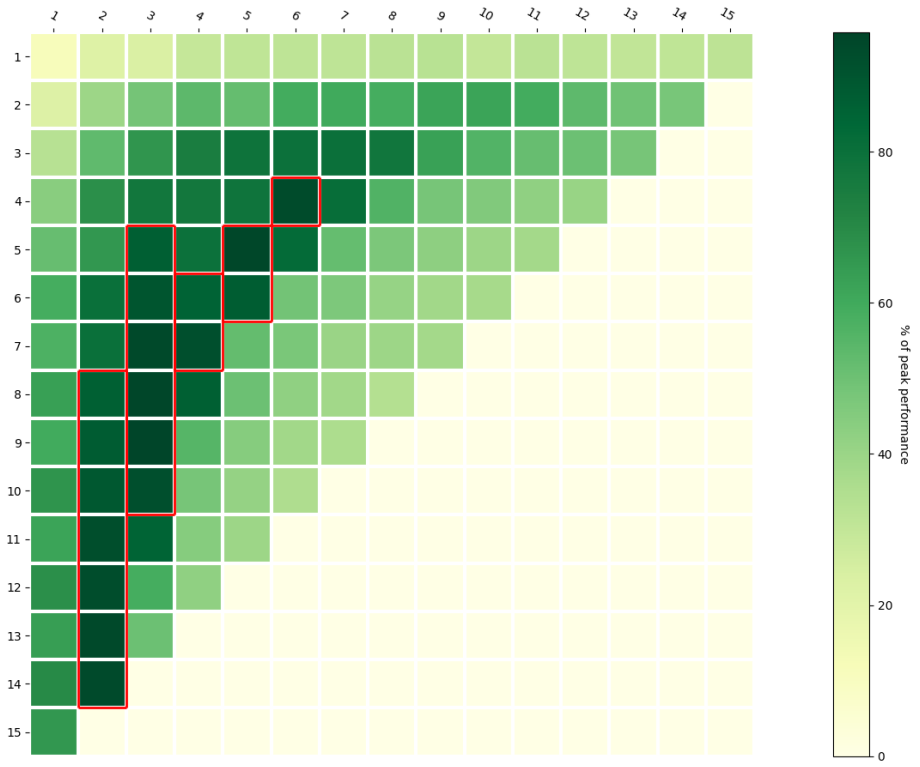


Figure 5.2: Performance of microkernels in isolation for AVX512 in percentage of the machine peak, for the slice of the space where $C = H = R = S = 1$. Microkernel sizes—α along the k dimension (horizontal axis) and β along the w dimension (vertical axis)—vary between 1 and 15. Only the upper-left triangle was evaluated. Red-bordered microkernels are the ones selected (offline).

```
1  for (int i0 = 0; i0 < I; i0+= 42)
2    for (int i = i0; i < MIN(i0 + 42, I); i++)
3      ...
```

Figure 5.3: example of a partial tile over dimension i

, we operate with the following constraints on α_i, β_i and γ_i :

$$\begin{cases} vecsize \times \alpha_0 \times \alpha_1 \times \alpha_2 = J \\ \beta_0 \times \beta_1 = I \\ \gamma_0 \times \gamma_1 = K \end{cases} \tag{5.4}$$

In other words, at every level of the loop nest the footprint of the current tile on a given dimension is a divisor of the full size of the dimension. We call that a *perfect loop nest*.

Other atoms such as $\mathsf{U}\lambda_d$, $\lambda_{\mathsf{seq}_d}.[ell]$, but also $\mathsf{Texct}_{\alpha,d}$ and $\mathsf{Tvar}_{\alpha,d}$ get rid of this restriction.

This problem arises mostly from the fact that we impose the use of a microkernel. As soon as the overall size of the problem is not a multiple of the microkernel size on every dimension, we have to handle the edges in one way or another. There is a subtlety though : Subsection 5.3.2 shows how these edge-cases can be costly in worst cases, even assuming a clever and architecture-dependent "partial" microkernel. Figure 5.3 shows an example of a partial tile. In this one-dimensional code, the full tile that starts at line 2 has a size of 42, and if I is not a multiple of 42 there will be one last tile iteration of size $I\%42$, which we call a partial tile. For completeness, Subsection 5.3.1 explains how to build such a "partial" microkernel, which is a microkernel that supports partial application (that means it can be applied to sizes under a given maximum and not only to a fixed size).

Subsection 5.3.3 gives an empirical basis to the claim that combination of microkernels is more efficient than any other solutions for small matrice multiplications.

There are a few different claims or competing intuitions that should be discussed around this question :

- Non-Divisibility is hard to reconcile with the requirements of a microkernel
- Control flow at a low level is harmful to performance
- Imposing divisibility can harm performance by reducing the tile size options

Before diving further into each of these statements, let us note first that we are facing a methodological difficulty. As we can see, we can imagine different more or less intricate ways divisibility could interact with performance. Some of these claims deal with core-level details, such as the difficulty of implementing

an efficient variable-sized microkernel, or the fact that control-flow instruction can empty the pipeline. The third one plays with cache hierarchy. Last but not least, the last one is a meta-argument about the search-space size. As we can see, by treating divisibility as a whole we smash together these elements that are not playing at the same level.

On the following array we visualize how different tools (including ours) can be positioned in this space :

	Partial microkernel	Lambda	Partial outer tile
Blis	X		X
TTiLe		X	
TVM/Ansor			X

By finding a rigorous comparative analysis between these points in our candidate space, we hope to isolate the specific contribution of these partial tiles on performance. Our initial intuition on this point was that using only perfect loop nests with the addition of the lambda constructor was enough. This would mean that adding the possibility to make partial tiles would only expand the space without improving our candidates.

To explain this intuition, we have to decouple the problem. At the lowest level, we ensure maximum performance by restricting the choice of microkernels to the ones that yield a high-enough performance in isolation. A way to implement partial tiles at the basic block level is to do superfluous computations for the last tile, and then to throw them away, for example with the help of architecture-specific mask operations, as we will see in 5.4 . Thus, given n is the problem size and u_k the microkernel size for a partial microkernel over dimension k, if we define $p = \frac{n \mod u_k}{u_k}$ then the last tile cannot be faster than $p *$ peak performance. This computation corresponds to a ratio of useful computation / effective computation. See Section 5.3.1 for more details.

At a higher level, the question is whether or not this choice can affect the interaction of our candidates with the cache hierarchy.

Essentially the problem boils down to finding a loop nest such that we have a big enough resident tile at every cache level to exploit reuse. Reducing the search space the way we did could hinder this requirement by reducing drastically the proportion of suitable solutions. Here our intuition is that the combinatorial space of tiling possibility makes it very likely that at least some point in the space will meet the criteria.

In a nutshell, by decoupling the space along these axes we can identify which hypotheses are correct or not. If we get a performance boost by using partial tile (which corresponds to switching from the middle column to the left column in 5.3), then it means that the use of microkernel combination (lambda) is not sufficient, and our first intuition is wrong. Similarly, if we get a performance boost by switching from the top row to the bottom row in Figure 5.3, it invalidates our second hypothesis.

5.3.1 How to implement a partial tile microkernel

In this section, we want to show an example of an implementation of a *partial microkernel*. This implementation relies on architecture-specific instructions, namely *masked vector arithmetic*. The point we want to highlight is the cost of a call to this microkernel is independent of the size of the tensor it operates on.

We define a partial tile as a portion of code that is flexible on at least one dimension. That is, a partial tile is a subroutine that does a convolution whose size for any dimensions is either fixed or included in an interval between 1 and some maximum max_{size}. For example, this is a valid microkernel size :

$$microk\_size = \{k{=}32;\ c{=}128;\ w{=}1;\ h{<=}12;\ r{=}1;\ s{=}1\}$$

k is again the dimension we choose to vectorize. It is possible to have multiple partial dimensions by combining multiple masks, however, we will not demonstrate it here.

Implementing an efficient partial tile microkernel is difficult and requires relying on architecture-specific features. One such feature is *masked vector operations*.

An example of a mask operation is a mask store :

```
1 // mask version
2 void _mm512_mask_store_ps (void* mem_addr, __mmask16 k, __m512 a)
3 // standard version
4 void _mm512_store_ps (void* mem_addr, __m512 a)
```

Given a vector v, an adress a and a mask m , a mask store stores $v[i]$ at $a+i$ if and only if $m[i]$ is equal to one. This operation can be used to implement a conditional store (storing a whole vector or nothing). It in turn allows us to implement our partial tile microkernel over either the vectorized or (one of) the unrolled dimension(s). For example here is the code that allows us to implement this partial microkernel using mask stores :

$$microk\_size = \{k{=}32;\ c{=}128;\ w{<=}12;\ h{=}1;\ r{=}1;\ s{=}1\}$$

An example of an implementation of a partial microkernel can be found in Figure 5.4.

There are variants of this technique, such as using mask addition instead of mask loads and stores. The rationale is that control flow inside of the inner basic block should be avoided at all costs.

The practical consequence of this type of microkernel is the following: in the case it is called with its partial dimension used at its maximum it is at best equivalent to a full-tile microkernel in terms of performance (assuming there is no observable penalty for substituting masked memory operations to standard ones). In the case where we use it with size $partial_w <= max_w$ then the amount of computation done is the same that for a complete tile, but a part of it is thrown away. The share of useful work is $p = \frac{partial_w}{max_w}$. Therefore the performance of this partial application of a microkernel is capped by $p * perf_{fullmicrokernel}$.

```
// implementing a microkernel [V k; U ( 1 <= W <= 6), T (256, C)]
int microk(int size_W, // varying y size
float * input,
float * output,
int strH_o, int strW_o
){
  // mask_ones is a mask where all elements are equal to one,
  // same for mask_zeroes
  __mmask16 mask0 = (size_W >= 1 ) ?  mask_ones : mask_zeros;
  __mmask16 mask1 = (size_W >= 2 ) ?  mask_ones : mask_zeros;
  __mmask16 mask2 = (size_W >= 3 ) ?  mask_ones : mask_zeros;
  __mmask16 mask3 = (size_W >= 4 ) ?  mask_ones : mask_zeros;
  __mmask16 mask4 = (size_W >= 5 ) ?  mask_ones : mask_zeros;
  __mmask16 mask5 = (size_W >= 6 ) ?  mask_ones : mask_zeros;
  mem_vec_0 = _mm512_mask_load_ps(&output[h][w][k], mask0);
  // This load is done if and only if w is greater than 1
  mem_vec_1 = _mm512_mask_load_ps(&output[h][w + 1][k], mask1);
  // This load is done if and only if w is greater than 2
  mem_vec_2 = _mm512_mask_load_ps(&output[h][w + 2][k], mask2);
  // etc.
  mem_vec_3 = _mm512_mask_load_ps(&output[h][w + 3][k], mask3);
  mem_vec_4 = _mm512_mask_load_ps(&output[h][w + 4][k], mask4);
  mem_vec_5 = _mm512_mask_load_ps(&output[h][w + 5][k], mask5);

  for (int c = 0; c < 256; c++) {
    // This basic block is exactly similar to those of a standard microkernel
    // of size K = 16, W = 6 (unrolled 6 times on w)
    input00 = input[h + r][ w + s][c];
    vec_input00 = _mm512_set1_ps(input00);
    vec_params0 = _mm512_load_ps(&params[h][w][c][k]);
    mem_vec_0 = _mm512_fmadd_ps(vec_input00, vec_params0, mem_vec_0);

    // unroll 1 on w
    input01 = input[h + r][(w + 1) + s][c];
    vec_input01 = _mm512_set1_ps(input01);
    mem_vec_1 = _mm512_fmadd_ps(vec_input01,
vec_params0, mem_vec_1);

    // unroll 2 on w
    input02 = input[h + r][(w + 2) + s][c];
    vec_input02 = _mm512_set1_ps(input02);
    mem_vec_2 = _mm512_fmadd_ps(vec_input02,
vec_params0, mem_vec_2);

    // omitted  - standard basic block vectorized on K and unrolled on W
  }
  // Same trick as for loads
  _mm512_mask_store_ps(&output[h][w][k], mask0, mem_vec_0);
  _mm512_mask_store_ps(&output[h][w + 1][k],mask1, mem_vec_2);
  _mm512_mask_store_ps(&output[h][w + 2][k], mask2, mem_vec_4);
  _mm512_mask_store_ps(&output[[h][w + 3][k], mask3, mem_vec_6);
  _mm512_mask_store_ps(&output[h][w + 4][k], mask4, mem_vec_8);
  _mm512_mask_store_ps(&output[h][w + 5][k], mask5, mem_vec_10);
}
```

Figure 5.4: Partial microkernel implementation on AVX512

```
1  for (c1 = 0; c1 < 1000; c1 += 49)
2    for (c0 = c1; c0 < MIN(c1 + 49, 1000); c0 += 12)
3      for (c = c0; c < MIN(c0 + 12, c1 + 49, 1000) ; c += 1)
4        <basic block>
```

Figure 5.5: Example of partial tile semantic

5.3.2 Discussion on partial tiles

In this section, we will demonstrate how the use of a partial tile at the inner level (that is: not making sure that every tile size is at least a multiple of the microkernel size on every dimension) can harm performance even when the size of the problem is much bigger than the size of the microkernel.

Here we assume the use of what we call a "partial" or "flexible" microkernel. A fixed microkernel operates on a fixed-size subtensor. On the contrary, a "partial" microkernel can be applied with a parametrized size over one or more dimensions. We will see how to implement such a microkernel in Section 5.3.1, and why such a microkernel has a constant cost, whatever the size it operates on.

For the sake of the example, we will reduce the problem to one dimension d. Let us assume the full size of d is 1000. Let us also assume for external reasons such as available microkernels and output of a model, we want to implement it with a two-level tiling scheme with successive tiles of size 12 and 49, surrounded by a loop of footprint 1000 (full size).

This amounts semantically to the code presented in Figure 5.5. Note that 12 does not divide 49 nor 20 (which is equal to 1000%49, nor that 49 divides 1000, which implies in both case the use of partial tiles, one of size $49\%12 = 1$ at each iteration of the inner tile, and one of size $1000\%49 = 20$ at the end. We assume here that 12 is the optimal size for our partial microkernel.

As 49 does not divide 1000, our computation consists of 20 tiles of size 49 followed by one last tile of size 20. Similarly, a tile of size 49 is implemented as 4 calls of size 12 to the microkernel followed by a call of size 1, and a tile of size 20 is computed as 1 call of size 12 followed by a call of size 8. Therefore the full operation is implemented as

$$20 \times (4 \times microk(12) + microk(1)) + 1 \times microk(12) + 1 \times microk(8)$$

which can be rewritten as :

$$(20 * 4 + 1) \times microk(12) + 20 \times microk(1) + microk(8)$$

However, every call to microk has the same performance cost, no matter how much computation it does semantically. This code does actually $20 \times 4 + 1 + 20 + 1 = 102$ calls to $microk$. This in turn means that our implementation does the same amount of computation as we would have needed for a dimension of size: $102 \times 12 = 1224$. As such, this partial tile implementation yields a wasted computation rate of 22.4%. What we want to highlight here is that in

the context of multi-level tiling, these boundary effects can have a significant impact even on big sizes, while one could have intuitively thought that they would become insignificant in this context. This is explained by the fact that these boundary effects happen at each iteration of the tile that is immediately outer of the microkernel on the dimension that induces a performance loss.

For a microkernel that is optimal for a size of $size_{uk}$, the worst-case scenario is when the size of the problem is equal to $size_{uk} \times \alpha + 1$ for a given α. This is the point where the last tile is the most underused. The impact decreases with α.

This effect will be even more obvious in the next section where we test different competitors on small matrix multiplications.

5.3.3 A case study on small matrix-multiplication

In this section, we illustrate 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 do 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 to continue using the microkernel one last time, at the cost of additional computation. In this work, 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 5.2.

Figure 5.6 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 percentages of the absolute peak performance, corresponding to the maximal utilization of the two vectorized FMA units of the architecture.

MKL [WZS+14], Blis [VZvdG15] and libxsmm [HHHP16] 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 of the size of their microkernel along the i dimension. Libxsmm also considers the combination of microkernels but restricted itself to predefined sizes such as powers of 2 along the i dimension. Our experiment shows that this is not enough to obtain consistent performance for all problem sizes.

"*Single microkernel, partial tile*" is the performance of code generated by our framework, but only 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

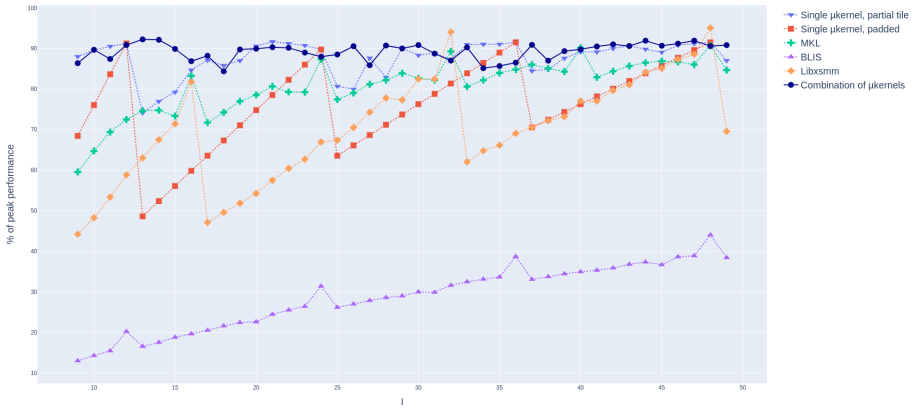


Figure 5.6: Performance of small matrix multiplication kernels, for $J = K = 128$ and $8 \leq I \leq 50$.

assumed that the padding overhead is free. As expected, the performance for low modulo is 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 combining 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 optimization space around this constraint.

5.4 Tiling above the microkernel

Once we have made our choices at the lowest level of the code, by picking microkernels, we still have to choose how we are going to iterate over this basic block. From a performance point of view, this matters mostly because of the interaction with the cache and the reuse pattern of our code. In the next section, we will consider two things: what we call the *permutation*, which is the order in which we iterate over dimensions, and the tile sizes, which are the sizes of the loops.

For example in the following example :

$$[\mathsf{T}_{4,w}, \mathsf{T}_{3,r}, \mathsf{T}_{4,h}, \mathsf{T}_{2,c}, \mathsf{T}_{4,w}, \mathsf{T}_{16,k}, \mathsf{T}_{3,s}, \mathsf{T}_{48,h}, \mathsf{T}_{256,c}, \mathsf{U}_{12,w}, \mathsf{U}_{2,k}, \mathsf{V}_k]$$

The permutation is $[w, r, h, c, w, k, s, h, c, w, k, k]$. This distinction matters,

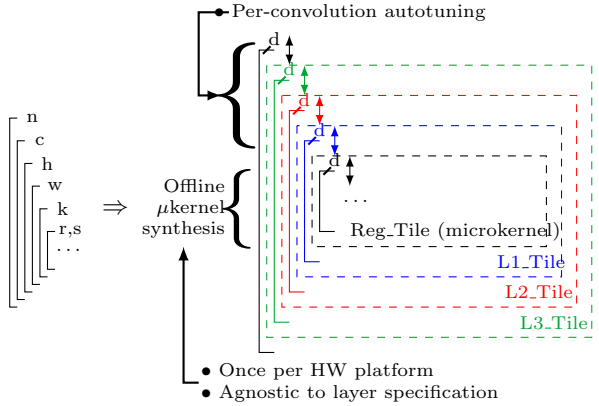


Figure 5.7: Multi-level tiling and split two-level optimization.

because most of the strategies we will see set these choices one after the other - first they fix the permutation, then the sizes at every level. In Section 5.4.4 we will present a way to couple these choices together.

Hierarchical tiling is a well-known technique in compilers that allows maximizing cache usage. It consists in making sure that at every level of cache usage a significant amount of the data is reused quickly enough that it does not get evicted. While the principle is simple, the application is difficult in practice. Even within a very simple framework such as Belady's cache model (perfect clairvoyant replacement policy) [JL16], finding the right tile size and shape, in general, is a difficult problem that yields several interesting works such as Ioopt [OIT+21]. This work aims to minimize cache misses in some sense.

More recently, Ioopt tries to maximize a metric called Operational Intensity. This consists of the ratio of the volume of data movements out and into a given cache level divided by the amount of computation at this level. Ioopt uses the polyhedral framework to generate a polynomial symbolic equation that is then fed into a solver. This in turn yields a schedule.

In practice, cache policy does not correspond to a simple least recently used algorithm. Thus there is always the risk to run into unexpected behavior, especially when the patterns of accesses are complicated (which can happen when they are the result of an optimization problem). There are several attempts to retro-engineer this policy : However, most of this behavior is still hard to fit into a model.

Also, these model-centric approaches struggle with the fact that final performance is multi-factorial and that weighing every factor against each other is challenging. We have tried a few different ways of dealing with this problem that we are going to present here.

5.4.1 On the question of permutation

It is worth spending some time explaining why the choice of the permutation matters - and to which degree. Choosing a dimension to iter on at some level has two consequences :

1. Choosing which tensor footprint is growing at this level
2. Among the tensors that are accessed, with which stride we access the tensor.

The second one matters a lot in terms of vectorization. As we explained in Section 2.1.1, we choose to restrict vectorization only to dimensions that have a stride 1 over all tensors it accesses. Therefore the innermost atom of our scheme (which is the only valid position for vectorization) is necessarily one that has this property - unless we are ready to pay the price of a reshuffling of our data, which is also a possible strategy as we discuss in Section 5.5. The first parameter matters at higher levels because it determines *reuse*: the part of data that is reaccessed across iteration at this level and therefore is more likely to stay in the cache. *Streaming over a tensor* is the process of iterating on a dimension that does not appear in the access function of this tensor, which means exploiting reuse over it.

Recall now the BLIS strategy we described in Section 2.1.1. The point of their tiling tactic was, at each level of cache, to make one matrix grow sufficiently that it gets close to the size of the cache, then to *stream over this tensor*, that is, to iterate over a *reuse dimension* of that tensor. From this follows a quite natural way of choosing permutation: for each level of cache, choose a tensor, first make it grow by iterating over one or several of its dimensions, then choose a reuse dimension that allows streaming. Some competitors - such as AutoTVM - choose to have a handcrafted permutation and focus their search on the tile sizes. In my work, we have tried a few different ways of finding this permutation.

5.4.2 Ioopt

Ioopt is an algorithm developed by Olivry et al. [OIT+21] that aims to modelize and minimize the data movement volume of a program. In this context, IO means the number of loads toward a given level of cache a program will make. An obvious lower bound of this is the footprint of the program. Every data used in the computation has to be brought into the cache at least once. The question is whether or not we can make sure it can be brought only once - that is if we can schedule the program in such a way that it is never evicted before its last use. Given a specification of a program satisfying some constraints (such as being a rectangular space - that is a program expressible as a set of perfectly nested loops) and a cache size, Ioopt can generate a symbolic expression that gives a higher bound on the program IO. Then it feeds this expression into a numeric solver that tries to minimize it, thus yielding a possible solution for the schedule.

We tried to leverage this model for our optimization. Some problems arise though. The first one is that the solver yields a fractional solution, while loop bounds are integers. We could solve this problem by approximating each loop bound to the nearest integer. However, this would still clash with our requirements on divisibility. Nothing constraining the solver to choose tile sizes that are consecutive multiples.

Therefore our first attempt was to minimize the distance to the best theoretical solution. We tried to find the candidate that minimizes the euclidian distance with the Ioopt solution. That alone has not proven to be very efficient. Essentially, the candidate that minimizes this distance does not necessarily have some other properties that are essential to performance.

In particular, there is a necessary condition that we need for the microkernel to yield performance. For reasons we explain in Section 2.1.1, a microkernel should be iterated many times by the loop just above it (at least a few tens times) to amortize the cost of memory accesses. This requirement can clash with the goal of minimizing data movement.

Otherwise, we could not observe any correlation between this distance to Ioopt solution and performance. There were several possibilities that are hard to discriminate :

- Ioopt model may not accurately model data movements and cache misses
- The approximation we had to make to satisfy our divisibility requirement may be too loose to retain the model
- The bottleneck of our executions could be something else than the cache effects

Otherwise, the integration between Ioopt and Ttile was not great - in fact, nonexistent - and needed improvement.

5.4.3 Model-based filtering

The struggles encountered in the last section motivated a search for a better-integrated flow between Ioopt and our work. One key observation was while Ioopt is a fine work, its use is probably disproportionated with regard to our needs. Indeed, Ioopt can analyze programs that are much more complex that the limited sets of applications we strive to optimize. This is great in theory but comes with a cost: the need to call a complex solver. This non-linear solver was a dominant part of the runtime of our research. This is a shame because the Ioopt model can be vastly simplified when it is specialized to our use case. This section describes a work that strives to eliminate the need for a solver. It does so by re-implementing a subset of the Ioopt method which aims to evaluate the volume of data movement of a given implementation. Instead of doing a symbolic resolution, this is done numerically for each candidate in the search space, which in turn allows us to select the candidates that yield the best scores. This implies that the space must be small enough to be entirely enumerated in

a reasonable amount of time, which relies heavily on the fact that we impose the use of a selected microkernel and that a given permutation is chosen by an external tool.

The principle is the following : as you recall, Ioopt works by finding a lower bound on the volume of data retrieved toward a given level of cache. We only give here an intuition of the way Ioopt as a whole is working, for more details see the work of Olivry et al. [OIT+21].

Let us take a candidate schedule, that is, a set of pairs of dimensions and tile sizes that represents the levels of a loop nest. We consider the shortest suffix of this schedule (the innermost part of the loop nest) whose footprint overflows the cache. We can make an approximation of a lower bound if we assume that the footprint of this schedule suffix is brought into the cache only once. Essentially, the takeout is that the model of Ioopt predicts that in the case of convolution and tensor contraction this lower bound is tight. This result depends on the assumption that the cache replacement policy behaves nicely (Ioopt assumes that one has complete control over which data is evicted from the cache at every moment). Hence, in this model, each iteration of the suffix will bring exactly its footprint into the cache - and not more, which would be the case if some of the data used were evicted and then reused later. The volume of data brought into the cache is therefore the product of the footprint of this suffix and the number of iteration above it. We recall the definition of a convolution given in Section 2.2.2, with the assumption that $b = 1$ as we explained in Section 2.2.2:

$$O[h, w, k] += I[h + r, w + s, c] \times K[r, s, c, k]$$

From this definition, we can deduce the footprints on every tensor given the sizes of a specific schedule suffix. k, c, h, r, w, s are the sizes for each dimension at a given level of a loopnest.

$$\text{Footprint of Input} = c \times (w + r - 1) \times (h + s - 1)$$

$$\text{Footprint of Output} = k \times h \times w$$

$$\text{Footprint of Parameters} = k \times c \times r \times s$$

$$\text{Global footprint} = \text{Input Footprint} + \text{Output Footprint} + \text{Parameters Footprint}$$

$$\text{Volume of computation} = k \times w \times h \times r \times s \times c$$

Let us look at an example. Here is a scheme for a hypothetical convolution of size $\{k : 256, c : 512, w : 192, h : 192, r : 3, s : 3\}$: We assume a vector size of 16 (which is the size of a single-precision float vector on AVX512).

$$[\mathsf{T}_{4,w}, \mathsf{T}_{3,r}, \mathsf{T}_{4,h}, \mathsf{T}_{2,c}, \mathsf{T}_{4,w}, \mathsf{T}_{16,k}, \mathsf{T}_{3,s}, \mathsf{T}_{48,h}, \mathsf{T}_{256,c}, \mathsf{U}_{12,w}, \mathsf{U}_{2,k}, \mathsf{V}_k]$$

We are going to add to every atom the corresponding footprint for all tensors at this level of the loop nest. The way the footprint is updated (from bottom to top) stems quite naturally from the definition of the footprint we gave earlier.

Each atom on a dimension makes the footprint of the tensors it accesses grow accordingly. For example, a $\mathsf{T}_{4,c}$ will multiply the footprint over *Input* and *Parameters* by 4 but let *Output* unchanged - because c accesses both the first two tensors but not the third. Sizes are given in number of single-precision floating points elements.

Atom	Input	Output	Parameters
$\mathsf{T}_{4,w}$	19M	9.4M	1.2M
$\mathsf{T}_{3,r}$	4.9M	2.3M	1.2M
$\mathsf{T}_{4,h}$	4.7M	2.3M	393K
$\mathsf{T}_{2,c}$	1.2M	589K	393K
$\mathbf{T}_{4,w}$	**614K**	**590K**	**197K**
$\mathsf{T}_{8,k}$	153K	147K	197K
$\mathsf{T}_{3,s}$	153K	18K	24K
$\mathsf{T}_{48,h}$	147K	18K	8K
$\mathsf{T}_{256,c}$	3K	384	8K
$\mathsf{U}_{12,w}$	12	384	32
$\mathsf{U}_{2,k}$	1	32	32
V_{k}	1	16	16

Now assume we are computing what is the data movement towards a cache of size 4Mb, that is 1Mb Single Precision floating points elements. This is a realistic size for an L2 cache in a modern server CPU. We need to find the level at which we overflow this cache - that is when the total footprint of the convolution is bigger than 1Mib. Here this happens at the level of $\mathsf{T}_{4,w}$ - the line that appears in bold. So following the model we are going to assume that each iteration of this suffix :

$$[\mathsf{T}_{4,w}, \mathsf{T}_{16,k}, \mathsf{T}_{3,s}, \mathsf{T}_{48,h}, \mathsf{T}_{256,c}, \mathsf{U}_{12,w}, \mathsf{U}_{2,k}, \mathsf{V}_{k}]$$

brings its data only once in the cache. Therefore the volume of data brought into the cache by this scheme is the footprint of this suffix :

$$\begin{array}{lll} & 614400 & \text{(footprint Input)} \\ + & 589824 & \text{(footprint Output)} \\ + & 196608 & \text{(footprint Parameters)} \\ = & 1400832 & \end{array}$$

multiplied by the number of times this suffix is iterated on, which is equal to the product of the number of iterations for each loop above it :

$$\text{Number of iterations} = 2 \times 4 \times 3 \times 4 = 96$$

Which yields a data movement volume of an estimated 1400832 floating point elements that will be brought to the cache.

This yields a metric for every level of cache. However, it is not obvious to decide how to combine the scores obtained on different levels of the cache. Each of the cache levels has the potential to be the bottleneck of the applications.

In this context, we would ideally want to compare two candidates at the level where performance is most critical. However, we don't have any way to know in advance which resource will be the bottleneck and it can be different across candidates. Still, by definition, there is always more data movement toward the L1 cache than toward the L2 cache (each access in L2 corresponds to at least one access in L1, but conversely there can be multiple accesses to L1 that do not correspond to an L2 access). This point is mitigated by the fact that L1 has more available throughput than L2, which means it can handle more accesses. Our choice was to take the sum of all cache contributions weighted by the available bandwidth at this level. This is motivated by the execution cache memory (ECM) model, which according to [STHW15] makes better predictions than the roofline model for tiled code.

$$\texttt{Ioopt}_{metric} = \sum_{i=1}^{3} \frac{\texttt{movement(}L_i\texttt{, cand)}}{bw_{l_i}}$$

bw_{l1}, bw_{l2} and bw_{l3} are homogeneous to a quantity of data divided by a time, and can be expressed in number of floats per cycle (or number of cache line per cycle). They allow us to weigh the contributions of the different levels of cache. As *movement* is homogeneous to a quantity of data, $\texttt{Ioopt}_{metric}$ is homogeneous to a time, which can be thought of as an estimation of the time needed to make all necessary moves from a level of cache to another. The main advantage of this metric is that it is extremely fast, sufficiently that it can be worth evaluating it on a large number of candidates.

Now that we have a metric supposed to measure the behavior of a candidate with respect to the cache, we need to use it to select our final candidates. If the space of possible implementations is not too big, the simplest way is to generate the whole space, run the model on each candidate, and select the best ones. Therefore the viability of this approach depends on how much we can constrain the number of candidates. The way we do this is by :

1. At the inner level, forcing the choice of a good microkernel (as we explained earlier)

2. At the outer level, choosing a permutation to avoid combinatorial explosion.

Figure 5.8 summarizes this flow. The empirical box followed by the microkernel selection at the top-left represents the process presented in Section 5.2 which consists in running a large set of microkernels to keep only those over a given threshold of performance. On the top right, the permutation selection is offloaded to Ioopt - a choice we will discuss afterward. The divisibility constraint then allows us keeping only the microkernels and combinations of microkernels that fit the problem sizes, and the set of possible tiling scheme given the chosen permutation. This is what we call "compatible optimization schemes". These candidates are then sorted according to the metric we just described ("sorted candidate schemes") to select the best ones. If we are running in parallel, we

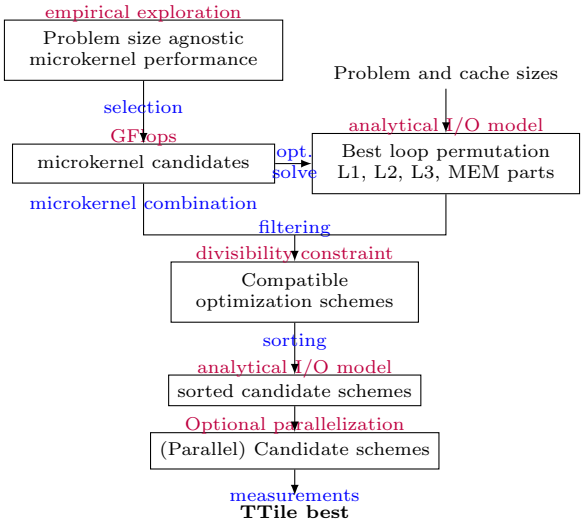


Figure 5.8: Flow of the optimization algorithm.

apply a pass of parallel adaptation of the candidates. Last but not least, the selected candidates are run and the best one is the final implementation.

Here, we chose to exploit the permutation given by Ioopt. This means that we still rely on this tool, which is unfortunate because we still pay the prohibitive cost of running the non-linear solver. Thus we undermine the benefit of our fast model evaluation. To reduce our experiment time, we used a memoization technic to cache the results of this model on every triplet architecture/microkernel/problem size. This improved significantly the running time of our experiments but did not solve the main problem. Also, this setting does not allow us to test the impact of the choice of permutation, as only one was tested for every problem size.

5.4.4 Tree search

In the last section, the search space was artificially limited by choosing a specific permutation. This is very unfortunate because we do not need a complete specification to run at least some of the evaluation. For example, when evaluating the data movement volume at the L1 cache level, we only care about the suffix of the scheme that resides in L1 and the loop above it. That means that there is a huge number of redundant evaluations. In this section, we describe an early pruning technique that allows us to evaluate innermost parts of a scheme only.

The rationales behind this search are the following, and are similar to the ones used in the BLIS methodology :

- Choose a microkernel at the inner level
- Saturate the cache with one tensor at every level

The idea is to describe the search space as a decision tree. Each node represents a subpart of a complete scheme. We start at the innermost level and build the scheme in a bottom-up fashion. Each edge in the tree corresponds to a couple *dimension/tile size*. At each level, we take the remaining size of every dimension and find every divisor of it, which represent the possible choices for the next atom. Then, the set of all tuples of *dimension/tile size* constitutes the set of choices at this node. This search tree is lazily evaluated: we only expand the nodes we visit.

As we discussed in the last section, this setting was motivated by different goals. The main one was to get rid of the dependency on Ioopt, especially since the heuristic it uses to choose permutation does not even need the call to the solver. By doing that we make permutation a part of our search space again, which allows more opportunity but also dramatically increases the number of potential candidates. Hence we need to perform some kind of search heuristic and/or prune at least some parts of the choices available at a given level in the tree.

As the space is way too big to be entirely explored, we need a way to cut it down. We chose to follow and generalize the guiding principles of BLIS tiling scheme. That is, putting it simply: saturating the cache by growing a tile on a specific tensor, then reusing this tile at the outer loop level. However, we let the algorithm decide which tensor to use at each level, instead of hardcoding the decision, like in BLIS.

The heuristic for expanding the tree is the following: at each level, we first generate all possible choices, apply a pruning algorithm that depends on the previous decisions taken (but is strictly local to this node and does not depend on other branch derivations), and iterate until either of the three following conditions is met :

1. we have completed our scheme (all dimensions have been fully iterated)
2. we reach a checkpoint condition (see below)
3. or we reach a dead-end - none of the possible choices pass the pruning phase (which we will detail next).

What we call a checkpoint is a point where a metric can be applied and thus, it makes sense to compare all branches that reach this point globally. In practice, this means every node whose footprints overflow the cache. Recall that all we need to compute the metric previously defined is the footprint of the tile and the number of iterations around it, which means we do not need to know the precise scheduling of the outer levels to compute the data movement in this setting.

As for the pruning constraints, they stem from our generation pattern. Firstly, we make sure that a dimension is used at most once at a given cache level. Second, we alternate two phases : *growing* and *reusing*. During the *growing* phase, the point is to make one tensor grows enough to be close to filling the cache. Therefore each choice is constrained by the choices taken earlier: all

dimensions chosen during the *growing* phase should be part of the access function of one given tensor. For example, let us assume the two first dimensions we choose are c and k. The only tensor whose access function contains these two dimensions is $Parameters$. Therefore until the *growing* phase is over the only possible dimensions are the one that access $Parameters$, namely r and s (c and k cannot be taken again). Then as soon as the footprint passes a given threshold, we switch phase to *reusing* mode. In this phase, we only take dimensions that allow reuse on the tensor that we were making grow. This means we want to choose dimensions that do not appear in the access function of the tensor we selected during the growing phase. If this tensor was $Parameters$, then the only possible choices are h and w.

Once the footprint reaches the size of the cache level we are working on, we interrupt the expansion phase. Then we gather all final (yet unexpanded) nodes, compute their associated data movement, and select the ones which perform best. The number of branches we select is a matter of tuning.

A few choices need to be discussed with the algorithm. There is a weakness when we compare it to the last version in terms of combining metrics at the different cache levels. The previous version used a flat space and as such, did not make an a priori preference between these levels. On the contrary, this version is by design hierarchical and imposes choosing candidates at the lower level before choosing at the higher ones. However, there is no guarantee that the best candidates at the L1 level are the best at the L2 level (and reciprocally). As a consequence, this algorithm privileges the metrics associated with the lower levels of cache.

Another question is how many branches to keep at each level. This relates to the previous question: if we were to choose one branch only at the L1 level, that would mean prioritizing the L1 entirely. However, selecting more candidates at the L1 level let us the opportunity of then applying a tradeoff to the next levels. We could for example alternate in the following way: best L2 child of the best L1 node, second best L2 child of the best L1 node, best L2 child of second best L1 node... This is again a limitation of our model: we do not have any obvious way of combining our predictions on different levels of cache.

Finally, there is the question of when to switch from the *growing* to the *reusing* phase. One of the potential drawbacks of having only perfectly divisible loop nest sizes is that we cannot target precisely a particular footprint size, as we would do if we were allowing ourselves the use of partial tiles. As such, we have to guarantee both that a big-enough tile can be reused and that this tile does not overflow the cache. So the point is to decide what is considered an acceptable window for a tile size that is supposed to be reused in the cache. We decided that a tile size should be bigger than half the size of the cache to pass in reuse mode - that is, the threshold is set to `cache size` /2 in the presented algorithm.

With this setting, we finally get rid of the Ioopt dependency entirely while still leveraging its model. We originally strived to have a one-shot implementation, that is to say, to be able to find a good-enough implementation in one try. The goal was even to decide on the fly instead of ahead of time so that we

```
1  def expand_rec(node, context, cache_level_size):
2    // computes footprint of current node
3    // (all decisions taken from root)
4    branch_footprint = footprint(node, context)
5    if context.grow && branch_footprint > cache_level_size:
6      // if we get out of the cache in grow phase,
7      // we abort this branch
8      return ABORT
9    if branch_footprint > cache_level_size:
10     // Here we are in reuse phase and we got out
11     // of the cache, so we stop here for now
12     return CHECKPOINT
13   // get a list of dimension and the size
14   //  that is still to be iterated left
15   dim_sizes = remaining_dim_sizes(node)
16   candidates = []
17   // building candidates
18   for (dim, size) in dim_sizes :
19     // all_divisors(i) returns all divisors of i
20     // different from 1
21     divisors = all_divisors(size)
22     for div in divisors:
23       append(candidates, (dim, div))
24   // dims chosen in this phase
25   previous_dims = context.dims
26   // list of tensors accessed by all these dimensions
27   accessed_tensors =
28     intersection(map(accessed_tensor, previous_dims))
29   // if we are in context GROW then we allow dims
30   // that access at least one tensor accessed
31   // by all previous dimensions, else we are looking for reuse
32   allowed_dims = if context.grow then dims(accessed_tensors)
33         else all_dims \ dims(accessed_tensors)
34   // remove dims that were already taken
35   allowed_dims = allowed_dims \ context.dims
36   // keep only candidates with legal dims
37   candidates =
38     filter(fun (dim, _) -> dim in allowed_dims, candidates)
39   for (dim, size) in candidates:
40     new_node = update(node, dim, size)
41     new_context = context
42     if footprint(new_node) > threshold:
43         new_context.grow = false
44       expand_rec(new_node, new_context)
```

Figure 5.9: Tree expansion algorithm - intermediate recursive function

```
function expand_until_level(root, cache_level_size):
  expand_rec(root, INITIAL_CONTEXT, cache_level_size)

function search():
  root = init_tree()
  for level in (l1, l2, l3):
    expand_until_level(root, level)
    best_branches = select_n_best(tree, num_branches)
    tree = rebuild_tree(best_branches)
  return candidates_from_tree(root)
```

Figure 5.10: Global search heuristic

could compete directly with OneDNN instead of TVM.

As such, this framework offers several advantages. First, it allows as we said to evaluate a whole class of implementations under the model at once. These classes of schemes are based on the equivalence relation "having a common suffix ". According to the model, two implementations belonging to the same class will behave the same at a given level of cache if their common prefix has the right reuse property (a tile fitting in the cache surrounded by a reuse loop). This property makes it easier to prune quickly: it is enough to generate all prefixes that reach the first level of the cache limit instead of generating all possible implementations, which would be way too big. This metric is still comparative instead of absolute though and as such, some kind of global comparison across candidates is needed.

It is not clear that even assuming our metric is a good proxy for cache misses we should necessarily find the candidate that minimizes it. Indeed, given that we face a bottleneck problem, the question is not whether or not a candidate generates more data movements than another but whether the cache bandwidth is the bottleneck in this particular configuration. There is no point in improving data reuse if the bottleneck is the instruction decoder, for example. Instead of comparing candidates against each other, we could reason in terms of bandwidth capacity and operational intensity. As such, the criteria could be to prune any candidate that does not reach a given ratio of computations over data movements.

Evaluation of this algorithm This algorithm was implemented and tested to some degree but never fully evaluated. There were a few reasons for that. The first one was that it was still suffering from a few shortcomings. While it was able to generate solutions for most of our benchmarks, in a minority of cases it did not find any solution at all. The reason for this was that some corner cases were not addressed, such as the case where a convolution was small enough to fit into the L2 cache. It would have been possible to make up for such shortcomings but at the price of many hard-coded fixes that would have obfuscated the point of the algorithm. Moreover, this implementation gave us the building tools for conducting a random exploration of the space, which made us reconsider some

of our assumptions. In consequence, we redirected our research in a direction that we are going to describe in the next section. We still think that at least some of the ideas presented here could be of interest for future work.

5.4.5 A baseline better than expected: Random search and metric evaluation

This part describes a late direction in our research. Essentially we found out that with the restrictions discussed earlier, the space of remaining candidates contains so many good ones that there is no need for a smart search strategy. In other words, given that we allow ourselves a bit of training, it is very difficult to beat random selection even when trying only very few candidates.

We describe here what we call "random search", how we explore our search space fairly.

We did multiple iterations in different directions taken in our search for a good heuristic for tiling. At first, we were only comparing these alternative methods with our competitors but it was unsatisfying. Indeed, it was not enough to get an understanding of the specific contribution of the tiling method to performance. To get that, we implemented a *random search* that we are going to describe now. It turned out to yield much better results than expected.

The implementation is quite simple, it stems from the same principles as the last section. We select tiling atoms from inner to outer, generate every possible pair of dimension/size at this level, choose one randomly, update the search state (tile sizes left to iterate) and recurse until all dimensions have been entirely consumed.

Random search makes for a very good baseline for a search strategy comparison. A strategy makes sense only if for a given "budget" of trials, we get consistently a better result among the candidates chosen by our strategy than with the same number of candidates chosen randomly. This can depend on the number of trials we are willing to spend. For example, a strategy could be on average better than random at selecting a single candidate but lose with as little as 5 or 10 candidates. When the space is small enough to be tractable, all strategies will converge with the number of trials - at the extreme, all strategies should find the best candidate if we let them the opportunity to try them all. Meanwhile, searches that rely on machine learning and thus try to leverage the knowledge built on successive trials (which amounts to building an ad-hoc metric on the fly) have a training phase. This means they are not expected to perform better than random at least for the first trials and need enough training to justify the choice of this method (there is no point in investing in machine learning tools if you do not let your model enough time to train).

We performed some simple experiments that hinted that random search was an efficient strategy able to quickly converge toward a very good result in as few as 20 or 30 trials.

This led us to conduct a more thorough evaluation of our space, hoping we could build a better understanding and distribution. Therefore, the next experiment was to draw randomly a thousand candidates for each benchmark

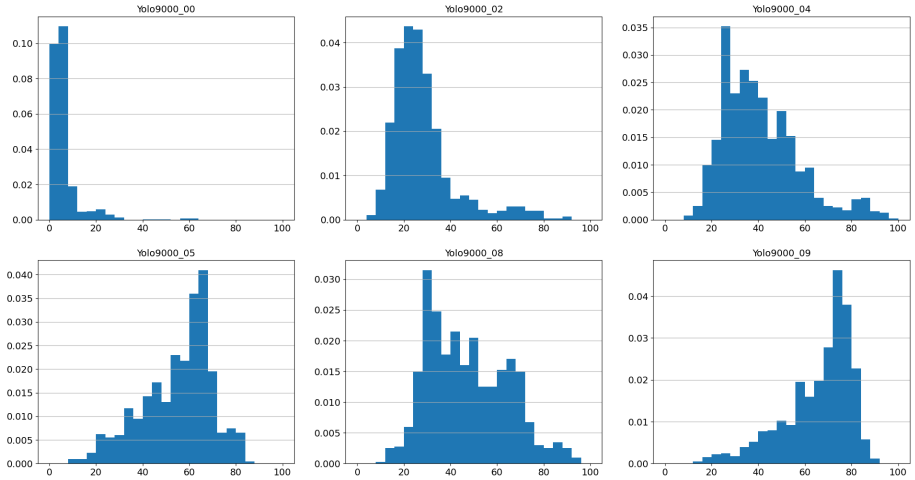


Figure 5.11: Distribution of candidates for Yolo9000 convolutional layers from 00 to 09

we've got and to evaluate them on hardware. From there, we built an histogram to visualize how these candidates are positioned in term of performance as shown in Figures 5.11, 5.12, 5.13 and 5.14.Each of these plots corresponds to a convolution of our benchmark, whose sizes can be found in Figure 6.1.

In these figures, absciss represents the performance in percentages of peak performance and the ordinate represents the ratio of candidates falling in the corresponding range of performance. For example in Figure 5.11, the ratio of candidates in our space that have performance falling in the 20% and 24% of peak performance bin for Yolo9000-2 can be found by multiplying the height and width of the bin. Hence this ratio is of 0.045 (approximate height) times 4 (size of a bin in percent) which is close to 18% of the whole space.

We see that the average performance is quite high. Assuming that this distribution is representative of the real space of candidates, we can use it to compute other information, such as the number of trials needed to reach a certain performance, within a confidence threshold.

5.4.6 Computation of the expect function

In this part, we are going to assume the 1000 candidates we draw randomly for each problem size are representative of the actual distribution of candidates in the space of implementations we built. That is, we assume that the percentage of candidates that have a performance of 70% of the peak performance or higher amongst these 1000 candidates is the same as the percentage of candidates over 70% in the full space. From this distribution, we are going to derive a function we call *expect* that computes, for a given number of trials n, the expected performance of the best candidate from n random trials. This also depends on

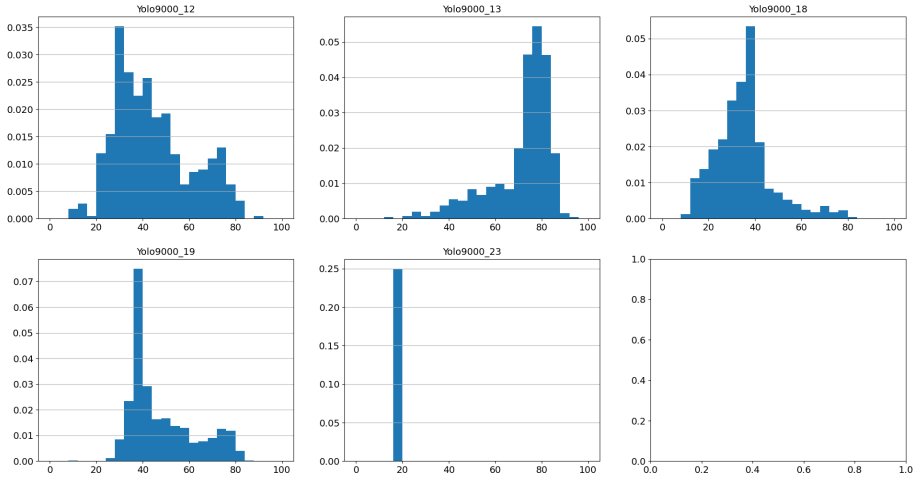


Figure 5.12: Distribution of candidates for Yolo9000 convolutional layers from 12 to 23

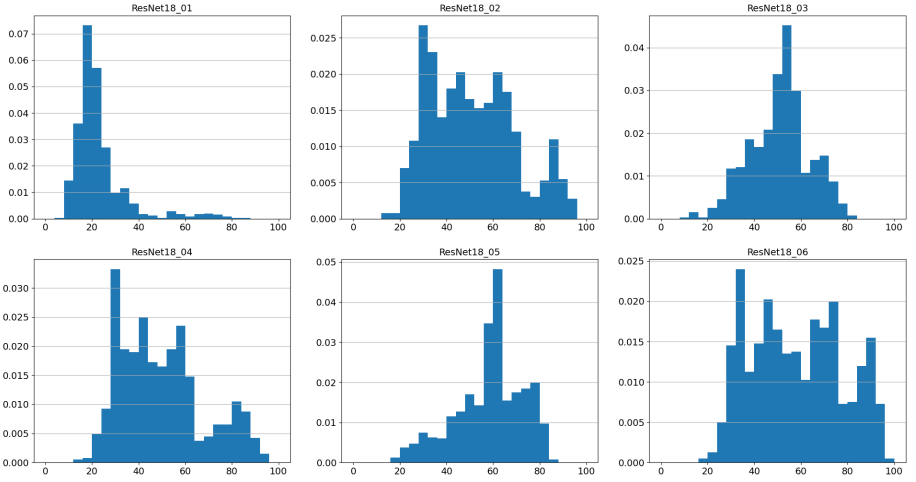


Figure 5.13: Distribution of candidates for ResNet18 layers from 01 to 06

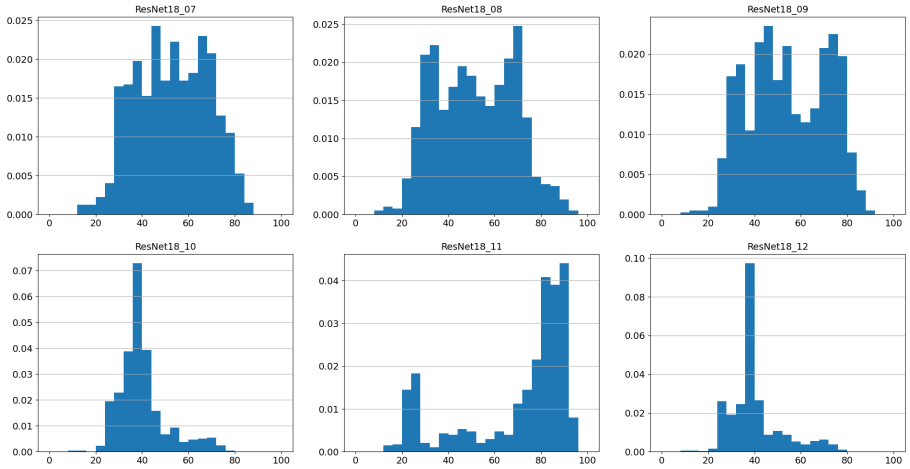


Figure 5.14: Distribution of candidates for ResNet18 layers from 07 to 12

the confidence we want: if we want confidence of 90%, then $p_{cand} = expect(40)$ means: Assuming S is a sample of 40 candidates drawn randomly, and $best_{cand}$ is the best candidates amongst them, then there is a 90% probability that $best_{cand}$ has a performance of p_{cand} or higher.

Now we are going to set up a few notations to help us define this function. We call the set of candidates we have $\mathcal{C}$ and we define a function perf that maps a candidate to its performance as measured when it was evaluated. As it is expressed in percentage of peak performance, the values taken by this function are in the interval $[0, 100]$. We call the image of this function $\mathcal{P}$.

$$\text{perf} : \mathcal{C} \mapsto \mathcal{P} \subset [0, 100]$$

Now we define a function $\pi(p)$ that yields the estimated probability that a randomly taken candidate performs better than p. This corresponds to the cumulative distribution of p :

$$\pi \colon \mathcal{P} \mapsto [0, 1]$$
$$\pi(p) = \frac{|\{c \in \mathcal{C} | \text{perf}(c) >= p\}|}{|\mathcal{C}|}$$

We want to build a function that, from a given number n of trials and a confidence threshold τ, returns the performance p such that the best candidate among the n we draw has a performance higher than p with probability τ.

$$expect \colon \mathbb{N} \times [0, 1] \mapsto \mathcal{P}$$
$$expect(n, \tau) = p \text{ such that } \mathbb{P}(\max_{c \in S_n}(\text{perf}(c)) >= p) >= \tau$$

where S_n is a set of n candidates chosen randomly.

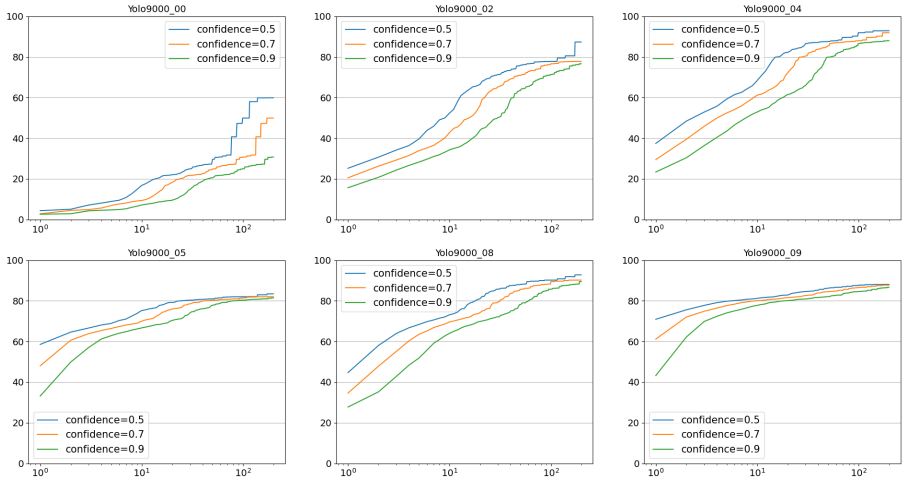


Figure 5.15: expect(n, τ) for Yolo9000 convolutions from 00 to 09 on Xeon-Gold6130 (sequential) with $\tau = 0.5, 0.7, 0.9$ in **log scale**

We have this equation :

$$\mathbb{P}(\max_{c \in S_n}\{\text{perf}(c)\} \geq p) = 1 - (1 - \pi(p))^n$$

And from this we can solve the definition of $expect(n, \tau)$ for p :

$$\begin{aligned} & 1 - (1 - \pi(p))^n = \tau \\ \implies & (1 - \pi(p))^n = 1 - \tau \\ \implies & 1 - \pi(p) = \sqrt[n]{1 - \tau} \\ \implies & \pi(p) = 1 - \sqrt[n]{1 - \tau} \\ \implies & p = \pi^{-1}(1 - \sqrt[n]{1 - \tau}) \end{aligned}$$

Inverting π amounts to find a percentile of our distribution and it can be computed numerically. Note that π^{-1} converges toward a maximum at some point (which is the maximum value found in our distribution).

In Figures 5.15 and 5.17, the function *expect* is plotted for $\tau = 0.5, 0.7$ and 0.9 for each convolution in our benchmark. The x-axis is in log scale. These charts read as follows: the x-axis shows the number of trials, and the y-axis shows a performance p (in percentages of peak performance) such that the probability of getting at least one candidate better than p from the n drawn is above τ. For example, for Yolo9000_8, taking the best out of 25 random candidates has more than a 90% probability of getting a performance of at least 82% of peak performance.

We can draw a few conclusions from that. First, the distribution of candidates is much better than we initially thought. There are a lot of very good

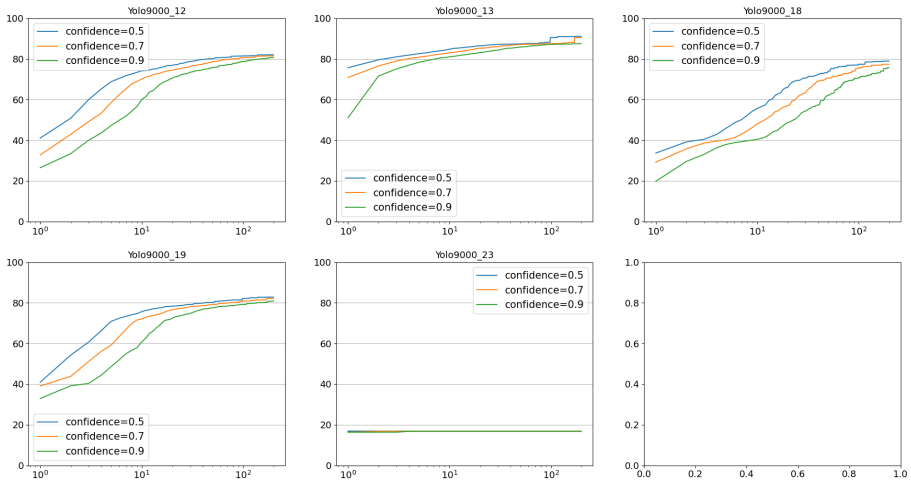


Figure 5.16: expect(n, τ) for Yolo9000 convolutions from 12 to 23 on XeonGold6130 (sequential) with $\tau = 0.5, 0.7, 0.9$ in **log scale**

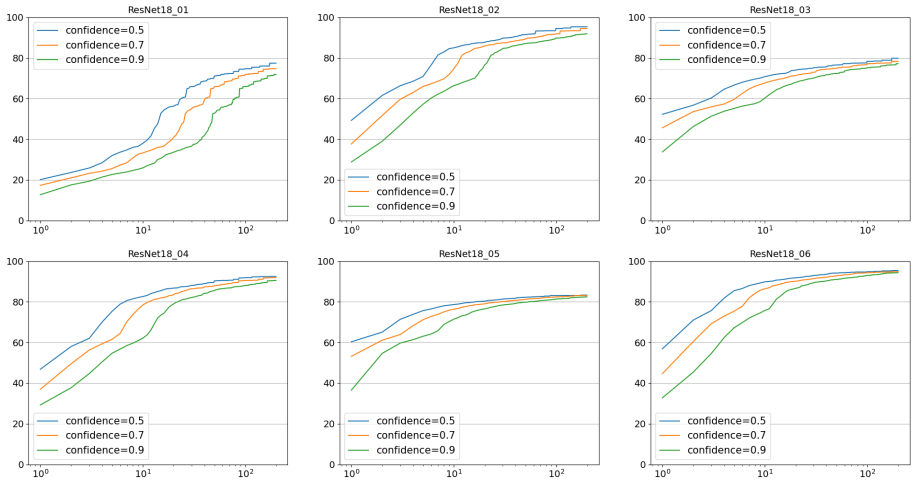


Figure 5.17: expect(n, τ) for ResNet18 convolutions 01 to 06 on XeonGold6130 (sequential) with $\tau = 0.5, 0.7, 0.9$ in **log scale**

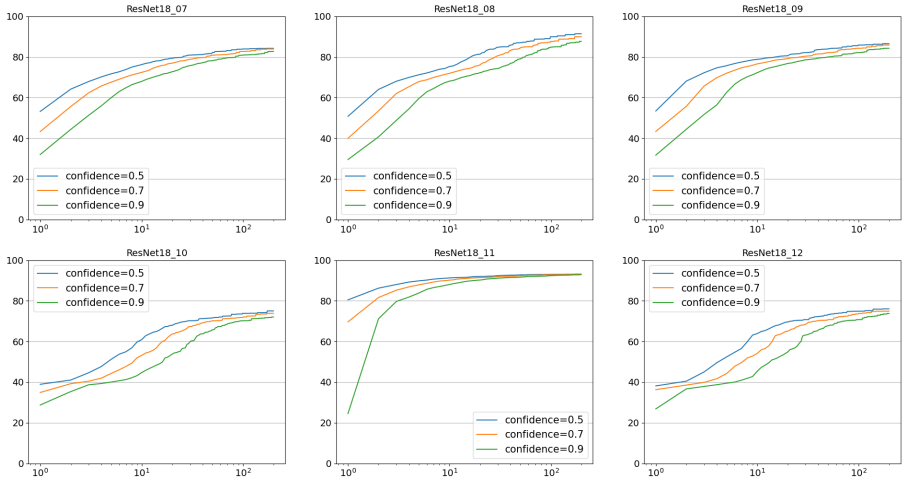


Figure 5.18: expect(n, τ) for ResNet18 convolutions 07 to 12 on XeonGold6130 (sequential) with $\tau = 0.5, 0.7, 0.9$ in **log scale**

candidates and the median is quite high for many of our benchmarks. As a result, a purely random search is likely to find a candidate close to the best one (or at least the best candidate we found each time among our random selection of 1000 implementations) in less than a few tens of trials, as shown in Figures 5.15 and 5.17.There are some convolutions for which the distribution is less impressive: the median is lower and we have fewer very good candidates. This shows on their associated *expect* function - see for example Yolo9000_0 or Yolo9000_18. In these cases, random strategy convergence seems to happen later than for others. We do not have clear explanations for these cases. Nevertheless, most of the benchmarks we have seen show a quick convergence of performance.

This setting put our search strategies into question. For example, any learning process is likely to be overcome, as this type of search can very quickly reach a candidate close to the best one in the whole space.

This raises another discussion: it is not clear what criteria should be used to declare that two candidates are "close" in general, that is, that we expect them to have similar behavior and performance. For example, two candidates could consist of the same atoms but for two levels but still have very different behavior. As a consequence, there is a risk of observing huge threshold effects which could hamper search strategies based on the continuity hypothesis. Associated with the fact that the constraints we set on our space allow us to have many good candidates in proportion, this makes a compelling argument for using a random search not only as a baseline but as a real strategy as soon as you allow yourself as few as 50 trials.

5.5 About layout and packing

A starting point of my work was to evaluate the need for shuffling the data for tensor-contraction. There was an intuition that the constraint of having a fixed microkernel and the consequence it implies (reshuffling data to make them suit the microkernel requirement) could be detrimental to performance. The reason for that was that the benefit of having all accesses contiguous in memory could be not as worthy as expected - or that the benefits would be strongly outweighed by the cost of the repacking. This intuition was reinforced by the fact that the sizes and layouts used by the common microkernels (BLIS ones, for example) are difficult targets for highly-optimized transpositions (such as the routine presented in [SSB17].

In practice, we observe that packing on AVX512 has not proven to be very useful. Indeed, one of the main benefits of having exclusively contiguous accesses in memory is to exploit spatial locality. For performance reasons, CPU caches have a granularity of cache lines, which are usually 64 bytes wide. This means that any cache miss will evict and bring 64 bytes exactly into the cache. Bandwidth usage is maximized only if the application makes use of these 64 bytes entirely before the line gets evicted. This is why packing is so important. However, on AVX512, a vector register has the same size as a cache line, which means that maximized cache line usage is guaranteed for any vector load or store. Moreover, other tools we compare against in Chapter 6 use packing and we still manage to match their performance, which also hints towards the fact that it is probably not critical. However, this observation stems from observations done mostly in the sequential case. There is a chance that the conclusion would be very different if we had focused more on the parallel case, although we did some preliminary work on that and did not observe any improvement either.

Therefore packing has not been considered as a part of our search space. It could be of interest on other architectures but we will let that it to future work.

5.6 Parallelism

Until most of what we discussed was related to single-thread consideration: instruction-level parallelism, register allocation, spilling, and cache behavior. However, on a modern CPU multicore usage is a crucial feature of the performance. The two architectures we used for our experiments offer respectively 18 and 32 cores that we need to exploit. The question is which level(s) of our tiling scheme should be made parallel.

We recall (again) the definition of a convolution :

$$O[h, w, k] += I[h + r, w + s, c] \times K[r, s, c, k]$$

We can see in this definition that a convolution has three parallel dimensions: h, w, and k (they are the dimensions that appear in the output tensor).

Parallelizing a reduction loop would imply a degree of synchronization, otherwise, there would be concurrent accesses to the same memory location and potentially incorrect results.

Given a scheme, we want to select which tiling atoms are going to be parallelized. We followed some basic principles here. First, as cores have a private L1 and L2 cache but a shared L3, it makes sense that each parallel chunk has a footprint that is at least as big as the L2. Indeed, as L2 is private it allows each chunk to benefit from its L2 capacity at full. Secondly, we have to make sure that the loop we parallelize has enough iterations to make use of all cores. If needed, it is possible to fuse consecutive parallel tile atoms to create a bigger parallel tile.

Moreover, we assume that the order of the higher-level tile atoms does not matter if they are outside of the last level of the cache that is overflown. Considering the tile consisting of the smallest suffix that overflows the last level of cache, this assumption is true as soon as there is no reuse across two iterations of this tile. This is true if no reduction dimension appears in the outer levels of the loop nest, or if all data from one iteration are evicted before they are used in another iteration. This hypothesis is the same as the one we use in Section 5.4.3. This allows us to reorder the last tiles if it allows us to fuse some of them to get a bigger tile.

Our strategy is, given a scheme, to select all outermost tiling atoms which make the footprint overflow the L2 cache. These tiles (that can be freely reordered without any effect on the cache under our assumption) are then reordered in such a way that all parallel dimensions were put at the outermost position and then fused and parallelized. Then the tile scheme is chosen at random similarly to what we do in the sequential case, except that we prune any candidate that does not have enough iterations at the outer level (outside the L2 cache) to have at least one parallel iteration per thread. This means we do not execute candidates that iterate all parallel dimensions at the inner levels of their scheme.

Chapter 6

Experimental results

There are two significant kinds of results we want to highlight in this Chapter. Firstly, we can match the performance of industrial tools such as TVM or OneDNN, be it in parallel or in sequential. Secondly, we present a dissection of the contributions of the different factors to the performance.

6.1 Performance evaluation - sequential

The comparison is done over 23 convolutions that are taken from two CNN used in real world settings that are common in convolution benchmarks: Yolo9000 and ResNet18. The sizes of these convolutions are detailed in Figure 6.1.

We compared against a few competitors that we presented in Section 2.1. Here is a quick recap :

1. OneDNN is a JIT-based (one-shot) library developed by Intel [Int18]
2. Autoscheduler/Ansor is the last version of the TVM autotuning framework [ZJS+20]
3. Mopt is a tool developed by Li et al. [LSV+19] based on a data-movement model close to Ioopt

The best one proved to be TVM, especially when it was extended with a new heuristic presented in Ansor [ZJS+20]. OneDNN was quite good but suffered from the fact that it is intended to be a one-shot tool. We also compare against Mopt, which we presented in Section 2.1.8.

Ansor is an autotuning tool, which means that it needs several runs before it can find a good implementation.

As of the end of this thesis, the best results were obtained with the following configuration: filtering the microkernels and taking only those that attain 85% of the peak performance or more, finding a permutation with the help of IOOPT, sorting all candidates with a custom metric, selecting the best n candidates according to this metric and finally execute these n candidates to find

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 6.1: 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.

the best amongst them. We choose 100 as value for n. This choice allows the total evaluation time of all selected candidates for our 23 convolutions examples (combined) to stay below 30 minutes. This is described in Section 5.4.3. At the moment of writing, there is ongoing work that we are going to describe in the next section that tries to better understand which part of this pipeline is beneficial to performance.

As we explained before, we place ourselves in a setting where an application is written and optimized once but then run thousands and thousands of times. Therefore it is worth allowing a budget of trials to allow autotuning in the case of Ansor or to compensate for a deficiency of our metric in our case.

This raises the question of which training budget we allow. In the case of Ansor, we let it have a budget of 1000 runs. As we can see in Figure 6.2 this is often too much - Ansor search converges to a maximum before we reach this limit of 1000 candidates.

On our side, we show two settings: one where we take the best of 100 random draws, and the other where we do the same with 1000 draws. Results are presented in Figures 6.3 and 6.4. We used three different architectures: two Intel AVX512 (Xeon Gold6130 and Xeon Gold5220) and one ARM architecture (ARM ThunderX2_99xx). ARM is a recent addition. As we worked mostly with Intel architectures until this point, we expected our process to be less efficient when applied to a new setting but it proved to yield decent results. This demonstrates that past a quite small number of draws, progress is only marginal and comes at the expanse of a much longer trial phase.

There are several conclusions to draw from these results. First, we can see that apart from a few convolutions our results with 100 runs are mostly on par with the results obtained after a thousand runs, which was hinted at by

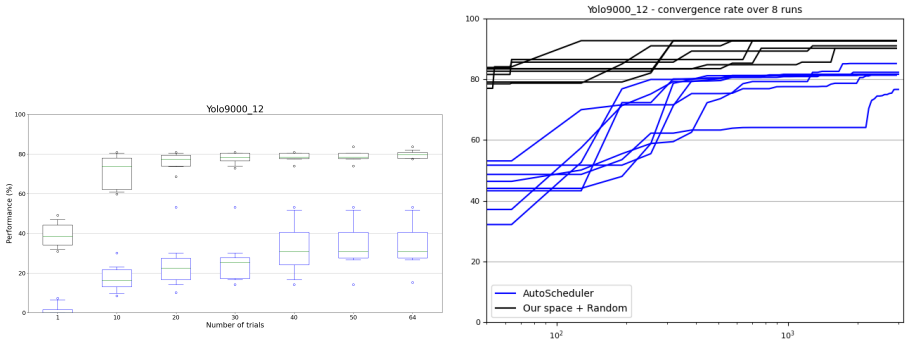


Figure 6.2: Comparison of the convergence rate of 8 random samplings in our space, against 8 independent executions of AutoScheduler (in blue), for sequential code generation, on Yolo9000_12, targeting an Intel Xeon Gold 6130. (i) The left figure shows the maximum of the performance of the first 64 chosen implementations. The boxplot for AutoScheduler summarizes the 8 executions, while the boxplot for random sampling represents 8 executions. (ii) The right figure shows the best candidate found by AutoScheduler after each batch of 64 runs, for a total number of runs of +3000, and compares it to the best candidate found by random sampling for an equivalent number of runs.

the experiments done in Section 5.4.5. Second, we outperform all competitors except for AutoTVM which beats us 14 times out of 23 on Gold5220, 6 times out of 23 on Gold6130, and 13 times out of 23 on ThunderX2_99xx. We beat them on average for every network on every architecture. The reason for AutoTVM being behind OneDNN on average is that Yolo9000-23 does a huge amount of computations and therefore outweighs all the others.

6.2 Performance evaluation - parallel

The work we described in the last section has not been done extensively in the parallel case yet. This does not mean we did not investigate into the parallel usecase though. We have some preliminary results on only one network (ResNet18), on a AVX512 XeonGold6230 architecture with 26 threads (which is a different architecture from the ones used in sequential, because this one was more easily accessible), and we compare only with AutoTVM/Ansor. The selection of our schemes is the one described in Section 5.6. In a nutshell, we select one hundred candidates by first collecting the best microkernels and then select the outer levels randomly (without any call to Ioopt, be it for permutation or for tile sizes) exactly as in the sequential case, except that we make sure that there is enough parallelism to feed all threads outside of the L2 cache. It led to decent results, as can be seen in Figure 6.5. We are outperformed by Ansor on all convolutions but are most of the time on par with them, with the exception of ResNet-01. Here we are clearly outperformed by Ansor in nearly all cases,

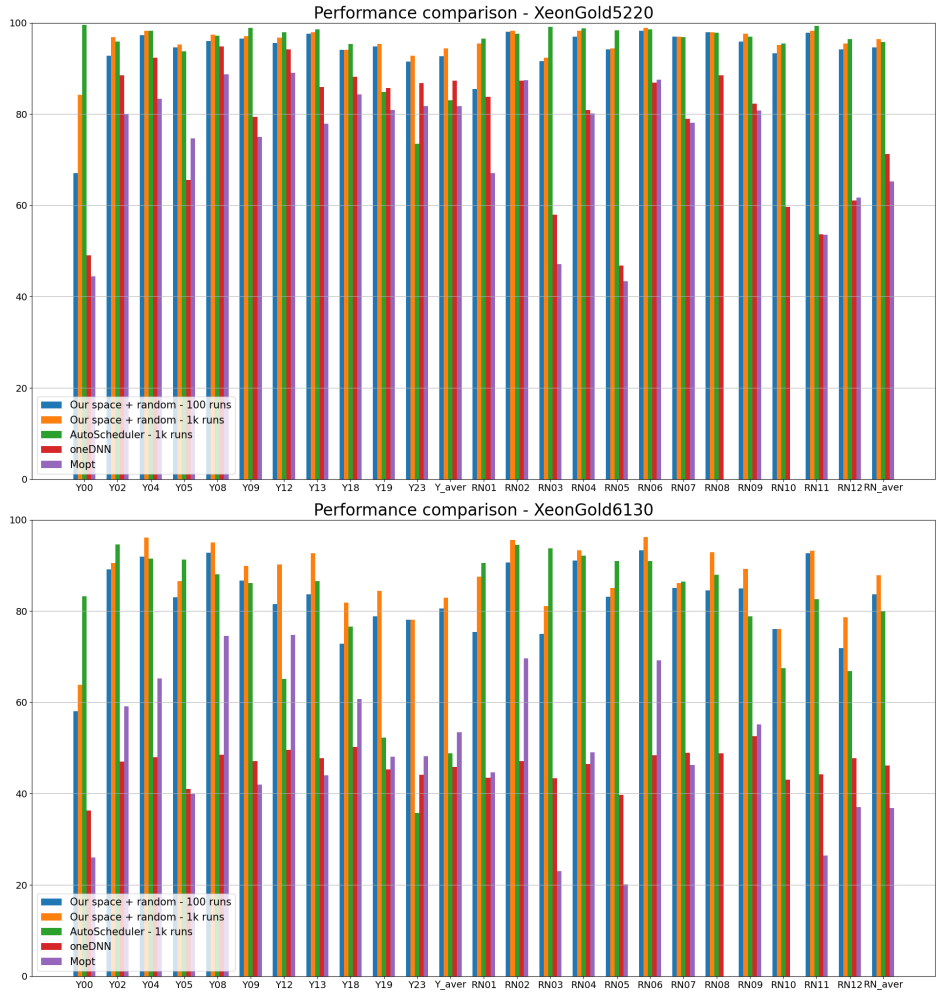


Figure 6.3: Sequential performance comparison with AutoScheduler, oneDNN, Mopt for AVX512 (Intel Xeon Gold 5220 and 6130) shown as percentage of machine peak. The averages given for each CNN are weighted by the amount of computation in every layer.

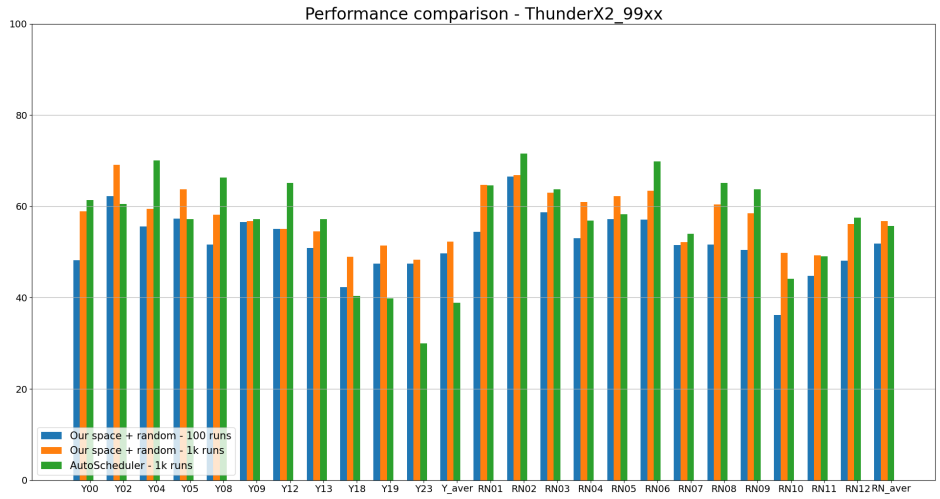


Figure 6.4: Sequential performance comparison with AutoScheduler for Neon (ARM ThunderX2_99xx), shown as percentage of machine peak. The averages given for each CNN are weighted by the amount of computation in every layer.

sometimes by a huge margin. We need to refine these results before we can say more about them.

6.2.1 Random search in parallel

We also have preliminary results with a methodology similar to the one described in Section 5.4.5 only on a specific benchmark and on a different architecture from usual. It turned out that in addition to the requirement of having a microkernel at the inner level as in the sequential case, parallel execution needed an extra rule to make random search competitive. The point is to make sure that enough iterations can be parallelized at the outermost level. To do that, assuming that the architecture we are working on has $n_{threads}$ cores, we select a parallel dimension for which the total number of iterations is a multiple of $n_{threads}$. Then we make sure a tile of size $n_{threads}$ on the selected dimension is placed at the outermost level. The remaining of the scheme is chosen randomly - apart from the microkernel.

We gathered the results on all benches of Yolo9000 in Figures 6.6 and 6.7. From this histogram we can get the same kind of chart that we built in Section 5.4.5, shown in Figures 6.8 and 6.9. We recall these charts represent the expected performance for a given number of draws. The convergence is slower than before, so contrary to the corresponding charts in Section 5.4.5 the x-axis is in linear scale.

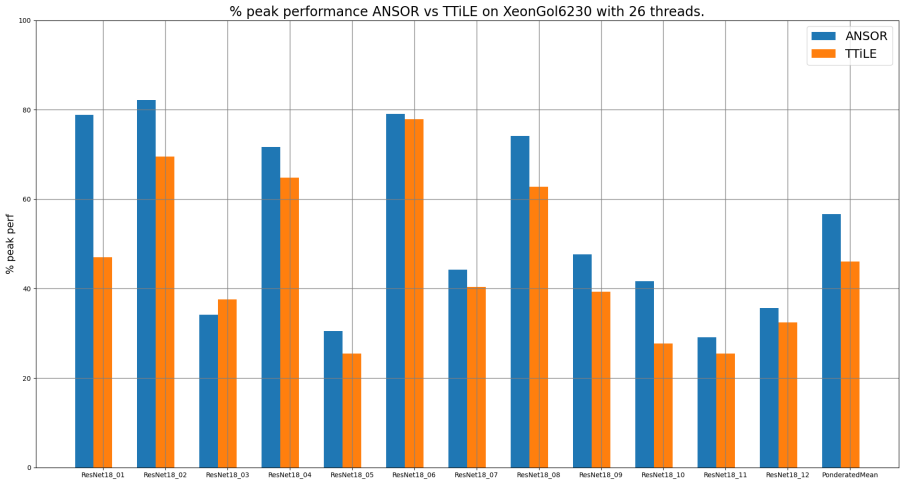


Figure 6.5: Parallel execution of ResNet18 on XeonGold6230 (26 threads) with Ansor and TTiLe

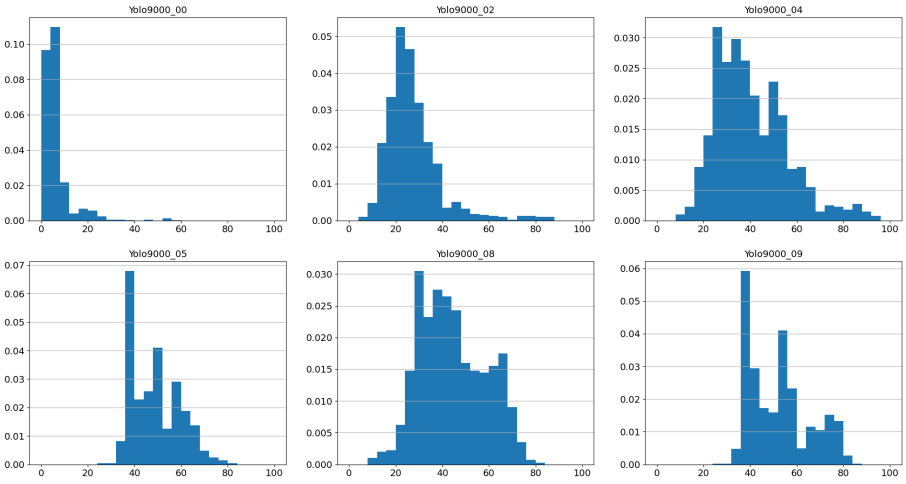


Figure 6.6: Histogram of performance for Yolo9000 00 to 09 on XeonGold6230 (parallel)

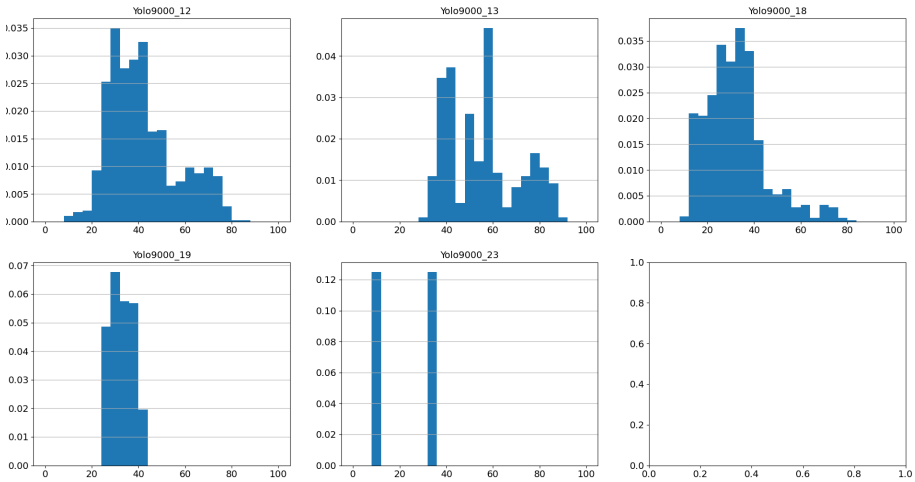


Figure 6.7: Histogram of performance for Yolo9000 12 to 23 on XeonGold6230 (parallel)

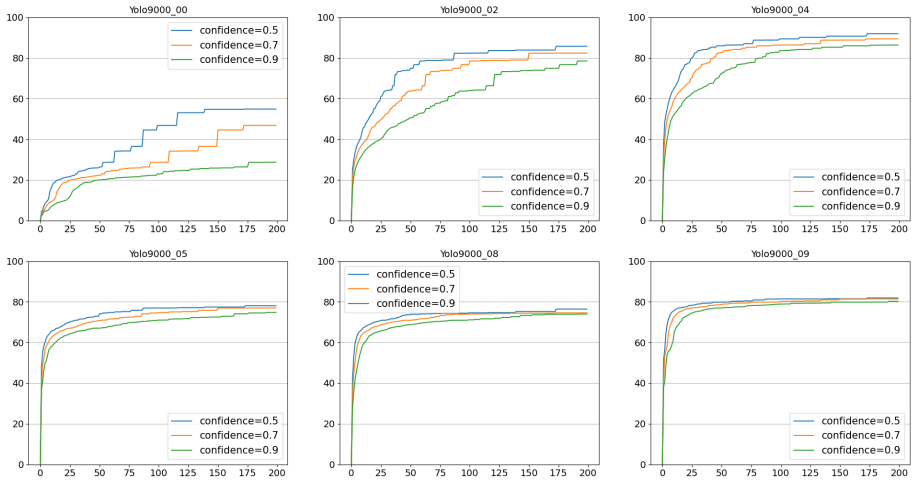


Figure 6.8: expect(n, τ) for Yolo9000 convolutions 00 to 09 on XeonGold6230 (parallel) with $\tau = 0.5, 0.7, 0.9$ **in linear scale**

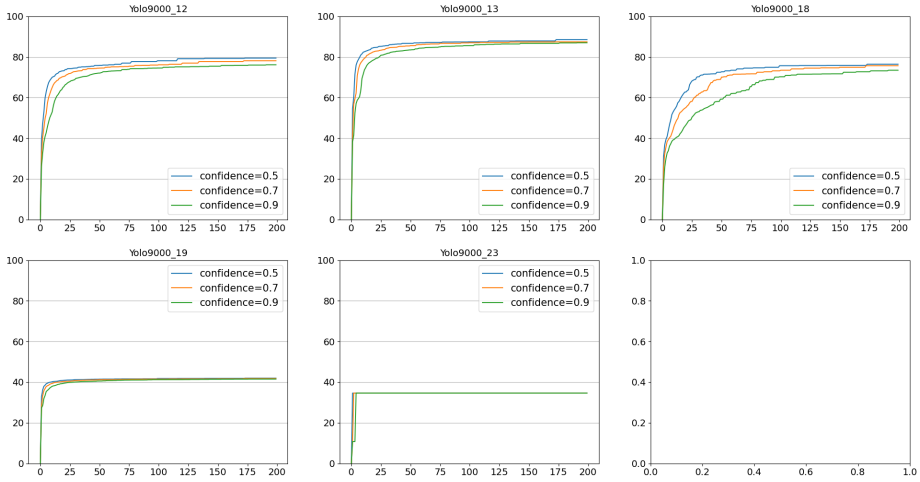


Figure 6.9: expect(n, τ) for Yolo9000 convolutions 12 to 23 on XeonGold6230 (parallel) with $\tau = 0.5, 0.7, 0.9$ **in linear scale**

6.3 Dissecting performance contributions : Ablation studies

In Chapter 5 we discussed many ways we could implement a tensor-contraction that has impacts on both practicability and performance. Some of these possibilities are mutually exclusive, while others are not. Here is a list :

- Using a single microkernel (which can be partial as described in Section 5.3.1)
- Allowing multiple microkernels
- Using multiple microkernels, but restrict by testing them in isolation and keeping only the best ones
- Restricting the sizes of the outer loops to divisors of the full size
- Allowing partial tiles on outer loop levels
- Using our lambda constructor
- Fix a permutation beforehand, using a heuristic or a model
- Make the permutation part of the search space

These criteria define a space of possible implementation and do not suffice to find the best implementation by themselves. But as we discussed in Section 5.4.4 these spaces can be characterized by their distribution: we can draw a sufficient number of candidates randomly into the space and look at how they

fare regarding performance. Looking only at the best candidate found is not enough: if it was the case, we could just take a superset of all spaces. There are two different motivations for restricting our space search: either to ease implementation (in terms of implementation, it is easier to rule out edge cases than to deal with them) or to improve the density of good candidates. In the first case, the objective is to prevent degradation of performance: we want to ensure that the distribution is not significantly worsened. In the second case, the goal is to improve the distribution : make sure that there is a significantly bigger percentage of good candidates. Note that we care only about the distribution and not about the overall size of the space. Unless one is willing to explore the space exhaustively, it has no impact on the final performance.

To evaluate this, we took inspiration from a methodology called *ablation study* [MLdPM19]. Originally developed in biology, then adapted to artificial neural network studies, it consists in studying how the removal of some part of a neural network affects the final results. In a nutshell, for each hypothesis the methodology is the following: we characterize two spaces that differ only on one specific hypothesis (use of lambda, use of partial tile, etc.). One of these spaces can be a superset of the other, or they could be completely disjunct. We draw a thousand candidates randomly in each of these spaces. In the same spirit as in the last chapter, we consider that these random draws give us an approximation of the distribution of performance over the space. The distributions of performance between these two sets of draws are compared with each other. In the following sections, we will evaluate our hypothesis one after the other. Then we will briefly discuss the few experiments we feel are still missing from this list, by lack of time.

6.4 Evaluation of the combination of microkernels

In this section, we evaluate the impact of the addition of the combination of microkernels (introduced by atoms $\mathsf{U}\lambda_d$ and $\lambda_{\mathsf{seq}_d}.\,[e_{ll}]$) in our space. We consider the randomly drawn optimization scheme from the previous section, and we split them into two sets: the set of schemes that uses a single microkernel, and the set of schemes that uses a combination of microkernels.

Figures 6.10 and 6.11 compare the distribution of both sets of optimization schemes on several convolution sizes and reports the proportion of schemes with a combination over the original set of 1000 random schemes. This alternative representation is a cumulative distribution: at a given point on absciss, the ordinate represents the ratio of candidates that are at least as good as this performance threshold. For example, for Yolo9000-04 nearly all candidates have a score of at least 20%, which is why the two curves are close to 1 and falls after this threshold of 1%.

We observe that the performance with and without combination of microkernels is comparable when both possibilities are available. However, for the

last three Yolo9000 convolutions there is no microkernel amongst the ones we selected whose sizes are divisors of the problem size. We recall the sizes of these convolutions :

- Yolo9000-18 : (K, C, H/W, R/S) = (1024, 512, 17, 3)
- Yolo9000-19 : (K, C, H/W, R/S) = (512, 1024, 17, 1)
- Yolo9000-23 : (K, C, H/W, R/S) = (28272, 1024, 17, 1)

The problem is that dimensions H and W are of size 17, and microkernels unrolled by a factor of 17 are not amongst the best ones, and 17 is a prime number so there are no divisors that would have been potentially better.As a result, all single microkernel solutions are ruled out immediately and the search space is devoid of them.

The only candidates in our microkernel space that divide these problem sizes are unrolling only on the k and c dimensions, and are around 30% of peak performance. Therefore, there is no single microkernel in our space that qualify.

To complement this observation, we considered the Yolo9000-18 benchmark and the microkernel with an unrolling factor of 17 on dimension h ($\mathsf{U}_{17,h}\mathsf{V}_k$). This amounts to falling back on microkernels that do not meet the performance requirement we made. We ran 500 random optimization schemes while forcing the use of this microkernel. We observed a maximum performance of 68% of the machine peak, which is much lower than the 85% maximal performance obtained with combination of microkernels.

Therefore, this analysis shows that combination of microkernels is needed when the problem sizes are too small and not easily divisible. Else, adding it to our space does not have a significant effect on the distribution of performance.

6.5 Evaluation of the divisibility hypothesis above the microkernel

In this section, we study the impact of the divisibility hypothesis above the microkernel. We still enforce that the tile sizes above the microkernel will be multiple of its sizes, such that only a complete microkernel (and not a partial one) is always used during the whole execution. However, we consider the situation where the tile sizes above this microkernel are not divisors of the full size. For that purpose we use the atoms $\mathsf{Texct}_{\alpha,d}$ and $\mathsf{Tvar}_{\alpha,d}$ that we described in Section 3.5. For example, if a microkernel size is 3 across a dimension, then having two tiles of sizes 6 and 9 above it is now allowed. That is, we now add to our space this kind of scheme :

$$[\mathsf{R}_w, \mathsf{Texct}_{9,w}, \mathsf{Tvar}_{6,w}, \mathsf{U}_{3,w}]$$

which yields the code presented in Figure 6.12.

Notice that this divisibility hypothesis is different from one evaluated in Section 5.3, which considers a situation where we exploit a partial microkernel,

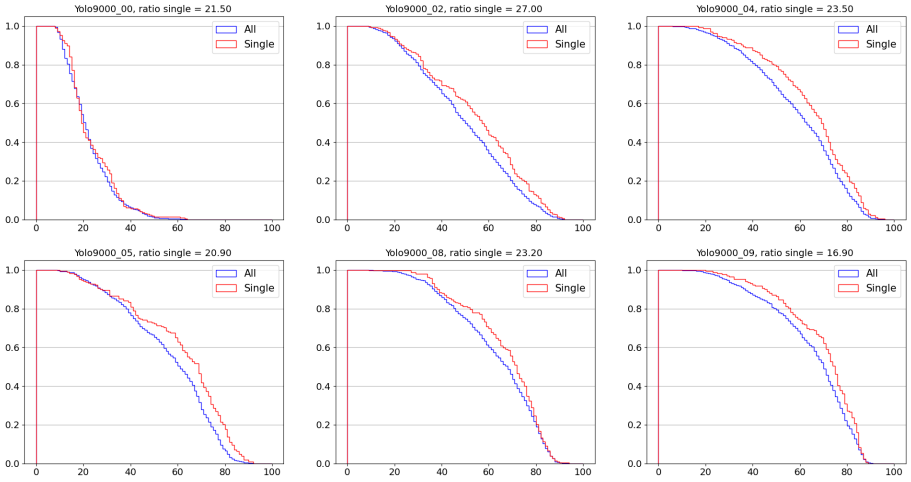


Figure 6.10: Study of the impact of the combination of microkernel on the distributions of Yolo9000-00 to Yolo9000-09. We report the cumulative distribution of the space where combinations of microkernels are allowed (All) and where these combinations are forbidden (Single). The ratio reported is the percentage of configurations using a single microkernel, on the totality of the draws.

that is, the matter of divisibility at the inner level, while here we consider the problem of divisibility at outer levels.

To compare both spaces, we consider a different random selection algorithm, with two variations, for the divisible scheme space, and the non-divisible scheme space. This is to ensure that we have the same kind of bias in the selection inside these spaces so that the comparison of the spaces is as fair as possible.

The random draw algorithm is the following:

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

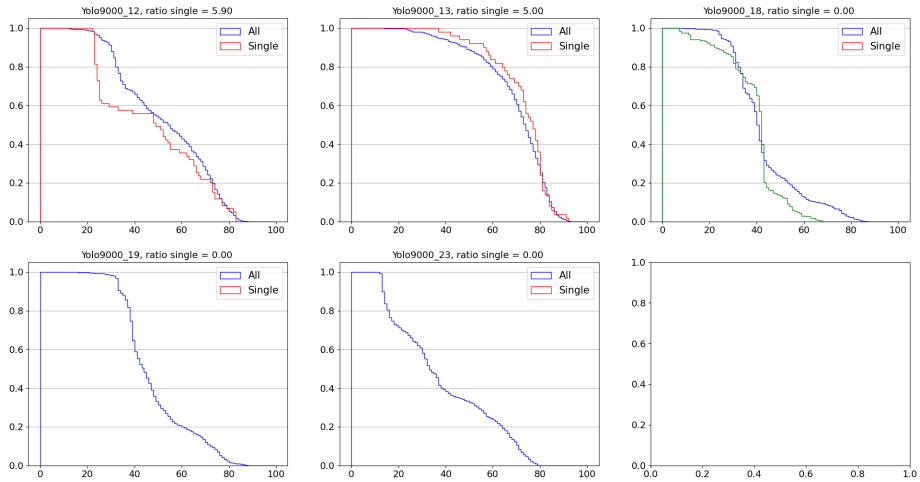


Figure 6.11: Study of the impact of the combination of microkernels on the distributions of Yolo9000-12 to Yolo9000-23 on an Intel Xeon Gold 6130. We report the cumulative distribution of the space where combinations of microkernels are allowed (All) and where these combinations are forbidden (Single). The ratio reported is the percentage of configurations using a single microkernel, on the totality of the draws. Note that all the draws for the last 3 Yolo9000s are combinations of microkernels. For Yolo9000-18, we have added (in green) 1000 runs that use a single sub-optimal microkernel. This microkernel falls outside of our classes of high-performing microkernels , but divides exactly the problem sizes. This is a situation where the combination of microkernels is particularly useful (nearly 90% of peak instead of 70%).

```
1  for (int w1 = 0; w1 < W; w1 += 9) {
2    for (int w0 = w1; w0 < w1 + 9; w0 += 6) {
3      int var_w = MIN(6, 9 - (w1 - w0));
4      for (int w = w0; w < w0 + var_w; w += 3) {
5        <basic block unrolled by 3 on dimension w>
6      }
7    }
8  }
```

Figure 6.12: Example of non-divisibility at outer level : $[\mathsf{Texct}_{9,i}, \mathsf{Tvar}_{6,w}, \mathsf{U}_{3,w}]$

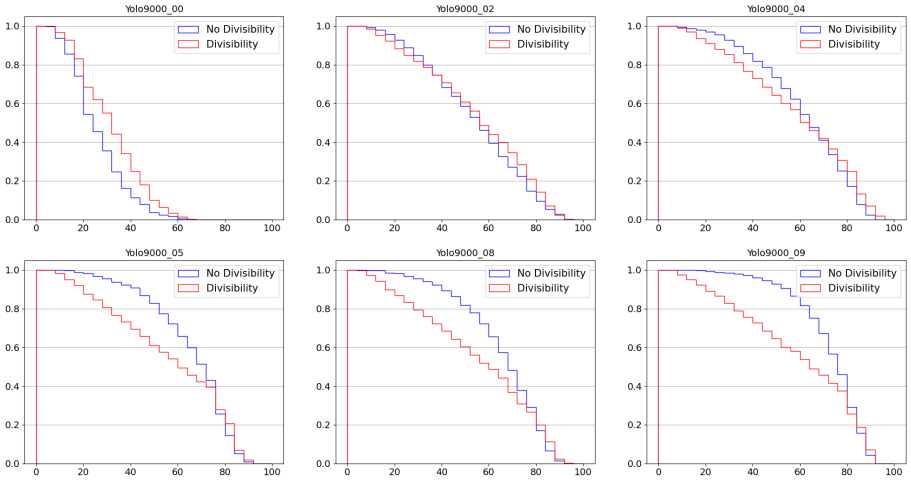


Figure 6.13: Study of the impact of the divisibility constraint on Yolo9000 convolutions 00 to 09, by choosing randomly 1000 configurations in a space with the divisibility constraint on the tile sizes above the microkernel, then without the divisibility constraint.

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

Figure 6.13 shows the distribution of 1000 random optimization schemes, for the divisible space, and for the non-divisible space. The non-divisible space looks better, as the cumulative distribution of candidates is often higher in the non-divisible case than in the divisible case. This means that for most thresholds of performance, there will be a bigger proportion of candidates above the threshold in the non-divisible space than in the divisible one. Nevertheless, the best candidates are most of the time at the same level. This shows that by having combination of microkernels with the divisibility constraint, we do not lose performance in respect to the non-divisibility space, even if we would need a bit more trials to find them on average. Therefore, restricting our search space to only divisible tile sizes with lambda combinations is a valid hypothesis.

6.6 Future Works

There are a few hypotheses that we did not test by lack of time, although we have all the necessary infrastructure to do so. For example, it would have been nice to prove definitely that the pruning of the microkernel space does improve the final distribution. We have preliminary results on the matter that hint that a vast majority of candidates without microkernels have performance below 1%

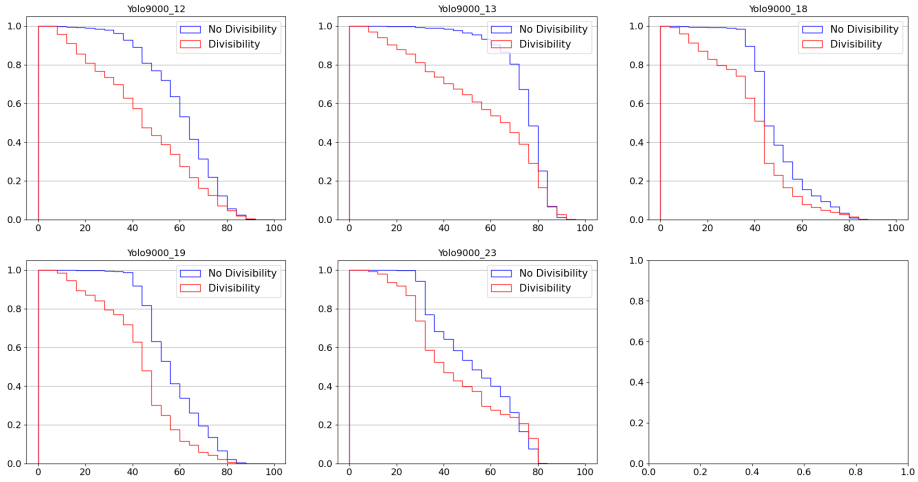


Figure 6.14: Study of the impact of the divisibility constraint on Yolo9000 convolutions 12 to 23, by choosing randomly 1000 configurations in a space with the divisibility constraint on the tile sizes above the microkernel, then without the divisibility constraint.

of peak performance, but this needs to be reinforced by a more exhaustive study. We strongly believe that this pruning is decisive but a thorough confirmation would still have been useful, be it only for completeness. We could find some interesting corner cases for which it is profitable to search outside of these selected microkernels. The case of Yolo9000-0 (whose sizes are (K, C, H/W, R/S) = 32, 3, 544, 3) is interesting in this regard.

For this specific problem, the c dimension that we usually iterate just over the microkernel to get reuse inside the basic block is of size 3. This is too small to amortize the costs of loads and stores. Indeed, the inner part of the code is usually structured this way :

```
1  ...
2  <loads from Output>
3  for (int c = c0; c < c0 + Tc; c++){
4    <basic block>
5    }
6  <stores to Output>
7  ...
```

As we explained before in Section 2.1.1 , we apply scalar promotion to make sure that all accesses to *Output* are promoted in registers, therefore the cost of accesses is much cheaper in the basic block than out. Making sure T_c is big (at least 32 iterations or more) is important because it allows the iterations of the inner loop to outweigh the cost of the enclosing loads and stores. In the specific case of Yolo9000-0, as dimension c is of size 3 we cannot have a tile on c that is big enough. We were able to find better solutions by broadening the microkernel space to microkernels that were unrolled on other reduction dimensions such as

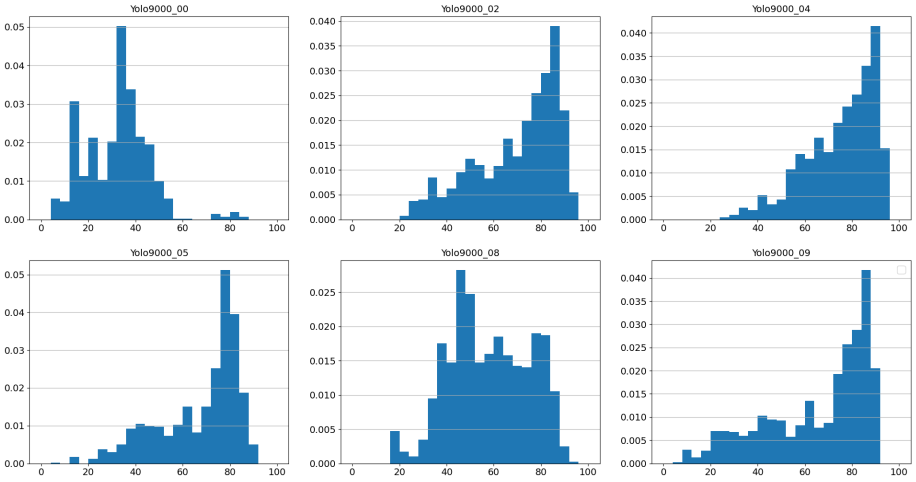


Figure 6.15: Yolo9000 00 to 09 distribution with Ioopt permutation in sequential

r and s, which solved the amortization problem.

We wish we had the time to characterize properly the impact of the choice of permutation on the final result. In a nutshell, does restricting the possible permutations in advance improve the quality of the search space by eliminating a large proportion of bad candidates, or are there so many sensible choices that trying to fix the permutation beforehand has no effect apart from potentially ruling out sensible candidates? Section 5.4.1 presents intuitions for the importance of correctly selecting a permutation and why it could have an impact on performance. However, experiments proved that the random search described in Section 5.4.5, despite imposing no constraint on permutation, still allows for a good convergence rate in many cases. This could be explained by a sampling argument (a good proportion of all permutations have good properties, thus we do not need to select a good one a priori). Comparing a space where permutations are fixed amongst a selected set with a space where permutations are free would give us a good hint. Figures 6.15, 6.16, 6.17 and 6.18 show the distributions of Yolo convolutions where the permutation has been fixed by Ioopt - so the order of the dimensions in the scheme is fixed, only the sizes change. This should be compared to the distributions obtained for a space where permutation is free, as we do in Figure 5.11. We did not have the time to analyze these results precisely.

Another point is that it would have been interesting to test the use of a fully-flexible partial microkernel. As we recall in Section 5.3, we have both theoretical and experimental arguments that hint toward a possible performance penalty for a single-microkernel strategy. However, a comprehensive study would be useful to corroborate or not this effect on real-life problem sizes.

Speaking of which, a fully-flexible microkernel could have been useful to test our cache model in a more meaningful way. Indeed, we have seen in Section 5.4.2

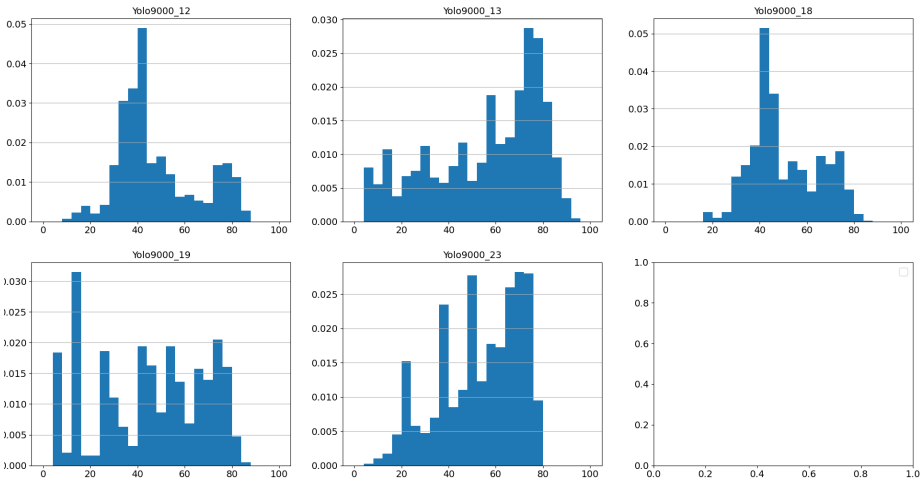


Figure 6.16: Yolo9000 12 to 23 distributions with Ioopt permutation in sequential

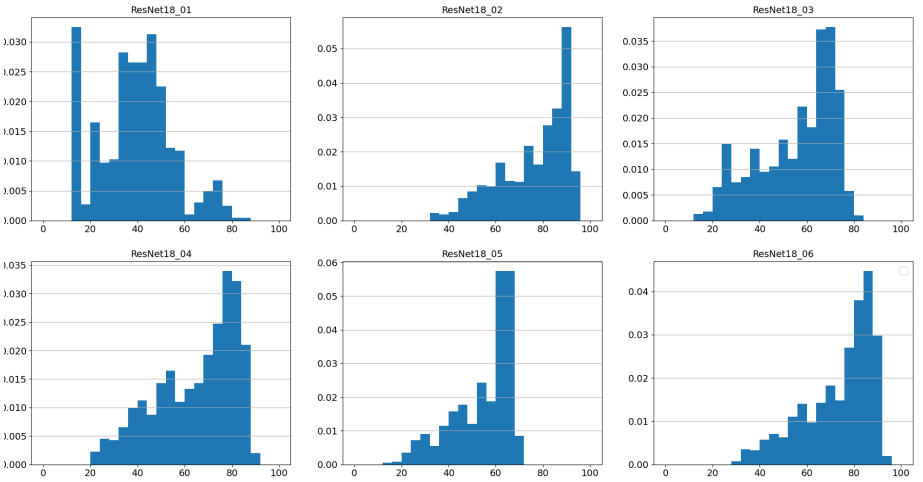


Figure 6.17: ResNet18 01 to 06 distributions with Ioopt permutation in sequential

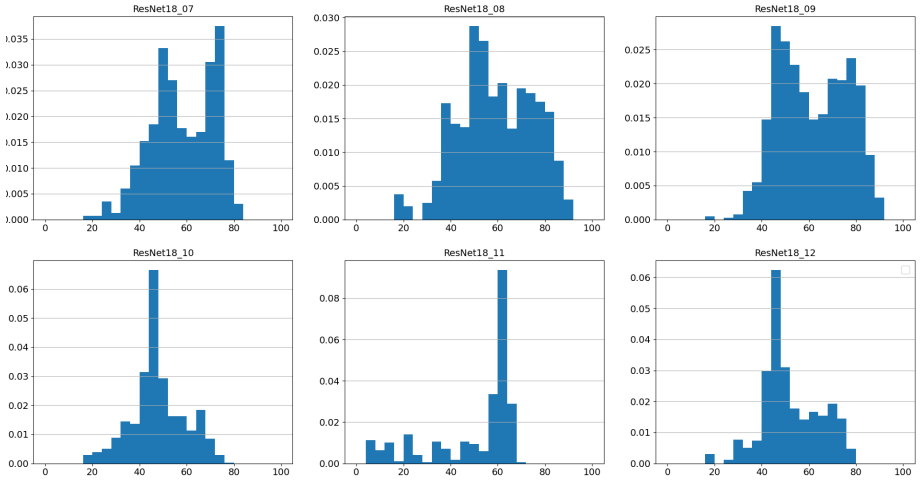


Figure 6.18: ResNet18 07 to 12 distributions with Ioopt permutation in sequential

that Ioopt as a tool delivers a theoretical tiling solution. This tiling can only be approximated in practice if we have a divisibility constraint on loop sizes. The lack of success of this approximation could be explained in two different ways :

- Inadequacy of the model (the proposed solution is bad)
- Inadequacy of the approximation (the proposed solution is good but our approximation is too far from it)

Having a fully-flexible microkernel allows us to get rid of the divisibility constraint entirely and therefore makes it possible to test implementations that are much closer to the solution given by Ioopt. This would allow us to settle the case between the two possible explanations.

Chapter 7

Discussion

7.1 Previous work and inspirations: Telamon

This thesis is inspired in part by the PhD work of Ulysse Beaugnon [BPP+17]. Telamon is a library implemented by Beaugnon which defines optimization as a search in a flat space. This contrasts with the "rewriting" technics used in most compilers where optimization consists in rewriting an Intermediate Representation while preserving semantics. One limitation of this approach is that some optimizations are not immediately available and thus depend on a necessary rewriting pass implicitly. For example, loop vectorization sometimes needs a loop permutation or a strip-mining to be legal. As a result, it is not clear whether a given optimization pass is beneficial by itself or only because it allows another one.

The key idea of Telamon is to expose every choice upfront - so that the search space is "flat" - and to handle the dependencies across optimization choices by constraining the set of legal solutions. For example, one can express that it is illegal that a vectorized dimension is not at the innermost level.

That way, choices can be made in any possible order, and it is easier to identify which choices contribute to performance.

Another key idea was that in some case it should be possible to predict things about a candidate even before it is entirely specified, that is before every decision has been taken. In our case, the metric we wanted to predict was the execution time, and given that the architecture is quite predictable it is possible to give a lower bound of the time taken by a given candidate. This is not groundbreaking by itself - after all, 0 seconds is a perfectly valid lower bound for any possible implementation. It becomes interesting when this lower bound becomes tight enough to allow for a massive pruning of the global search space. Telamon is in practice an autotuning tactic. The point is to explore the space of possible candidates, begin by specifying some of them entirely, run and evaluate them on actual hardware, and then use this feedback to prune some parts of the space entirely. Whenever taking a decision (that is refining a candidate)

Telamon uses its performance model to predict a lower bound of the time taken by this partial specification. If this lower bound is higher than the best score we already saw, then this candidate is pruned entirely.

We evaluated the size of the pruned space to be orders of magnitude smaller than the total space, which in turn validated the usefulness of the approach. My first contribution to this work was to implement another type of exploration in this space. This consisted of a reimplementation of the work of Desmay and al. [dMRVP09]. This work presents a search heuristic based on a decision tree whose nodes are *multi-armed bandit*.

The original search strategy was a greedy exploration. This work tries to be smarter by finding a tradeoff between exploring new paths and going back to nodes that have proven to yield good candidates. The ratio of "finding new information/exploiting information we already have" at a given level (exploration versus exploitation) is dynamic and depends on the previous results found there. The hope was that this algorithm would lead to earlier pruning by finding good candidates in less time. A greedy search could indeed get stuck in "bad parts" of the search tree and thus miss the opportunity to find a good candidate that would have allowed to prune these parts. Indeed the results were quite convincing and allowed us to expand the search space further - by adding tiling as part of the decisions to make.

This work in Telamon inspired the rest of my PhD in different ways. It was an important step in envisioning optimization otherwise than successive rewriting passes on an Intermediate Representation. Telamon also helped me to emphasize the importance of the mere definition of the optimization space, which is often conflated into the question of how to explore it. It also emphasizes a white box approach. As opposed to other works that rely on machine learning tools to discover the best schemes of optimization and give up on understanding the relative contributions to the performance of each decision, Telamon allows one discriminating more easily between relevant and irrelevant features. Telamon could even allow expert users to start their search with partially specified implementations and custom optimization decisions with guarantees that they would never be invalidated by further rewriting, but we did not investigate further.

This background also helped me understand better the autotuning strategy used by AutoTVM.

7.2 Chronology and lessons learned during this PhD

This document as every other thesis dissertation presents an after-the-fact reconstruction of the directions I took during these three years. I believe taking the time to reflect on the order we did things can be valuable as some work should have retrospectively been done earlier, and we were mistaken on a few aspects. While the principles guiding the search for a microkernel were right-

fully set early, one of the difficulties we met was finding the right way to choose the outer level tiles and to justify this choice accordingly.

7.2.1 Perfectly nested loop and divisibility constraint

Another choice we struggled to justify was the need to restrict ourselves to perfectly nested and divisible loops. This choice yielded great results and was based on reasonable assumptions - namely, that a simpler code generation has less chance to expose bugs, and that a too contrived control flow can severely hurt performance in the worst case. This has been observed with some of the code generated from polyhedral tools before - where skewing can lead to unreasonable loop nest shapes. This work [Bas04] describes an algorithm that given any polyhedral schedule generates a valid corresponding imperative code. Moreover, the shapes of microkernels were already introducing some hard divisibility constraints that

1. cannot be removed without help from architecture-specific instructions (See Section 5.3.1)

2. even with these special instructions induce some performance drops when the problem size is small and is not a multiple of the underlying microkernel size as seen in Section 5.3.3

So the idea to restrict ourselves to perfectly nested loops made sense at the inner levels at least. However, these intuitions were not a strong enough argument to defend the idea of restraining divisibility at the outer level too. Or at least, we tried to justify that this was a necessary choice for getting performance, while it was merely a convenient way to restrict the search space and to exploit what we already had in terms of code generation.

This error (in retrospect) was a consequence of the organic development of my work. At some point in my thesis, I had a quite limited code generation implementation that only supported perfectly nested loops. We decided, as an exercise and also to get beyond a purely implementation-oriented work, to *make the best of what we had* and strive to find a search process to implement an optimized convolution. The results turned out to be much better than expected. Thus I tried to expand the code generation just enough to support the few problem sizes that were still not performing as well as we hoped. The solution to this was the so-called lambda specifier described in Section 3. This, along with a rather naive tiling algorithm for the outer level, allowed us to reach levels of performance close to or higher than state-of-the-art tools such as TVM or OneDNN. Here we decided to use our concurrents as a baseline only, without taking the time to evaluate each of the design decisions in isolation with our tools. Moreover, instead of presenting these assumptions (perfectly or nearly perfectly-nested loops, use of a basic and rather ad-hoc metric) as a simplification of the implementation process, we tried to prove that they were a key point in reaching a high-level of performance.

7.2.2 Random is all you need

A reflection that made its way far too late into the global process was to evaluate our search model against a flat random search. Random search strategy is counter-intuitive and very often much more efficient than we think. The birthday paradox is a great example of this. The birthday problem asks, for a group of n people, the probability of having at least two of them sharing the same birthday. It turns out that this probability reaches 50% for a group of 23 people or more, which is much less than most people would have intuitively said (thus the so-called "paradox").

At some point in our candidate selection, we allowed ourselves to run up to 200 candidates selected by a metric of some sort. The best out of these 200 candidates became our final implementation. This "autotuning" budget was deemed reasonable for several reasons :

- It was much less than our main concurrent, TVM which was running thousands of candidates
- It was possible to run these candidates in less than 20 minutes, which meant at the time that this running time was dominated by the time it took for the Ioopt solver to run
- As the space of all possible candidates was two orders of magnitude bigger than this, we felt like succeeding in selecting 200 candidates amongst which we were confident to find a competing implementation was akin to find a needle in a haystack.

We fell into the birthday fallacy here to some degree. When trying 200 candidates randomly, we have an 83% chance that at least one of them will belong to the first percentile of performance, and a 98% chance that at least one of them belongs to the two first percentiles. That is, we can be highly confident that at least one of the candidates we tried will be better than 99% of all possible candidates present in our search space. Finding a candidate in the first percentile can be excellent or terrible, depending on the space we are looking into. This is why we began at some point to characterize the quality of the *search space* itself separately from the quality of the *search heuristic*.

It is not difficult altogether to make a selection method better than random *on average*. The crucial point is that we do not care about the average here, what matters is the performance of the *best* candidate we tried.

Another key point here is that what matters is the quality of the *distribution* of a space, not its *size*. A good pruning of a search space does not only reduce the size of the initial space without removing the best candidates, it *improves* the density of good candidates in it.

Therefore random makes a useful baseline to compare against. Given a budget of n trials, and choosing some confidence level c, we can compute the percentile level p that we expect to get with a random search. That is, we are looking for p such that there is at least c % chance that we can find a candidate among the n we took that performs better than p% of all possible

other candidates. Assuming that we consider c to be "sufficiently high", a metric is worth trying only if it is able to identify at least one candidate much better than p - for example if p turns out to be 99% then our metric should be able to confidently find a candidate better than 99.5% or 99.9% of all candidates.

7.2.3 One-shot versus learning

Originally this PhD was biased toward the search for a one-shot solution: a pure compiler flow that would have produced a single candidate without any autotuning. The introduction of AutoTVM made us reconsider this choice and allow ourselves a budget of trials. Comparing our autotuning process to the more sophisticated one used by AutoTVM raises some interesting questions. We spent time trying to identify which parts of the process were critical in finding performance. We ended up baking in our heuristic much more hypotheses than AutoTVM, such as the use of a pre-selected microkernel. While these hypotheses did not allow us to get rid entirely of a light trial-and-errors phase for discriminating the last candidates, it still let us use a much simpler research heuristic than AutoTVM - to the point where this heuristic became trivial when we decided to resort to random search. Our results hint that the selection of a microkernel is the most critical part and that once one fixes this choice many possibilities are not hard to find. This would mean that the training of AutoTVM is only necessary to find such a microkernel at the lowest level. Testing this hypothesis would make for interesting future work.

Chapter 8

Conclusion and future work

In this dissertation, we presented many directions in which we could derive an optimized implementation for a tensor computation (and in particular for a convolution), knowing the problem sizes ahead of time. It extends mostly the principles used in several BLAS implementations such as BLIS or GotoBLAS. The initial point was to challenge some of the usual assumptions and practices of the field such as the use of a single microkernel or the use of packing at every level. The goal was also to leverage some of the theoretical work done in performance modeling, with works such as Ioopt. Exploring these design choices led us to implement a code generation framework and an experimental platform. We also had to think about a robust way to compare fairly different approaches and optimization choices.

8.1 Tensor computation optimizations

To explore easily the design space of a tensor computation, we implemented a code generation framework along with an experimental platform that we describe in Chapter 3 and 4. The guidelines behind our work were that making the code generation as simple as possible would make it more portable, less error-prone, and possibly more efficient by avoiding excessive control flow. We expanded the basic GotoBLAS microkernel strategy by finding out that many efficient microkernels existed in addition to the ones usually used. Section 5.3.2 proves that the use of a microkernel whose sizes do not divide the problem size perfectly can be detrimental to performance. The trick of combining microkernels with our new *Lambda* constructs allowed us to find a way to avoid the use of such a partial microkernel.

Then we looked for a way to choose the sizes of the tiles over the microkernel. In the sequential case, it is mostly a matter of accommodating the cache behavior. We tried different ways to leverage the ideas present in Ioopt, first by approximating its solution directly, then by reimplementing a specialized version of the data movement volume estimation, and finally by an alternative

algorithm that tries to couple the search of a dimension permutation with the search of tile sizes.

At some point, we had to take a step back and reflect on the way we evaluate the adequation of a search metric. This led to the work described in Section 5.4.5 that shows how the random strategy can be used as a baseline for search strategies. We introduced a clear cut between the definition of a space and the search heuristic. This allowed us to compare two search spaces against each other by looking at their performance distributions. This methodology called *ablation study* is a good way to identify the contribution of a specific feature to the performance, by asking whether this feature improves or not this distribution. With this methodology, we were able to characterize which features had a potential benefit and which ones did not.

We also show that we can compete with the state-of-the-art in terms of performance, be it in sequential or in parallel. However, we did not succeed in proving that any of our search strategies was better than a random search.

This does not necessarily mean that relying on a cache model is never beneficial though.

In a nutshell, we made the following contributions:

- We developped a code generation tool that allows easy experimentation and modularity, along with an experimental platform
- We defined multiple search spaces in which we incorporate expert knowledge
- By decoupling of core-level and memory-level considerations, we cut the complexity of the search space exploration

Our work also provided some insights, first with respect to GotoBlAS expert approach:

- There are more available choices of microkernels than what is usually used
- By combining these microkernels, we found an alternative to partial tile strategies
- At least in sequential, GotoBLAS Tiling tactic seemed unconclusive

And then also with respect to the Ansor exploration approach:

- Incorporating expert knowledge directly in the definition of the search space improves convergence
- It is very likely that Ansor spends most of its learning budget rediscovering tactics known by experts
- Random exploration can be surprisingly efficient when the space is well-chosen

8.2 Future work

8.2.1 Packing and Layout

We've seen in Section 5.5 that while building a packing decision scheme was an initial target of our work, it has proven less useful than initially expected on the architecture we were targetting. There are good reasons for that, in particular the fact that on AVX512 a vector register holds the same number of bytes as a line of cache, which means that spatial locality of the code is ensured automatically as soon as we have a vectorized code.

However, the question of whether to pack or not and at which level is still open at least for older architectures than the one we were using. From the few experiments we did on the matter it looks like packing can sometimes improve performance critically on AVX2. I did not have the time to investigate a systematic search on the matter, so the reasons for these improvements remain opaque for now. This means that the work engaged in the implementation of packing inside the code generation remains unexploited for now.

Therefore, one of the initial goals of this PhD which was to evaluate systematically the contribution of packing and data layout to the performance is still an open question. It is also motivated by the late work started on Tensor Contraction. The presence of multiple, possibly small dimensions could provide further motivation for a strong data-shuffling decision process. Packing was initially designed under the assumption that loops were perfectly nested. Therefore, the subsequent addition of partial tiles described in Section 3.5 is not tested as exhaustively as we would like, which implies it may need some additional work.

8.2.2 Compiler as a language ?

One of the main takeaway from this work was that architectures became so complex that building a useful analytical model is a bewildering task. We found out that while some common principles - such as the way to build a microkernel - can be derived from expertise, it can be extremely difficult to go beyond that in terms of predictability.

When a 5% speedup is considered a significant and desirable improvement, there is at the moment no way to avoid the intervention of an expert. In this setting, compilers can sometimes be counterproductive. Indeed, some optimizations can get in the way of what the expert is trying to produce at this precise moment, and compiler options do not necessarily provide the required level of granularity. The point here is that while compilers do a great job at exposing heuristics that generate a good binary in the average case (often better than an expert would do in the same situation) they are not convenient for the research and experiment process. Intermediate Representations, either of compilers or larger projects such as MLIR or TVM are more suitable as the targets for an automatic generation (usually from a higher-level representation) than to be directly manipulated by human users. As such, they are verbose, they lack

most of the tools that are available for mainstream languages (completion, linting, typing...). We have also seen that compilers may be much better at some tasks than humans (register allocation for example), but fail at others. There is already a great area of research in the domain of meta-programming. These languages provide users tighter control over code generation.

8.2.3 Experimental evaluations

We already mentioned in Section 6.6 that many experiments were still left to be done. In particular, we need to evaluate properly how the pruning of the micro-kernel space impacts the quality of the search space. We also need a definitive evaluation of performance models such as Ioopt. In general, the methodology we tried to develop ("ablation studies") would need some further refinements, in terms of which parameters should be chosen to compare two spaces, depending on what we strive to optimize.

Appendix A

Résumé étendu

A.1 État de l'art

L'optimisation de calculs linéaires est un problème bien étudié qui a mené à la création d'un grand nombre d'outils et d'un certain nombre de principes de design. Cette section présente quelques uns d'entre eux, ainsi que les concepts les plus importants dans cette optimisation, à commencer par la notion de micronoyau et de tuilage hierarchique, que nous allons développer plus longuement. Avec GotoBLAS, nous nous concentrerons plus particulièrement sur l'optimisation de multiplication de matrices. Bien que cela puisse paraître restrictif, nous verrons ensuite comment étendre les mêmes techniques à d'autres opérations, en particulier la convolution.

A.1.1 GotoBLAS/BLIS : construction systématique de multiplications de matrices efficaces

BLAS signifie *Basic Linear Algebra Subroutines* et consiste en un ensemble d'interfaces standards, ainsi que d'implémentations pour certaines routines d'algèbre linéaire. Nous allons étudier plus particulièrement l'une de ces implémentations, appelée GotoBLAS, car elle a aidé à poser certains des principes les plus importants, notamment ceux de micronoyau et de tuilage.

A.1.2 Micronoyau

Un micronoyau est une implémentation de petite multiplication de matrice optimisée pour tirer parti des caractéristiques du CPU. Un tel calcul est représenté dans la figure **??** L'idée derrière GotoBLAS est de se servir de ce micronoyau comme d'une brique de base pour le calcul total.

Il y a plusieurs paramètres importants dans le design d'un micronoyau :

1. Utiliser les opérations et registres vectoriels de la machine
2. Amortir la latence d'une opération vectorielle

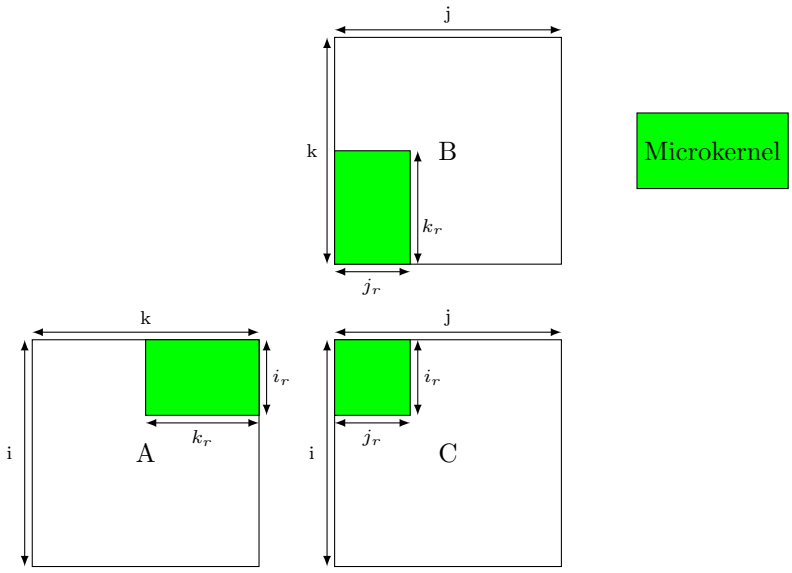


Figure A.1: Micronoyau GotoBLAS

3. Exploiter au maximum le parallélisme d'instruction pour saturer les ressources disponibles.

En pratique, on y parvient en jouant sur les facteurs de déroulage du code. Les facteurs exacts dépendent des propriétés de la machine sur laquelle on optimise, en particulier de la latence des instructions et du nombre de registres. Comme nous le verrons, il y a en général plus d'une seule solution répondant à des critères de bonnes performances, mais GotoBLAS choisissent de n'en implémenter qu'une seule par machine.

A.1.3 Tuilage

L'utilisation de micronoyau permettait d'exploiter au mieux les caractéristiques au niveau du cœur. Un autre aspect important des machines modernes est qu'au lieu d'utiliser une simple mémoire "à plat", elles exposent une hiérarchie de caches. Cette hierarchie permet d'exploiter deux observations importantes :

1. Une donnée accédée en mémoire à un point du programme a de grandes chances d'être réaccédée dans un futur proche (localité temporelle)

2. De manière similaire, une donnée proche en mémoire d'une donnée accédée à un point du programme a de grandes chances d'être accédée rapidement (localité spatiale).

Les caches sont ainsi construits de façon à permettre un accès plus rapide à une donnée qui vient d'être accédée, ainsi qu'à ses voisins proches, en les maintenant dans une zone dédiée jusqu'à éviction.

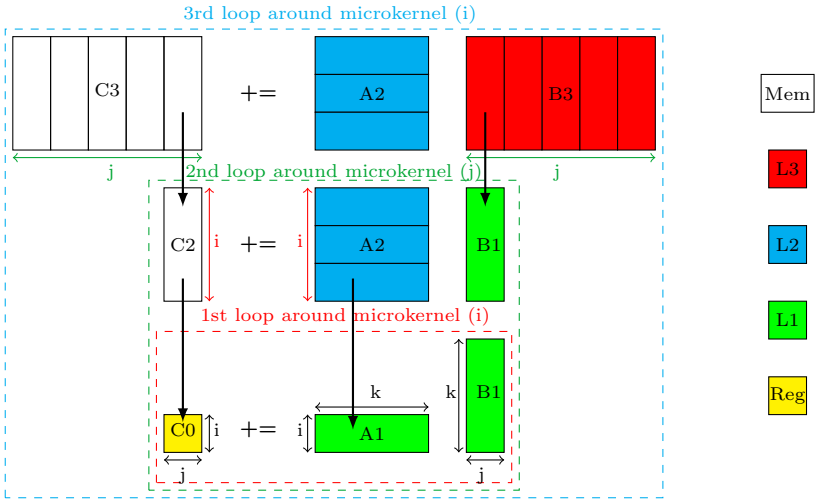


Figure A.2: Stratégie de tuilage BLIS

Côté logiciel, le tuilage hiérarchique est une façon de réaliser un programme de manière à maximiser la localité des accès. Le schéma **??** illustre la stratégie de tuilage utilisée par BLIS. L'idée est simplement d'alterner les tenseurs pour lesquels on cherche à exploiter la localité à chaque niveau.

A.1.4 Multiplication de matrice et convolution

On a jusqu'ici présenté des optimisations dédiées à la multiplication de matrices. Cependant, ces optimisations se généralisent à une classe plus large de programme, et en particulier à la convolution. En effet, la deuxième opération peut être ramenée à la première.

Pour le montrer, voici d'abord la formule d'une multiplication de matrices :

$$C[i,j] += A[i,k] * B[k,j]$$

Cette formule utilise la notation d'Einstein, à savoir que les indices n'apparaissant qu'à droite du signe égal font l'objet d'une réduction implicite. Dans le cas présent, il y a une somme sur k implicite pour chaque i, j.

Voici ensuite la formule de la convolution, toujours avec la même convention :

$$O[b,h,w,k] += I[b,h+r,w+s,c] \times K[r,s,c,k]$$

Une série de transformation permet de rapprocher ces deux formules. En effet, si nous appliquons les groupements de dimensions suivants :

$$\begin{cases} b,h,w \Rightarrow bhw \\ r,s,c \Rightarrow rsc \\ k \Rightarrow k \end{cases} \tag{A.1}$$

La formule de la convolution devient :

$$O[bhw, k] += \hat{I}[bhw, rsc] * K[rsc, k]$$

où $\hat{I}$ est le résultat d'une transformation appliquée à I, qu'on peut définir par la correspondance suivante :

$$\hat{I}[b][h][r][w][s][c] = I[b][h + r][w + s][c]$$

La dernière formulation rend le parallèle avec la multiplication de matrices évidente, modulo quelques renommage (O en C, bhw en i ...). Les premières implémentations de convolutions optimisées consistaient d'ailleurs à faire ces transformations explicitement pour ensuite appeler une bibliothèque de multiplication de matrices. Cette étape de préparation des données appelée im2col peut être évitée par une implémentation directe. Ces implémentations directes peuvent néanmoins utiliser les mêmes techniques.

A.1.5 Autotuning

À l'opposé d'une approche "experte" de l'optimisation exemplifiée par GotoBLAS, une autre stratégie populaire existe : l'autotuning. Cette stratégie consiste dans un premier temps à exposer une spécification sémantique du problème à optimiser, ainsi qu'une série de choix d'optimisations (entre autre des choix de facteurs de déroulage, de stratégie de vectorisation ou de tuilage). Dans un second temps, le but est de confier à un algorithme de recherche le soin de trouver une combinaison de choix d'optimisation efficace. Cet algorithme n'est pas guidé par un modèle de performance analytique, mais par une recherche empirique : les différentes combinaisons sont exécutées puis évaluées directement sur la machine, dans le but de faire converger la recherche à l'aide de méthode d'apprentissage. Il s'agit donc d'une forme de recherche par essai et erreur, les algorithmes d'apprentissage pouvant varier (descente de gradient, algorithme génétique ou autre). Un exemple très puissant d'une telle méthode est Ansor, basé sur le framework TVM. Un exemple d'un code TVM est donné dans l'encadré **??**.

Si cet outil permet d'obtenir des performances impressionnantes, les temps d'entrainement sont souvent très grands et peuvent demander l'exécution de plusieurs milliers ou dizaines de milliers de candidats.

A.1.6 Autres outils

D'autres outils d'optimisation existent. Par exemple, les outils polyédriques permettent de fournir automatiquement une solution de tuilage répondant à un certain problème d'optimisation. Diesel [ERR+18], Tiramisu [BRR+19], Polly [GGL12] ou Tensor Comprehension [VZT+19] sont deux exemples de tels outils.

D'autres outils, tels que MLIR [LAB+21] offre une plateforme permettant d'implémenter des langages spécifiques plus aisément, et donc d'intégrer des invariants au cœur de la génération de code.

```
C = te.compute((N,M),
        lambda i, j: te.sum(A[i, k] * B[k, j], axis=k),
        name="C")

s = te.create_schedule(C.op)

# START OF Scheduling (this is optional)
axis_i, axis_j = C.op.axis          # get axis list
axis_k, = C.op.reduce_axis          # get reduce axis list
axis_i_o, axis_i_in = s[C].split(axis_i, nparts=8)
axis_j_o, axis_j_in = s[C].split(axis_j, factor=64)
s[C].reorder(axis_i_o, axis_j_o, axis_i_in,
        axis_k, axis_j_in)
s[C].vectorize(axis_j_in)       # vectorize inner loop
s[C].unroll(axis_j_in)          # unroll inner loop
```

Figure A.3: Exemple d'utilisation d'Ansor pour une multiplication de matrices

A.1.7 Quels axes de travail ?

On voit donc que l'état de l'art propose un éventail riche d'outils et de méthodologie d'optimisation, dont l'une au moins fait référence dans la littérature, à savoir l'approche par tuilage hierarchique et micronoyaux de GotoBLAS. Cette méthode présente néanmoins certains inconvénients, comme le fait de devoir implémenter manuellement les micronoyaux, et de ne pas adapter le code à la taille du problème. D'autre part, l'approche par autotuning atteint des performances intéressantes mais décale l'expertise du champ de la compilation vers celui des méthodes d'apprentissage automatique. De plus, cette technique ne permet pas d'exploiter des résultats bien connus du champ. L'objectif est donc d'aboutir à un compromis entre les deux approches.

A.2 Génération de code

Avant de chercher à trouver des techniques d'optimisation précises, il faut se demander comment explorer et évaluer les différents candidats de façon efficace. En effet, la taille de l'espace d'optimisation que nous souhaitons explorer ne nous permet pas d'envisager d'implémenter nos candidats manuellement. À cet effet, nous avons implémenté une bibliothèque de génération de code afin de pouvoir aisément tester une grande variété d'implémentation.

Cette bibliothèque a été codée dans le langage OCaml. Contrairement à TVM qui décrit une implémentation par le biais d'une série de transformations classiques telles que le "splitting" de boucle ou l'échange de boucles, l'idée ici est d'avoir un modèle qui s'attache à décrire la fome du code qu'on cherche à générer. Pour cela, le principe est de fournir un ensemble d'atomes à l'utilisateur, chaque

atome correspondant à un niveau du nid de boucle. On suppose ici que l'espace des itérations est rectangulaire et entièrement permutable.

Les atomes disponibles sont les suivants :

- R_d insère une boucle la plus externe sur la dimension d. Elle apparait au plus une fois sur la dimension d, et itère la tuile interne jusqu'à épuisement de d.
- V_d insère une opération vectorisée selon d, qui correspond virtuellement à une boucle de la taille d'un vecteur de la machine. Elle ne peut apparaître qu'en position la plus interne du nid de boucle.
- $\mathsf{T}_{\alpha,d}$ introduit une tuile de taille α sur la dimension d. α doit donc être un diviseur de la taille de d.
- $\mathsf{U}_{\alpha,d}$ déroule selon la dimension d d'un facteur α. À nouveau, α doit diviser la taille de d.
- $\mathsf{U}\lambda_d$ insère un déroulage paramétrisé. Ce paramètre sera instancié de multiples fois avec différentes valeurs à un niveau supérieur du nid de boucle.
- $\lambda_{\mathsf{seq}_d}.[\ell]$, où $\ell = [(r_i, a_i)]_{1 \leq i < s}$ est une liste de $s \geq 2$ paires introduisant une séquence de s boucles de taille r_i selon la dimension d. Cet atome génère des tuiles non parfaitement imbriqués, ce qui permet d'échapper à la règle de divisibilé selon une dimension.
- $\mathsf{Texct}_{\alpha,d}$ insère une tuile dont l'empreinte est d'exactement α sur la dimension d, Cela peut impliquer des itérations partielles sur le bord du domaine.
- $\mathsf{ScalP}[d_0, d_1, ...]$ sélectionne tous les accès à des tenseurs qui n'inclut pas les dimesions $[d_0, d_1, ...]$, et déplace ces accès à l'extérieur de ce niveau de boucle.
- Pack_A crée un buffer temporaire qui correspond au sous-ensemble de A qui est accédé par les niveaux internes du nid de boucle, et modifie le code de manière à initialiser ce buffer, puis à changer tous les accès au tenseur original par un accès à ce buffer à l'intérieur du nid de boucle.
- $\mathsf{PackT}_A\,[ell]$ fait la même chose que Pack_A mais permet également d'opérer une permutation de dimensions entre le tenseur original et le buffer créé. Cette permutation est spécifié par ell.
- $\mathsf{Tvar}_{\alpha,d}$ insère une boucle dotée d'une borne variable, ce qui permet d'utiliser ensuite $\mathsf{Texct}_{\_,\_}$ à un niveau supérieur.
- $\mathsf{ExternalCall}_{name,\alpha,d}$ insère un appel à une fonction externe $name$. Cette fonction doit prendre un paramètre α sur la dimension d. Cela permet l'utilisation d'un micronoyau écrit à la main.

```
1  for (k0 = 0; k0 < K; k0 += 32){
2    for (j = 0; j < J; j += 1){
3      for (i = 0; i < I; i += 1){
4        // Tiling dim k by 32
5        for (k = k0 ;
6          k < k0 + 32;
7          k += 1){
8            scal_1 = C[J * i + j];
9            scal_3 = A[K * i + k];
10           scal_4 = B[J * k + j];
11           scal_2 = scal_3 * scal_4;
12           scal_0 = scal_1 + scal_2;
13           C[J * i + j] = scal_0;
14       }
15     }
16   }
17 }
```

Figure A.4: $[\mathsf{R}_k, \mathsf{R}_j, \mathsf{R}_i, \mathsf{T}_{32,k}]$

- $\mathsf{Tpar}_{\alpha,d}$ a la même sémantique que $\mathsf{T}_{\alpha,d}$, mais est exécuté en parallèle.
- $\mathsf{Tfused}_{[(d1,i1),(d2,i2)...]}$ iterates on all dimensions $[d1, d2...]$ in a unique loop and executes it in parallel.
- $\mathsf{Tfused}_{[(d1,i1),(d2,i2)...]}$ itère sur l'ensemble des dimensions $[d1, d2...]$ en une boucle unique et est exécuté en parallèle.

Nous allons donner très rapidement quelques exemples d'utilisation. Figure A.4 montre un exemple de tuilage simple. Figure A.5 montre un exemple de déroulage.

Toutes ces configurations permettent d'exprimer l'ensemble des transformations que nous pensons nécessaire à l'obtention d'une performance optimale. Nous qualifions cet ensemble d'espace de recherche, c'est au sein de cet espace qu'il nous faudra ensuite trouver une implémentation satisfaisante. Mais auparavant, en plus de générer le code, il faut également l'évaluer, s'assurer de sa correction et de la stabilité des résultats. C'est l'objet de la partie suivante.

A.3 Plateforme expérimentale

L'objet de cette section est de décrire les précautions prises pour assurer, premièrement, que le code que nous produisons est correct, deuxièmement, qu'il est mesuré de manière pertinente, en éliminant correctement les différents bruits.

Pour cela, plusieurs mesures ont été prises. En premier lieu, la bibliothèque d'évaluation a été conçue de sorte à autoriser de manière flexible de pouvoir tester différents types de bench et différentes implémentations, d'en vérifier la correction avec plusieurs stratégies, en assurant de bien comparer des choses

```
for (k = 0; k < K; k += 2){
  for (j = 0; j < J; j += 1){
    for (i = 0; i < I; i += 1){
      scal_1 = C[J * i + j];
      scal_3 = A[K * i + k];
      scal_4 = B[J * k + j];
      scal_2 = scal_3 * scal_4;
      scal_0 = scal_1 + scal_2;
      C[J * i + j] = scal_0;

      scal_7 = A[K * i + k + 1];
      scal_8 = B[J * (k + 1) + j];
      scal_6 = scal_7 * scal_8;
      scal_5 = scal_1 + scal_6;
      C[J * i + j] = scal_5;
    }
  }
}
```

Figure A.5: $[\mathsf{R}_k, \mathsf{R}_j, \mathsf{R}_i, \mathsf{U}_{2,k}]$

comparables (par exemple de ne pas tester une sortie de multiplication de matrice avec un test de convolution). La bibliothèque permet de tester plusieurs compteurs hardware et d'en récupérer les résultats.

Pour assurer autant que possible la reproducibilité, la fréquence est fixée, on s'assure d'empêcher les migrations de processus. Chaque exécution peut être répétée de multiple fois, et différentes stratégies d'évaluation sont utilisées, entre utiliser la meilleure performance du lot, ou la médiane.

Ainsi armé, nous pouvons maintenant passer au coeur de cette thèse : comment faire pour trouver automatiquement une bonne solution au problème de l'optimisation de calculs de tenseur sur CPU.

A.4 Recherche de solutions optimales

La partie génération de code nous a permis de décrire l'espace des possibles. Il s'agit maintenant d'établir la meilleure stratégie pour trouver dans cet espace potentiellement très grand une implémentation qui utilise au mieux la machine.

Rappelons que notre objectif est de parvenir à un compromis entre une tactique d'autotuning à la Ansor et une approche experte à la GotoBlAS. L'essentiel à retenir est que notre solution n'est pas "one-shot" comme peut l'être GotoBLAS, elle nécessite d'exécuter un certain nombre de candidats pour trouver le meilleur. Par contre, nous avons incorporé certains apports de GotoBLAS afin d'éviter d'avoir à redécouvrir tous les paramètres importants. Ces améliorations nous permettent de converger sensiblement plus vite que ne le fait Ansor. Comme nous le verrons, l'élément déterminant a été de forcer l'utilisation d'un micronoyau performant. Une de nos idées principales a aussi été de profiter du nombre important de micronoyaux possibles pour adapter le choix du mi-

cronoyau à la taille du problème. Pour la partie supérieure du nid de boucle et le tuilage, nous avons tenté de mettre à profit des outils analytique de prédiction de cache. Cependant, comme on le verra, la sélection au hasard s'est révélé la meilleure solution.

Nous avons commencé par une observation : sur les petites tailles de multiplication de matrices, la divisibilité de la taille du micro-noyau vis-à-vis de la taille du problème acquiert une grande importance. En effet, les effets de la tuile partielle - la partie du calcul non couverte par le micro-noyau optimisé, et donc implémentée de manière sous-optimale - deviennent non négligeables. Cela est mis en évidence par une expérience sur des outils concurrents utilisant un micronoyau simple et faisant apparaitre une périodicité.

En parallèle, notre outil de génération de code nous permet de tester un nombre important de formes de micronoyaux. La conclusion est la suivante : sur les architectures récentes, le nombre de micronoyaux atteignant des performances proches de l'optimal (défini comme supérieur à 85% de la performance crête) se compte en dizaines. En conséquence, il est presque toujours possible de choisir une forme de micro-noyau adaptée à la taille du problème. Ce sera l'une des hypothèses de génération que nous retiendrons : éviter l'usage de tuile partielle pour ne pas subir la pénalité de performance en augmentant l'espace des micronoyaux possibles, tout en gardant l'idée d'utiliser un micronoyau performant. Quand, pour des raisons de tailles de problèmes difficilement divisibles, il n'est pas possible de trouver un micronoyau adapté, notre solution a été de combiner deux micro-noyaux pour obtenir un pavage complet. De la sorte, on parvient systématiquement à implémenter une solution sans tuile partielle en utilisant uniquement des micro-noyaux sélectionnés pour leur performance.

Ce choix de se restreindre à des micro-noyaux performants permet de garantir que la solution utilisée fait bon usage des caractéristiques du cœur (pipeline, unités vectorielles, parallélisme d'instruction). Reste à gérer la question de l'utilisation du cache.

Plusieurs pistes ont été explorées, qui s'appuyaient notamment sur des outils de modélisation de cache développés dans l'équipe. La solution finalement choisie est surprenante : comme la configuration autorisait un degré d'autotuning (au sens où on s'autorise à évaluer plusieurs solutions empiriquement avant de choisir la meilleure), la meilleure stratégie de choix s'est avérée être... le hasard.

En effet, il s'est avéré qu'une fois fixé la contrainte de l'utilisation des micronoyaux, l'espace est suffisamment dense en bonne solution pour qu'une recherche aléatoire converge de manière très rapide, au point d'en être difficile à concurrencer.

Différentes expériences mettent en évidence cette convergence très rapide. Par exemple, l'évaluation de l'espace des candidats pour plusieurs convolutions (par sélection aléatoire de 1000 candidats), permet de visualiser à la fois cet espace (par un histogramme comme en figure A.6) et d'évaluer la vitesse de convergence d'une recherche aléatoire (avec un niveau de certitude donné), comme on peut le voir en figure A.7.

Pour résumer le tout, la solution choisie finalement a été la suivante :

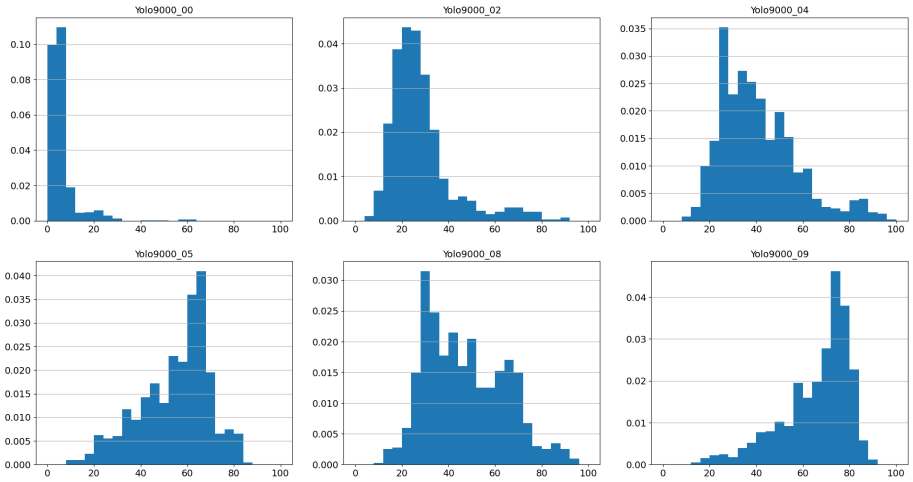


Figure A.6: Distribution of candidates for Yolo9000 convolutional layers from 00 to 09

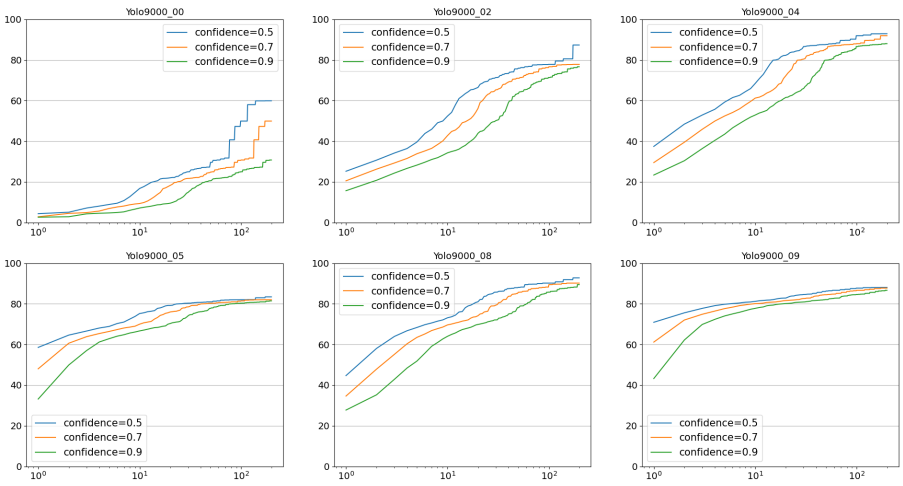


Figure A.7: expect(n, τ) for Yolo9000 convolutions from 00 to 09 on Xeon-Gold6130 (sequential) with $\tau = 0.5, 0.7, 0.9$ in **log scale**

Figure A.8: Performance sur Intel XeonGold5220

- Sélectionner des micro-noyaux performants par évaluation empirique (une seule fois par architecture)
- Une fois connue la taille du problème, sélectionner les micro-noyaux ou combinaison de micro-noyaux dont les tailles divisent parfaitement la convolution optimisée
- Pour chacun de ces micro-noyaux et combinaison de micro-noyaux, sélectionner aléatoirement un certain nombre de tuilages (par exemple 100)
- Faire tourner ces 100 candidats, mesurer leur temps d'exécution et garder le meilleur.

A.5 Resultats

Nous parvenons en séquentiel à égaler voire à battre nos concurrents sur des architectures récentes. On peut le voir sur la figure A.8, où nous nous comparons à Ansor, ainsi qu'à OneDNN un outil sans autotuning produit par Intel) et MOpt (un outil de recherche basé sur la modélisation de cache).

Nous disposons également de quelques résultats en contexte multicœur trouvable en figure A.9. La convergence y est plus difficile et les résultats moins satisfaisants, mais représentent une base de travail intéressante, surtout pour comprendre pourquoi on ne parvient pas à atteindre des résultats équivalents au séquentiel.

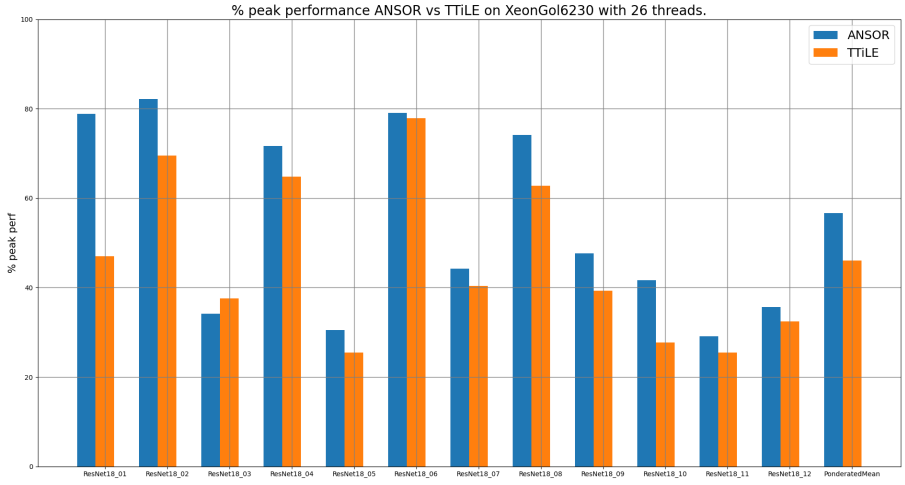


Figure A.9: Execution parallèle de ResNet18 sur XeonGold6230 (26 threads) avec Ansor et TTiLe

A.6 Conclusion

La thèse a permis de développer les outils suivants :

- Un outil de génération de code simple et modulaire permettant de tester facilement une grande diversité d'implémentation
- Une plateforme de test destinée à contrôler l'environnement d'exécution, ainsi qu'à tester la correction du code et certain paramètres comme l'impact du choix du compilateur
- Une méthodologie d'optimisation des calculs de tenseurs qui produit en séquentiel des résultats compétitifs vis-à-vis de l'état de l'art.

Cette méthodologie repose en premier lieu sur le choix de s'interdire l'usage des tuiles partielles et de se reposer uniquement sur l'utilisation de micro-noyaux performants. La méthode de choix notamment du tuilage a cependant donné des résultats surprenants et inattendus. Plusieurs leçons intéressantes ont été tiré. Ainsi, au niveau du coeur :

- L'espace des micro-noyaux peut être étendu sur les architectures récentes
- Cette extension, accompagné de la combinaison de micronoyaux offre une nouvelle solution au problème des tuiles partielles.

Et, au niveau de la hierarchie mémoire :

- La corrélation entre les modèles de cache et la performance n'a pas pu être bien établie

- Dès lors que les micro-noyaux étaient bien choisis, une recherche aléatoire permet, avec peu d'essais, de trouver avec une grande certitude des candidats très performants.

Du travail reste à faire quant à l'optimisation du cas parallèle. La question du packing reste aussi en suspens : nous n'avons pas pu mettre en évidence des gains de performances sur des architectures récentes pour le moment. Des travaux futurs permettront peut-être de comprendre mieux dans quel cas l'usage de cette technique est intéressant.

Bibliography

[ABC+16] 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.

[Bas04] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *13th International Conference on Parallel Architectures and Compilation Techniques (PACT 2004), 29 September - 3 October 2004, Antibes Juan-les-Pins, France*, pages 7–16. IEEE Computer Society, 2004.

[BCAC+13] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[BHRS08] 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.

[Bon20] Uday Bondhugula. High performance code generation in MLIR: an early case study with GEMM. *CoRR*, abs/2003.00532, 2020.

[BPP+17] 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.

[BRR+19] 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.

[CM95] Stephanie Coleman and Kathryn S McKinley. Tile size selection using cache organization and data layout. *ACM SIGPLAN Notices*, 30(6):279–290, 1995.

[CMJ+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018.

[CPS06] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. 10 2006.

[dMRVP09] Frédéric de Mesmay, Arpad Rimmel, Yevgen Voronenko, and Markus Püschel. Bandit-based optimization on graphs with application to library performance tuning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 729–736, New York, NY, USA, 2009. Association for Computing Machinery.

[DZ03] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 245–257. ACM, 2003.

[ERR+18] 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.

[GG08] Kazushige Goto and Robert A. Van De Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, 2008.

[GGL12] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letter*, 22(4), 2012.

[GVDG08] Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 blas. *ACM Transactions on Mathematical Software*, 35(1), 2008.

[HHHP16] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. LIBXSMM: Accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16. IEEE Press, 2016.

[Int18] Intel. oneAPI deep neural network library (oneDNN). `https://01.org/`, 2018.

[JL16] Akanksha Jain and Calvin Lin. Back to the future: Leveraging belady's algorithm for improved cache replacement. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pages 78–89. IEEE Computer Society, 2016.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[LAB+21] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[LDF+13] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 4.01 documentation and user's manual, 2013.

[LISQO16] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. Analytical modeling is enough for high-performance blis. *ACM Trans. Math. Softw.*, 43(2), aug 2016.

[Lou88] Margreet Louter-Nool. Algorithm 663: Translation of algorithm 539: basic linear algebra subprograms for FORTRAN usage in FORTRAN 200 for the cyber 205. *ACM Trans. Math. Softw.*, 14(2):177–195, 1988.

[LSV+19] 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 Perfor-*

mance Computing, Networking, Storage and Analysis (SC). ACM, 2019.

[LXSR+21] 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.

[MBDH99] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[MLdPM19] Richard Meyes, Melanie Lu, Constantin Waubert de Puiseau, and Tobias Meisen. Ablation studies in artificial neural networks, 2019.

[MVB15] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, pages 429–443. ACM, 2015.

[NVI18] NVIDIA. Cublas: Dense linear algebra on gpus, 2018.

[OIT+21] 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.

[RBA+13] 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.

[RT99] 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.

[SSB17] Paul Springer, Tong Su, and Paolo Bientinesi. HPTT: a high-performance tensor transposition C++ library. In Martin Elsman, Clemens Grelck, Andreas Klöckner, David A. Padua, and Edgar Solomonik, editors, *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ARRAY@PLDI 2017, Barcelona, Spain, June 18, 2017*, pages 56–62. ACM, 2017.

[SSF+12] Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan, and Vivek Sarkar. Analytical bounds for optimal tile size selection. In *Proceedings of Compiler Construction - 21st International Conference (CC)*, volume 7210 of *Lecture Notes in Computer Science*, pages 101–121. Springer, 2012.

[STHW15] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, page 207–216, New York, NY, USA, 2015. Association for Computing Machinery.

[VCJC+13] 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.

[VPG+18] Brice Videau, Kevin Pouget, Luigi Genovese, Thierry Deutsch, Dimitri Komatitsch, Frédéric Desprez, and Jean-François Méhaut. BOAST: A metaprogramming framework to produce portable and efficient computing kernels for HPC applications. *Int. J. High Perform. Comput. Appl.*, 32(1):28–44, 2018.

[VZT+19] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically. *ACM Trans. Archit. Code Optim.*, 16(4), oct 2019.

[VZvdG15] 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):14:1–14:33, June 2015.

[WZS+14] 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.

[ZFL18a] Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. High performance zero-memory overhead direct convolutions. *CoRR*, abs/1809.10170, 2018.

[ZFL18b] Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. High performance zero-memory overhead direct convolutions. volume 80 of *Proceedings of Machine Learning Research*, pages 5776–5785. PMLR, 2018.

[ZJS+20] 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.