Algorithm XXX: Automatic Generators for a Family of Matrix Multiplication Routines with Apache TVM

GUILLERMO ALAEJOS, Universitat Politècnica de València, Spain
ADRIÁN CASTELLÓ, Universitat Politècnica de València, Spain
PEDRO ALONSO-JORDÁ, Universitat Politècnica de València, Spain
FRANCISCO D. IGUAL, Universidad Complutense de Madrid, Spain
HÉCTOR MARTÍNEZ, Universidad de Córdoba, Spain
ENRIQUE S. QUINTANA-ORTÍ, Universitat Politècnica de València, Spain

We explore the utilization of the Apache TVM open source framework to automatically generate a family of algorithms that follow the approach taken by popular linear algebra libraries, such as GotoBLAS2, BLIS and OpenBLAS, in order to obtain high-performance blocked formulations of the general matrix multiplication (GEMM). In addition, we fully automatize the generation process, by also leveraging the Apache TVM framework to derive a complete variety of the processor-specific micro-kernels for GEMM. This is in contrast with the convention in high performance libraries, which hand-encode a single micro-kernel per architecture using Assembly code. In global, the combination of our TVM-generated blocked algorithms and micro-kernels for GEMM 1) improves portability, maintainability and, globally, streamlines the software life cycle; 2) provides high flexibility to easily tailor and optimize the solution to different data types, processor architectures, and matrix operand shapes, yielding performance on a par (or even superior for specific matrix shapes) with that of hand-tuned libraries; and 3) features a small memory footprint.

Additional Key Words and Phrases: Portability and maintainability, software lifecycle, matrix multiplication, BLIS framework, Apache TVM, blocking, SIMD vectorization, high performance

ACM Reference Format:
Guillermo Alaejos, Adrián Castelló, Pedro Alonso-Jordá, Francisco D. Igual, Héctor Martínez, and Enrique S. Quintana-Ortí. —. Algorithm XXX: Automatic Generators for a Family of Matrix Multiplication Routines with Apache TVM. —, —, Article — (—), 34 pages. https://doi.org/---

1 INTRODUCTION

Over the past decades, there has been a persistent labor toward developing high performance realizations of linear algebra (LA) libraries for a variety of architectures, such as vector processors, multicore processors with SIMD (single-instruction, multiple-data) units, data-parallel graphics processing units (GPUs) and, more recently, domain-specific architectures and accelerators. This effort has come from major hardware vendors, with some relevant products being Intel MKL, AMD AOCL, IBM ESSL, ARMPL and NVIDIA cuBLAS, as well as from the academic side, with software packages such as GotoBLAS2 [15], OpenBLAS [33], BLIS [30] and ATLAS [13].

The general matrix multiplication (GEMM) is a crucial computational kernel upon which these LA libraries are built. In addition, GEMM is also a key operation for deep learning applications that leverage transformers for natural language processing or convolutional deep neural networks

Authors' addresses: Guillermo Alaejos, Universitat Politècnica de València, Valencia, 46022, Spain, galalop@upv.es; Adrián Castelló, Universitat Politècnica de València, Valencia, 46022, Spain, adcastel@disca.upv.es; Pedro Alonso-Jordá, Universitat Politècnica de València, Valencia, 46022, Spain, palonso@upv.es; Francisco D. Igual, Universidad Complutense de Madrid, Madrid, 28040, Spain, figual@ucm.es; Héctor Martínez, Universidad de Córdoba, Córdoba, 14071, Spain, el2mapeh@uco.es; Enrique S. Quintana-Ortí, Universitat Politècnica de València, Valencia, 46022, Spain, quintana@disca.upv.es.

for signal processing and computer vision [2, 29]. Unfortunately, these LA libraries present a few obstacles:

1. The optimized routines in these libraries are hardware-specific. This is the case for Intel, IBM, ARM and NVIDIA's products. To a lesser extent, it also applies to GotoBLAS2, OpenBLAS and BLIS,[1] which leverage a collection of processor-customized micro-kernels [31].
2. Developing highly optimized micro-kernels for GEMM requires a deep knowledge of high performance computing and computer architecture.
3. The code in these libraries is very large. Besides, it is hard to master due to the abundant use of productivity-enhancing macros, templates and high level programming techniques. Maintaining the libraries thus mostly lies in the hands of the original developers.
4. The software misses some relevant cases such as, for example, support for half (i.e., 16-bit) floating point precision or integer arithmetic.
5. The implementation of GEMM in these libraries is sub-optimal under certain circumstances as the code is usually tuned for "squarish", large-scale cases.
6. The memory footprint of the libraries is often in the order of Mbytes.

In this paper we address the limitations of LA libraries by demonstrating that it is possible to automatically generate a family of blocked algorithms for GEMM, together with a collection of micro-kernels for GEMM, using Apache TVM (Tensor Virtual Machine) [8]. This alternative solution offers the following advantages:

1. At a high level, the library is "replaced" by a collection of TVM generators (in the form of Python routines), reducing the maintainability effort to a bare minimum and largely improving the portability of the solution.
2. Using the appropriate backend compiler, the generation/optimization can be easily specialized for distinct data types, further enhancing the portability and maintainability of the solution.
3. By adjusting the algorithm and micro-kernel to the problem, it is possible to outperform high performance realizations of GEMM in commercial products as well as academic libraries.
4. The optimization process for each problem dimension is largely seamless, boiling down to the evaluation of a reduced number of micro-kernels. In other words, the optimization search space is limited.
5. The memory footprint of the resulting realization of GEMM is very small and the entire framework library is also very small.

Our paper makes two significant contributions. Firstly, it offers a comprehensive and pedagogical tutorial on building a high-performance implementation of GEMM using TVM, complete with extensive code examples and detailed explanations. Secondly, and perhaps more importantly, it provides compelling evidence for the advantages of automating the development process. These advantages include enhanced portability, reduced maintenance costs, insulation from hardware-specific optimization details, and ultimately, improved performance. This performance boost arises from the ability to explore multiple micro-kernels, which outperforms libraries relying on a single general-purpose micro-kernel.

The rest of the paper is structured as follows. Section 2 provides an in-depth review of the state-of-the-art in automatic realizations of high performance libraries in the fields of dense linear algebra in general, and deep learning in particular, and discusses different alternatives towards automatic code generation. In Section 3 and 4 we respectively review the modern realization of algorithms and micro-kernels for GEMM. This is then followed, in Sections 5 and 6, by the introduction of the automatic generation of a baseline GEMM algorithm and a variety of micro-kernels, respectively;

[1]The same comment applies to AMD's library, which is simply a customized version of BLIS.

and in Section 7, by the extension of the automatization techniques to cover a complete family of GEMM algorithms. In Section 8 we evaluate the performance of the resulting solution in a platform equipped with ARM cores, and demonstrate its portability to other modern architectures from different vendors. Finally, in Section 9 we summarize the main results from this work.

2 RELATED WORK

Automatic code generation is gaining interest in the last years as a means to attain performance portability across existing and new architectures, for deep learning (DL) models, with a minimal intervention from the programmer [20]. Recent languages and compiler frameworks, such as Halide [24] or TVM [8], propose a clear separation of concerns between the definition of the operation and its optimization, in order to ease development of operators to a plethora of target architectures, including general-purpose processors, GPUs, digital signal processors (DSPs), and specific-purpose accelerators [23]. Starting from a *computational graph* which defines the operator and the *data flow*, optimization techniques for performance portability are applied at the graph level, via operator fusions and transformations; and at the operator level, with hardware-specific optimizations. Some of these optimizations are framework-specific (e.g., TensorFlow XLA [26]) while others, developed by experts, are realized within specialized libraries (e.g., NVIDIA cuDNN[12]).

From a technical perspective, DL compilers can be broadly classified as JIT (*Just-in-time*) or AOT (*Ahead-of-time*). JIT compilers generate executable code on-the-fly. Hence, they can exploit extended runtime knowledge to fine-tune the final executable, at the cost of a certain overhead due to the code generation logic. Many common DL frameworks and compilers rely or support JIT, including TFLite and its Micro variant[2], XLA, MLIR [19] and TVM. Contrarily, AOT compilers generate executable code a priori, and execute common versions at runtime without further modification. The advantage of the AOT approach is two-fold: first, it can extend the analysis scope, hence accommodating more thorough optimizations; second, it eases the development of cross-compilation schemes for embedded architectures [18] or remote execution-only architectures [10]. The use of external libraries for DL primitives (mainly convolutions and linear algebra primitives) can be also considered as AOT, since in general they are implemented statically and optimized prior to execution time.

Armed with hardware-agnostic IRs (Intermediate Representations), these compiler frameworks effectively decouple schedule from computation, and enable the automatic exploration of the scheduling and configuration spaces by means of auto-tuning techniques. A clear example is AutoTVM [11], integrated within TVM, which performs a complete exploration of the search space; at each iteration, the framework tests the generated code on the target device, and stores some feedback which is eventually leveraged to find the optimal combination of configuration parameters. This *brute force* optimization scheme guarantees finding the optimal solution (configuration setup), but the number of tested configurations grows exponentially with the dimension of the design space. Hence, the use of these naive schemes is limited to problems with reduced search spaces, or where online testing is time-inexpensive. (Unfortunately, this is not the case of the architecture-aware adaptation of DL models to a specific hardware setup.) In order to alleviate the expensive search-and-test procedure, automatic learning schemes (such as XGBoost [7], within TVM) have been enhanced [35] with Random Search [3], Bayesian optimization [17] and Genetic algorithms [32]. More recently, Deep Reinforcement Learning techniques have been proposed to autonomously guide and extract policies for optimal execution configuration [1, 22, 25, 37]. While all these efforts alleviate the cost of the hyperparameter search, they are still computationally expensive and, in many cases, yield solutions that are difficult to explain or reproduce for developers, or to port to embedded architectures or systems with reduced compute capabilities.

[2]http://www.tensorflow.org/lite.

Auto-tuning techniques within DL compiler frameworks follow similar ideas to those of auto-tuning in dense LA routines. ATLAS [13] selects the cache configuration parameters and the micro-kernel at installation time via a comprehensive search procedure. Recently, in the framework of the BLIS project, the work in [21] demonstrated that deriving analytical models for optimal configuration parameters selection is an effective way to attain high performance without the necessity of an auto-tuning scheme for configuration parameter exploration. Replacing the auto-tuning scheme with model-based solutions was also successfully explored in [33, 34].

Our approach combines the advantages of existing JIT compiler frameworks for easily deriving high performance codes for GEMM-based DL primitives, with analytical models in order to avoid expensive auto-tuning. Our work differs from [4], which uses MLIR to describe early experiences exclusively with GEMM, as well as from [36], which proposes advanced auto-tuning schemes for this primitive. Concretely, we leverage TVM to extend and further analyze the automatic generation of GEMM-based primitives for DL, integrating analytical models to ease the optimization process without the need of expensive auto-tuning procedures. In addition, we apply our ideas to GEMM (Section 5 and 6).

The use of analytical models to determine optimal execution parameters eliminates the time penalty introduced by auto-tuning, and avoids the lack of accuracy and reproducibility. While not discussed in this work, 1) extending the analytical models to address other algorithmic variants for GEMM, which favor certain cache hierarchy setups [5], as additional configuration parameters; and 2) supporting optimal parallelization schemes [27] for GEMM-based operators, and supporting this additional degrees of freedom within the compilation stage are also clear benefits of our proposal. In comparison, these extensions would introduce a non-affordable complexity to auto-tuning schemes.

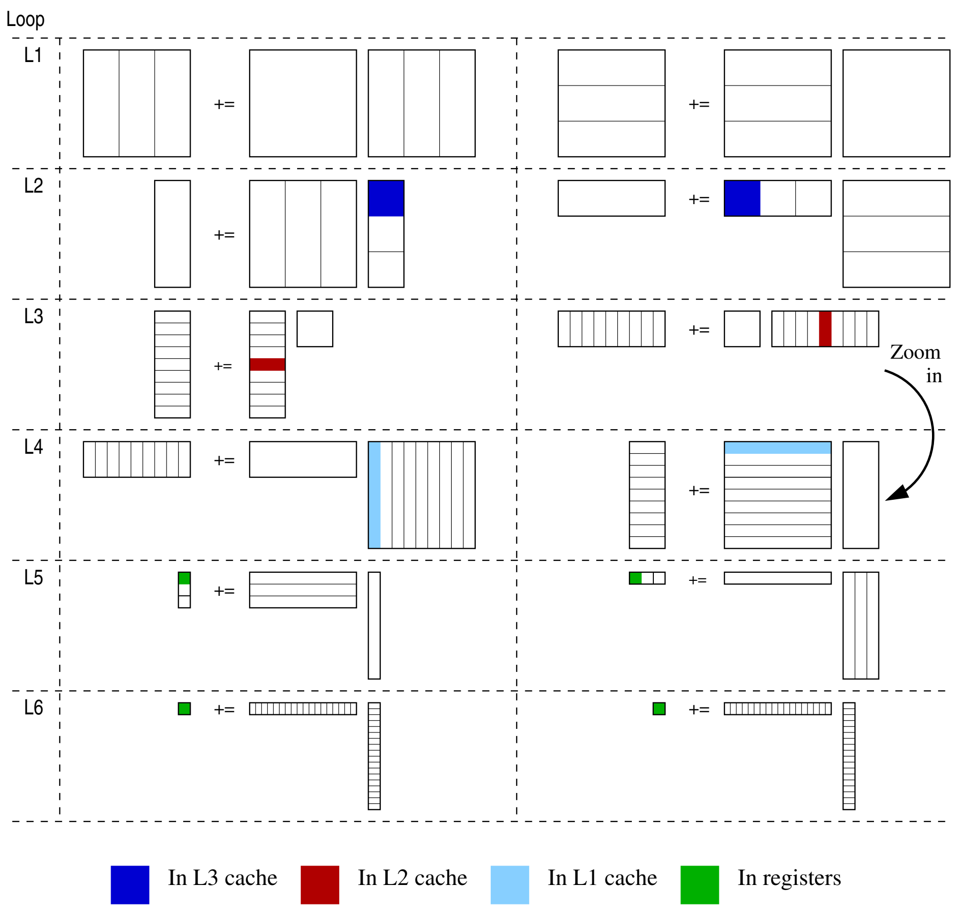
3 A FAMILY OF BLOCKED ALGORITHMS FOR GEMM

In this section we review the modern realization of GEMM in current high performance libraries, and generalize the ideas to discuss a full family of algorithms for this operation.

3.1 The baseline algorithm

Consider the matrix multiplication $C = C + AB$, abbreviated as $C \mathrel{+}= AB$, where the matrix operands present the following dimensions: $A \rightarrow m \times k$, $B \rightarrow k \times n$, and $C \rightarrow m \times n$. The realization of this kernel for general-purpose processor architectures with hierarchically-layered memories, in libraries such as OpenBLAS, BLIS, AMD AOML and, possibly, Intel MKL/oneMKL, follow the basic ideas of GotoBLAS2 [15] to decompose the computation into five nested loops, traversing the m, n, k dimensions of the problem in a specific order. Inside these loops, two packing routines copy (and re-arrange) certain blocks of the input operands A, B into two buffers, $A_c \rightarrow m_c \times k_c$, $B_c \rightarrow k_c \times n_c$, in order to favor an efficient utilization of the cache memories. (For simplicity, in the following we assume that m, n, and k are integer multiples of m_c, n_c, and k_c, respectively.) Furthermore, the fifth loop comprises a micro-kernel that is often vectorized to exploit the SIMD units in most current general-purpose processors [30]. In short detail, the *baseline algorithm for* GEMM aims at maintaining a block B_c in the L3 cache and a block A_c in the L2 cache, while streaming C directly from the main memory into the processor registers. Furthermore, a small micro-panel of B_c is to reside in the L1 cache.

The orchestration of the data movements across the memory hierarchy in the baseline algorithm for GEMM is endowed by the specific nesting of the algorithm loops, in combination with a careful choice of the loop strides/sizes of the buffers [21]. The operands' partitioning induced by the loops as well as the target level of the memory hierarchy for each operand block are illustrated in Figure 1 (left). We will also refer to the baseline algorithm as B3A2C0, where each letter denotes one of the three matrix operands, and the subsequent number specifies the cache level where a part of the



```
for (jc=0; jc<n; jc+=nc) // Loop L1
 for (pc=0; pc<k; pc+=kc) {   // L2
  // Pack B
  Bc := B(pc:pc+kc-1,jc:jc+nc-1);
  for (ic=0; ic<m; ic+=mc) {  // L3
   // Pack A
   Ac := A(ic:ic+mc-1,pc:pc+kc-1);
   for (jr=0; jr<nc; jr+=nr)  // L4
    for (ir=0; ir<mc; ir+=mr) // L5
     // Micro-kernel
     C(ic+ir:ic+ir+mr-1,
       jc+jr:jc+jr+nr-1)
       += Ac(ir:ir+mr-1,0:kc-1)
       *  Bc(0:kc-1,jr:jr+nr-1);
 }}
```

```
for (ic=0; ic<m; ic+=mc) // Loop L1
 for (pc=0; pc<k; pc+=kc) {   // L2
  // Pack A
  Ac := A(ic:ic+mc-1,pc:pc+kc-1);
  for (jc=0; jc<n; jc+=nc) {  // L3
   // Pack B
   Bc := B(pc:pc+kc-1,jc:jc+nc-1);
   for (ir=0; ir<mc; ir+=mr)  // L4
    for (jr=0; jr<nc; jr+=nr) // L5
     // Micro-kernel
     C(ic+ir:ic+ir+mr-1,
       jc+jr:jc+jr+nr-1)
       += Ac(ir:ir+mr-1,0:kc-1)
       *  Bc(0:kc-1,jr:jr+nr-1);
 }}
```

Fig. 1. Baseline (B3A2C0) and A3B2C0 algorithms (left and right, respectively) for GEMM.

operand is to reside, with 0 referring to the processor registers. As *B* is to reside both in the L1 and L3 caches, we do not specify the former in the notation.

3.2 Other members of the GEMM family

We continue the discussion on the high performance realization of GEMM by noting that there exist five other algorithmic variants which can be obtained by re-organizing the loops of the baseline algorithm in a different manner [5, 16, 28]. For example, a "twin" version is directly obtained by swapping the roles of A and B in the baseline algorithm, yielding the A3B2C0 variant, where two blocks of A respectively occupy the L1 and L3 caches and a block of B resides in the L2 cache; see Figure 1 (right). In the same line, Figure 2 displays two additional variants of the GEMM family: B3C2A0 (left) and A3C2B0 (right) that maintain a block of C in the L2 cache. The two missing variants, where C resides in the L3 cache, (C3B2A0 and C3A2B0, omitted for brevity,) are derived by swapping the roles of A/B with C in the two algorithms given in that figure; see [5].

4 HIGH PERFORMANCE MICRO-KERNELS FOR GEMM

In this section, we connect the six blocked algorithms for GEMM with three types of micro-kernels that differ in the matrix operand that resides in the processor registers [5, 16, 28]. For simplicity, we assume that the cache configuration parameters m_c, n_c, k_c are respectively integer multiples of the micro-kernel parameters m_r, n_r, k_r where, depending on the type of micro-kernel, two of the latter parameters specify the dimension of the micro-tile that resides in the processor registers.

4.1 Operand Resident C (in the processor registers)

The baseline algorithm for GEMM and its "twin" A3B2C0 (both in Figure 1) cast the innermost computation in terms of a micro-kernel that computes a smaller GEMM, $C_r \mathrel{+}= A_r B_r$, where $A_r \rightarrow m_r \times k_c$, $B_r \rightarrow k_c \times n_r$ respectively denote two micro-panels of the buffers A_c, B_c; while $C_r \rightarrow m_r \times n_r$ is a small micro-tile of C that resides in the processor registers during the execution of the micro-kernel. This corresponds to the operation performed inside loop `L5` of the baseline algorithm B3A2C0 (see Figure 1), with

A_r = `Ac(ir:ir+mr-1,0:kc-1)`,
B_r = `Bc(0:kc-1,jr:jr+nr-1)`, and
C_r = `C(ic+ir:ic+ir+mr-1,jc+jr:jc+jr+nr-1)`.

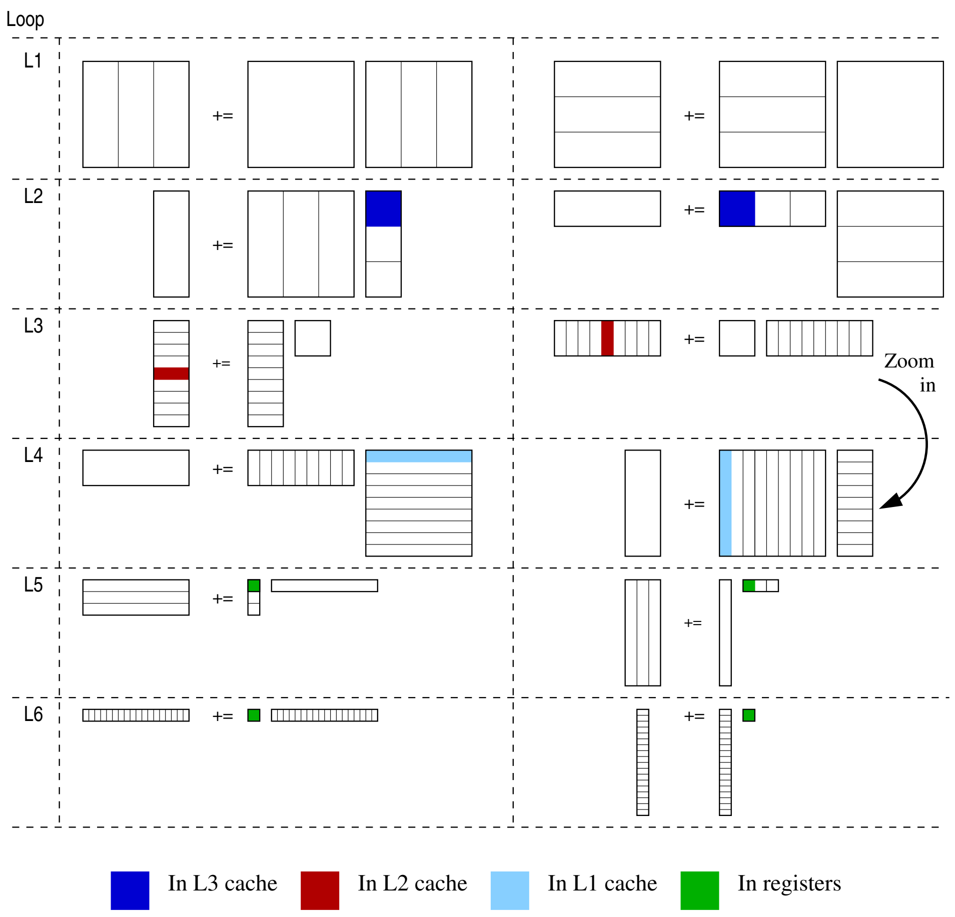
The realization of this micro-kernel iterates across the k_c dimension of the problem (as part of an additional loop, labeled as `L6`), at each step performing an outer product involving a single column of A_r and a single row of B_r to update the entire micro-tile C_r; see Figure 3 (top).

For high performance, the data in A_c, B_c are carefully packed as illustrated in Figure 4, in order to ensure access with unit stride to the columns of A_r and the rows of B_r from within the micro-kernel. This reduces the number of cache evictions during these accesses as well as accommodates the use of efficient SIMD instructions to load these elements into vector registers and operate with them.

4.2 Other types of micro-kernels: Operand Resident A or Resident B

The four remaining variants of GEMM (see, e.g., Figure 2) leverage two other types of micro-kernels: The first one maintains a micro-tile of A in the processor registers while performing an $m_r \times k_r$ matrix-vector product per iteration of loop `L6`. The second one keeps a micro-tile of Resident B in the processor registers, and carries out an $k_r \times n_r$ vector-matrix product per iteration of loop `L6`. These two types of micro-kernels are illustrated in Figure 3 (middle and bottom).

To enable SIMD loads/stores, each type of micro-kernel requires a specialized packing scheme for two of the matrix operands. For the micro-kernel with Resident A (in the processor registers), C_c and B_c are packed following the same pattern as A_c in Figure 4 (with the entries of B_c arranged into micro-panels of k_r rows). For the micro-kernel with Resident B, the buffers for C_c and A_c are packed as B_c in the same figure (with the entries of A_c arranged into micro-panels of k_r columns).

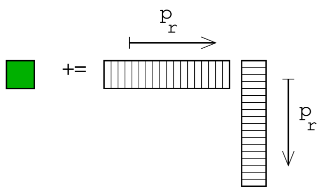


```
for (jc=0; jc<n; jc+=nc) // Loop L1
 for (pc=0; pc<k; pc+=kc) {   // L2
  // Pack B
  Bc := B(pc:pc+kc-1,jc:jc+nc-1);
  for (ic=0; ic<m; ic+=mc) {  // L3
   // Pack C
   Cc := C(ic:ic+mc-1,jc:jc+nc-1);
   for (pr=0; pr<kc; pr+=kr)  // L4
    for (ir=0; ir<mc; ir+=mr) // L5
     // Micro-kernel
     Cc(ir:ir+mr-1,0:nc-1)
        += A(ic+ir:ic+ir+mr-1,
             pc+pr:pc+pr+kr-1)
        *  Bc(pr:pr+kr-1,0:nc-1);
   // Unpack C
   C(ic:ic+mc-1,jc:jc+nc-1) := Cc;
 }}
```

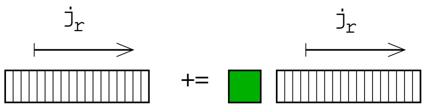
```
for (ic=0; ic<m; ic+=mc) // Loop L1
 for (pc=0; pc<k; pc+=kc) {   // L2
  // Pack A
  Ac := A(ic:ic+mc-1,pc:pc+kc-1);
  for (jc=0; jc<n; jc+=nc) {  // L3
   // Pack C
   Cc := C(ic:ic+mc-1,jc:jc+nc-1);
   for (pr=0; pr<kc; pr+=kr)  // L4
    for (jr=0; jr<nc; jr+=nr) // L5
     // Micro-kernel
     Cc(0:mr-1,jr:jr+nr-1)
        += Ac(0:mr-1,pr:pr+kr-1)
        *  B(pc+pr:pc+pr+kr-1,
             jc+jr:jc+jr+nr-1);
   // Unpack C
   C(ic:ic+mc-1,jc:jc+nc-1) := Cc;
 }}
```

Fig. 2. B3C2A0 and A3C2B0 algorithms (left and right, respectively) for GEMM.

```
for (pr=0; pr<kc; pr++) // Loop L6
  C(ic+ir:ic+ir+mr-1,
    jc+jr:jc+jr+nr-1)
    += Ac(ir:ir+mr-1,pr)
    *  Bc(pr,jr:jr+nr-1);
```



```
for (jr=0; jr<nc; jr++) // Loop L6
  Cc(ir:ir+mr-1,jr)
      += A(ic+ir:ic+ir+mr-1,
           pc+pr:pc+pr+kr-1)
      *  Bc(pr:pr+kr-1,jr);
```



```
for (ir=0; ir<mr; ir++) // Loop L6
  Cc(ir,jr:jr+nr-1)
      += Ac(ir,pr:pr+kr-1)
      *  B(pc+pr:pc+pr+kr-1,
           jc+jr:jc+jr+nr-1);
```

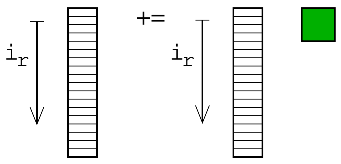


Fig. 3. Micro-kernels with Resident C, Resident A or Resident B in the processor registers (top, middle and bottom, respectively).

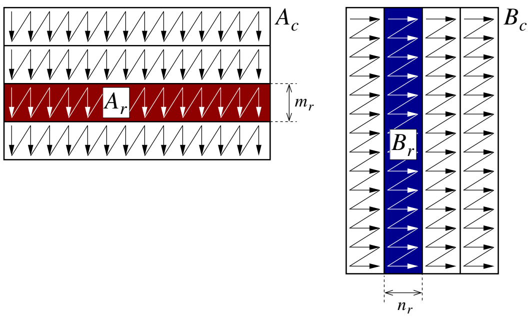


Fig. 4. Packing in the baseline algorithm.

4.3 High performance

A few rules of thumb guide the design of a high performance micro-kernel for a processor architecture with a multi-layered memory hierarchy [21]. We discuss them for a micro-kernel with Resident C, but they are easily to derive for the two other types of micro-kernel:

- Considering the k_c successive updates of the micro-tile C_r occurring in loop L6, the micro-kernel parameters m_r, n_r should be chosen sufficiently large so as to avoid stalls due the latency between the issuance of two instructions that update the same entry of C_r.
- Ideally, m_r should be equal to n_r as this maximizes the ratio of computation to data movement during the update of C_r in loop L6.

These two principles suggest maximizing the values for m_r, n_r as part of a "large" micro-kernel. In practice though, the limited number of vector registers constrains the practical values of m_r, n_r within a couple of dozens.

A comparison of the micro-kernel with Resident C and those with Resident A/B reveals some differences:

- The variants with Resident C presents a higher arithmetic intensity since, while all types of micro-kernels perform the same number of flops and number of loads from memory, the variants with Resident A/B have to write a column or row of C back into the memory at each iteration of the loop.
- Unlike the micro-kernel with Resident C, the variants with Resident A/B do not present a RAW (read-after-write) dependency between consecutive iterations of the micro-kernel.

The implementation of the micro-kernels in general-purpose processor architectures equipped with SIMD arithmetic units is in practice done in Assembly code; vectorized using architecture-specific SIMD instructions (e.g., Intel SSE/AVX, ARM NEON, etc.); and enhanced with high performance computing techniques such as loop unrolling, software pipelining, data prefetching, etc. [14].

We close this section by noting that for large, squarish GEMM problems, the optimal values for the micro-kernel parameters m_r, n_r, k_r can be determined, for a given processor architecture, via a few experimental tests. However, for problems with one dimension much larger than the others, the optimal values of these parameters may vary considerably, and they can only be determined experimentally by first implementing them. Unfortunately, developing a high performance micro-kernel is mostly a manual process, which requires an expert with a deep knowledge of high performance computing and computer architecture to attain optimal performance.

5 AUTOMATIC GENERATION OF THE BASELINE ALGORITHM FOR GEMM

Apache TVM is an open source compiler framework that allows generating, optimizing, and executing machine learning routines for (general-purpose) multicore processors, GPUs (graphics processing units), and other accelerator backends [9]. In our effort toward automatically generating high performance algorithms for GEMM, we start from the TVM tutorial on how to optimize GEMM on a general-purpose processor by blocking (tiling) and packing the matrix operands.[3] Concretely, we build upon these instructions to assemble a TVM generator that automatically produces the baseline algorithm for GEMM. Our approach extends the tutorial in that we apply the optimizations in a BLIS-like manner instead of following a general optimization approach. More concretely we introduce the cache-aware packings as well as software prefetching. In addition, we provide guidelines to derive different algorithmic variants for GEMM.

5.1 Basic GEMM with TVM

Consider a basic realization of GEMM consisting of three loops that compute each element of the output matrix by performing a reduction (i.e., an inner or dot product) across the k dimension of the problem:

```
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    for (p=0; p<k; p++)
      C(i,j) += A(i,p) * B(p,j);
```

This specific realization of GEMM, with the loops traversing the dimensions of the problem in the order $i \Rightarrow j \Rightarrow p$, can be obtained using the (Python-based) TVM generator in Figure 5. We

[3]https://tvm.apache.org/docs/tutorials/.

distinguish nine fundamental parts in the code (in this basic example, some of the parts are empty, but they will appear in the refined versions of the generator described in subsequent sections):

Part P0. Parameter list The function receives the matrix dimensions `m, n, k`, and the data type of the operands (`dtype`) as parameters.

Part P1. Declaration of the input operands: Lines 3 and 4 define two operands, or placeholders, respectively for A and B, of the appropriate dimensions and data type.

Part P2. Definition of the operation: Lines 7–10 specify the operation to be computed in terms of the two placeholders. In particular, line 7 defines the computation in terms of a reduction (`sum`) across the dimension `k`.

Part P3. Preparation of the schedule: Line 13 creates a *schedule*. In TVM, this corresponds to the order in which the loops within the program are executed and, in this particular case, to the three nested loops induced from the application of the lambda function for each (i, j) entry across the reduction axis p. By default, the schedule processes a tensor serially following a row-major order.

Part P4. Specification of the loop ordering: The order in which the loops are generated with the previous schedule may not match that of the baseline algorithm. Lines 16-18 extract the desired axes induced from the computation loops defined in Part P2, and ensure that the loops follow the scheme in the basic GEMM: $i \Rightarrow j \Rightarrow p$.

Part P5. Placement of the packings: No packing occurs in the basic GEMM and, therefore, this part is empty.

Part P6. Application of fine-grain optimizations: Fine-grain optimizations such as unrolling and SIMD vectorization are not included in this initial version.

Part P7. Loop-level parallelization: Selection of the specific loop to parallelize, if necessary.

Part P8. Generation of code: Finally, line 39 instructs TVM to generate the code, in this case, for an LLVM backend.

Other loop orderings are easily obtained using TVM by simply re-arranging differently the loop variables in line 18. (However, this is also straightforward to do in the basic triple nested loop code written in C.) More interestingly, this example also illustrates that code for distinct backends can be obtained by simply changing the `target` in line 39. Concretely, the generic `llvm` target there generates code for the machine in which the generator is executed, but the commented lines in Part P8 offer several examples which generate code for other architectures. To close the discussion of this first TVM generator, note that it is independent of the matrix operands' data types.

5.2 Blocking for the baseline algorithm with TVM

Our next goal is to build a blocked algorithm that partitions the matrix operands mimicking the three outermost loops of the baseline algorithm (B3A2C0), labeled as `L1`, `L2` and `L3`. For this purpose, remember that these loops divide A into blocks of dimension $m_c \times k_c$, B into blocks of dimension $k_c \times n_c$, and C into blocks of dimension $m_c \times n_c$; see Figure 1 (left). For clarity, consider the following blocked algorithm, which realizes the partitioning scheme in the baseline algorithm for GEMM in order to decompose the matrix multiplication into a collection of finer-grain computations:

```
for (jc=0; jc<n; jc+=nc)
  for (pc=0; pc<k; pc+=kc)
    for (ic=0; ic<m; ic+=mc)
      for (jr=0; jr<nc; jr++)
        for (ir=0; ir<mc; ir++)
          for (pr=0; pr<kc; pr++)
            C(ic+ir,jc+jr) += A(ic+ir,pc+pr) * B(pc+pc,jc+jr);
```

```python
def basic_GEMM(m, n, k, dtype):
  # P1) Declare the input operands
  A = te.placeholder((m, k), name="A", dtype=dtype)
  B = te.placeholder((k, n), name="B", dtype=dtype)

  # P2) Define the operation
  p = te.reduce_axis((0, k), "p")
  C = te.compute((m, n), lambda i, j:
                 te.sum(A[i, p] * B[p, j],
                 axis=p), name="C")

  # P3) Prepare the schedule
  sched  = te.create_schedule(C.op)

  # P4) Specify the loop ordering (schedule i->j->p)
  (i, j) = C.op.axis
  (p,)   = C.op.reduce_axis
  sched[C].reorder(i, j, p)

  # P5) Emplace the packings
  #     No packings in this version

  # P6) Apply fine-grain optimizations
  #     No optimizations in this version

  # P7) Loop-level parallelization
  #     No parallelization in this version

  # P8) Generate code with LLVM backend
  #     A few examples for specific targets:
  #      - ARMv8a (8.2) with NEON support
  #         llvm -device=arm_cpu -mattr=+v8.2a,+fp-armv8,+neon"
  #      - ARMv8a (8.2) with NEON and FP16 support
  #         llvm -device=arm_cpu -mattr=+v8.2a,+fp-armv8,+neon,+fp16fml
  #      - AMD Zen2 with AVX2 support
  #         llvm -mcpu=znver2
  #      - Intel Icelake with AVX512 support
  #         llvm -mcpu=icelake-server
  return tvm.build(sched, [A, B, C], target="llvm")
```

Fig. 5. TVM generator for the basic GEMM.

Figure 6 displays a TVM generator that produces a code for GEMM that will perform the computation adopting the same blocking scheme. Compared with the TVM generator for the basic GEMM in Figure 5, the routine presents the following differences:

Part P0. Parameter list: In addition to the parameters from the basic GEMM, the function receives the blocking parameters `mc, nc, kc`.

Part P3. Preparation of the schedule (with blocking): In preparation for the operands' tiling, lines 9–13 prompt the sought-after splittings of the problem dimensions m, n, k.

Part P4. Specification of the loop reordering: Line 16 specifies a loop ordering which matches that of the baseline algorithm for GEMM: $j_c \Rightarrow p_c \Rightarrow i_c \Rightarrow j_r \Rightarrow i_r \Rightarrow p_r$.

Here and in the following, for brevity, we will omit the parts that remain the same with respect to the prior generators.

The code for GEMM produced by TVM using the algorithm in Figure 6 differs from the high performance realizations of GEMM in libraries such as GotoBLAS, BLIS and OpenBLAS, in two important aspects:

1. There are no packing routines that re-arrange the contents of the input matrices A, B into buffers in order to streamline the operation of the micro-kernel.

```
def block_GEMM_B3A2C0(m, n, k, mc, nc, kc, dtype):
  # P1) Declare the input operands
  #     As in basic_GEMM. Omitted for brevity

  # P2) Define the operation with blocking
  #     As in basic_GEMM. Omitted for brevity

  # P3) Prepare the schedule with blocking
  sched = te.create_schedule(C.op)
  ic, jc, \
  ir, jr = sched[C].tile(C.op.axis[0],
                         C.op.axis[1], mc, nc)
  pc, pr = sched[C].split(p, factor=kc)

  # P4) Specify the loop ordering (schedule as in baseline/B3A2C0)
  sched[C].reorder(jc, pc, ic, jr, ir, pr)

  # P5) Emplace the packings
  #     No packing available in this version

  # P6) Apply fine-grain optimizations
  #     No optimizations applied

  # P7) Loop-level parallelization
  #     No parallelization in this version

  # P8) Generate code with LLVM backend
  return tvm.build(sched, [A, B, C], target="llvm")
```

Fig. 6. TVM generator for GEMM mimicking the blocking scheme of the baseline algorithm.

2. The innermost loop does not formulate the computation in terms of an outer product that updates a small $m_r \times n_r$ micro-tile C_r. Instead, the TVM generator produces a code which decomposes this last computation into fine-grain multiply-add operations involving individual elements of A, B, C.

In the next subsection we deal with the first issue while the second one is discussed in Section 6.

5.3 Packing for the baseline algorithm with TVM

Figure 7 refines the TVM generator for the blocked algorithm to include hints to TVM in order to produce a specialized version that packs A and B into two buffers, A_c and B_c, with the same layout as that present in the baseline algorithm; see Figure 4. We note the following differences with respect to the TVM generator for the blocked algorithm:

Part P0. Parameter list: The new routine receives two additional parameters: `mr, nr`.

Part P2. Definition of the operation and packing schemes for A and B: Lines 8–12 declare a 4D TVM tensor `Ac` which will hold a packed version of the complete operand A. `Ac` is instantiated as a (2D) matrix of small (packed) matrices (*micro-panels* of A), each of dimension $m_r \times k_c$; see the red micro-panel labeled as A_r in Figure 4. `Ac` will potentially store *all* packed micro-panels of A that will be generated during the iterations of loop `L3` in Figure 1, and hence its global dimension initially matches that of A. However, depending on the placement of the packing specified by the user, TVM will evaluate the potential reuse level of the buffer(s) and will accordingly adapt the buffer dimensions. (This will be refined in the following discussion of part P5.) The lambda function in lines 10–11 specifies the desired packing scheme by mapping the elements of `Ac` to their counterparts in A. Concretely, the four arguments (`i, j, q, r`) of the function determine how the element `Ac[i,j,q,r]` is extracted from A conformally with the packing scheme for the baseline algorithm.

```
 1 def packed_GEMM_B3A2C0(m, n, k, mc, nc, kc, mr, nr, dtype):
 2   # P1) Declare the input operands
 3   #     As in block_GEMM_B3A2C0. Omitted for brevity
 4 
 5   # P2) Define the operation with (blocking and) packing
 6 
 7   #     Ac: 4D tensor with packed version of A
 8   Ac = te.compute((math.ceil(k/kc),
 9                    math.ceil(m/mr), kc, mr),
10                    lambda i, j, q, r:
11                    A[j * mr + r, i * kc + q],
12                    name="Ac")
13 
14   #     Bc: 4D tensor with packed version of B
15   Bc = te.compute((math.ceil(k/kc),
16                    math.ceil(n/nr), kc, nr),
17                    lambda i, j, q, r:
18                    B[ i * kc + q, j * nr + r ],
19                    name="Bc")
20 
21   p = te.reduce_axis((0, k), "p")
22   C = te.compute((m, n), lambda i, j:
23          te.sum(Ac[p//kc, i//mr,
24                     tvm.tir.indexmod(p, kc),
25                     tvm.tir.indexmod(i, mr)] *
26                  Bc[p//kc, j//nr,
27                     tvm.tir.indexmod(p, kc),
28                     tvm.tir.indexmod(j, nr)],
29                  axis=p), name="C")
30 
31   # P3) Prepare the schedule with blocking
32   #     As in block_GEMM_B3A2C0. Omitted for brevity
33 
34   # P4) Specify the loop ordering (schedule as in baseline/B3A2C0)
35   #     As in block_GEMM_B3A2C0. Omitted for brevity
36 
37   # P5) Emplace the packings
38   sched[Bc].compute_at(sched[C], pc)
39   sched[Ac].compute_at(sched[C], ic)
40 
41   # P6) Apply fine-grain optimizations
42   #     No optimizations applied
43 
44   # P7) Loop-level parallelization
45   #     No parallelization in this version
46 
47   # P8) Generate code with LLVM backend
48   return tvm.build(sched, [A, B, C], target="llvm")
```

Fig. 7. TVM generator for GEMM mimicking the (blocking scheme and) *packing* of the baseline algorithm.

Lines 15–19, involving the tensor Bc and the matrix B, play an analogous role to that described previously for Ac and A.

Lines 21–29 define the operation to be computed in terms of the tensors Ac and Bc. Here Ac is transposed (with respect to A), which is necessary to ensure that TVM generates a code that accesses the entries of Ac with unit stride. Also the computation is defined in terms of a reduction (sum) across the dimension k.

Part P5. Placement of the packings: Lines 38–39 place the computation of tensors Bc and Ac at the desired points of the loop nesting: Concretely, inside the loop indexed by pc for Bc and the loop indexed by ic for Ac; see the algorithm on the left of Figure 1. As a result of combining the correct placement of the tensor buffers, loop ordering, and access pattern in the blocked computation of C, the packed tensors Ac and Bc do not need to have the same size as A and B, respectively. Instead, they are computed at each iteration of loop L3 (indexed by ic) and loop L2 (indexed by pc). In the latter case, for example, the reuse pattern of elements for Ac guides the compiler to reduce the dimension of the packed buffer from $m \times k$ to $m_c \times k_c$ only (which matches the dimension of the buffer A_c in Figure 4). An analogous comment applies to matrix B and its packed tensor counterpart.

The purpose of the packing operations is to favor a higher number of cache hits when accessing the data in the L2 and L1 caches from the micro-kernel. The packing introduces a certain overhead, but this is usually low because the packed data is re-utilized multiple times [21]. As a side note, let us comment that, depending on the problem dimensions (m, n, k) and the loop strides (m_c, n_c, k_c), in some cases the costs of the packing operations may exceed their benefits. Fortunately, with TVM we can easily eliminate any of the packing operations. For example, in the baseline algorithm in Figure 7, the two packing operations can be eliminated by 1) removing lines 8-19; changing the references from Ac and Bc to A and B in lines 23, 26; and 2) adapting accordingly the indexing on A and B to accommodate the dimensionality modification from Ac and Bc (from 4D to 2D).

The intermediate representation (IR) produced by TVM using the generator for the packed realization of GEMM can be described as follows:

```
for (jc=0; jc<n; jc+=nc) {           // Loop L1
  for (pc=0; pc<k; pc+=kc) {         // L2
    for (i=0; ...) {                 // Packing Bc
      for (j=0; ...)
        for (q=0; ...)
          for (r=0; ...)
            Bc[...] = B[...];
    }
    for (ic=0; ic<m; ic+=mc) {       // L3
      for (i=0; ...) {               // Packing Ac
        for (j=0; ...)
          for (q=0; ...)
            for (r=0; ...)
              Ac[...] = A[...];
      }
      for (jr=0; jr<nc; jr++)       // L4
        for (ir=0; ir<mc; ir++)     // L5
          for (pr=0; pr<kc; pr++) // L6
            C[...] += Ac[...] * Bc[...];
} } }
```

(Here we omitted a few indexing details and introduced some formatting to ease the interpretation.)

This takes us one step forward toward automatically generating a high performance realization of GEMM that mimics the baseline algorithm for GEMM by removing one of the caveats that appeared in the previous blocking algorithm: the lack of packing routines. In the next section we address the second missing detail: *the formulation of the innermost loop of the algorithm as a micro-kernel that performs an outer product.*

6 AUTOMATIC GENERATION OF HIGH PERFORMANCE MICRO-KERNELS FOR GEMM

The final stage in our journey to obtain a high performance realization of GEMM has the goal of integrating the automatic generation of high performance micro-kernels within the TVM routine for GEMM.

6.1 Micro-kernel with Resident C

Starting from the TVM generator in Figure 7, we introduce two main changes with the result shown in Figure 8:

```
 1 def packed_GEMM_B3A2C0_ukernel(m, n, k, mc, nc, kc, mr, nr, dtype):
 2   # P1), P2) Declare the input operands and define the operation
 3   #          As in packed_GEMM_B3A2C0. Omitted for brevity
 4 
 5   # P3) Prepare the schedule with blocking
 6   sched = te.create_schedule(C.op)
 7   ic, jc, \
 8   ir, jr = sched[C].tile(C.op.axis[0],
 9                          C.op.axis[1], mc, nc)
10   pc, pr = sched[C].split(p, factor=kc)
11 
12   #     Expose loops inside micro-kernel
13   ir, it = sched[C].split(ir, factor=mr)
14   jr, jt = sched[C].split(jr, factor=nr)
15 
16   # P4) Specify the loop ordering (schedule as in baseline/B3A2C0)
17   sched[C].reorder(jc, pc, ic,
18                    jr, ir, pr, it, jt)
19 
20   # P5) Emplace the packings
21   #     As in packed_GEMM_B3A2C0
22 
23   # P6) Apply fine-grain optimizations
24   #     No optimizations applied
25 
26   # P7) Loop-level parallelization
27   #     No parallelization in this version
28 
29   # P8) Generate code with LLVM backend
30   return tvm.build(sched, [A, B, C], target="llvm")
```

Fig. 8. TVM generator for GEMM mimicking (the blocking and packing of) the baseline algorithm, and *integrating the optimized micro-kernel with Resident C*.

Part P3. Preparation of the schedule (with additional blocking): The m_c, n_c dimensions of the problem are respectively partitioned using factors m_r, n_r, in order to expose the loops that will operate with the individual elements of the micro-panel C_r; see lines 13–14.

Part P4. Specification of the loop ordering: Lines 17–18 place the newly created loops for the micro-kernel, `it` and `jt`, within the global schedule.

6.2 Fine-grain optimizations for high performance and parallelization

The TVM generator can explore relevant optimizations at the micro-kernel level which pursue goals that are similar to those of the low-level optimizations applied by the expert Assembly programmer. By off-loading these optimizations to TVM, in exchange for less flexibility, there is a considerably more reduced programming effort and higher performance portability across different processor architectures.

Starting from the TVM generator in Figure 8, the enhanced alternative in Figure 9 illustrates the necessary additions to introduce some conventional optimization techniques:

Part P6. Fine-grain optimizations: This invokes TVM methods to address three types of optimizations:

- *SIMD vectorization:* The generator includes vectorization of three different stages of the blocked algorithm to leverage the SIMD units of the architectures: micro-kernel computation (lines 8–10); packing A into A_c (lines 13–16); and packing B into B_c (lines 19–22). The basic scheme remains the same for all of them: the innermost loop (jt for the former and l in latter two) is split to expose an additional inner loop, using a split factor (`lanesize`) that matches the SIMD width of the target architecture. For example, for 32-bit floating point data, `lanesize` is set to 4 for ARMv8a NEON (128-bit SIMD registers) and 16 for

Intel AVX512 (512-bit SIMD registers). In general, in our experiments we will employ the exact number of elements for a SIMD register, or an integer multiple of it. From the newly exposed loops, the outer one is unrolled and the inner is vectorized (`unroll` and `vectorize` methods, respectively) This produces a code comprising vector instructions for the desired target processor.

- *Prefetching and vector loads within the micro-kernel:* `ac` and `bc` are tensors defined as read-only copies of `Ac` and `Bc`. The purpose of creating these artifacts is two-fold: first, it helps the compiler to introduce software-prefetching instructions in order to pre-load the micro-panels of `Ac` and `Bc` that will be used throughout the computation of the micro-kernel; second, it instructs the compiler to use vector registers in order to store the active parts of `ac` and `bc` in the micro-kernel, and to use vector instructions to load them from `Ac` and `Bc`. Diving into the details, the construction, re-use pattern and computation point (loop indexed by `pr`) of `ac` and `bc` induce buffers that store the strictly necessary micro-panels of `Ac` and `Bc` used within the computation of each iteration of loop `pr` within the micro-kernel, that is, $m_r \times 1$ for `ac`, and $n_r \times 1$ for `bc`. Splitting the innermost loop (`l`) using a factor that matches `lanesize`, unrolling and vectorizing it yield vector instructions to load the micro-panels of `Ac` and `Bc` into $m_r/lanesize$ and $n_r/lanesize$ vectors.

Part P7. Loop-level parallelization: TVM also allows exploiting loop parallelism via multi-threading. Line 41 shows how to instruct TVM to split the iteration space for a given loop across the active threads. The selection of the optimal loop to parallelize depends on the problem dimensions and target architecture specifications; see [27] for a detailed discussion.

Figure 10 displays two examples of the assembly code generated by TVM: An $m_r \times n_r = 4 \times 4$ micro-kernel for an ARMv8a ISA (instruction set architecture) with NEON vector instructions and unrolling factor of 4 on the left; and an $m_r \times n_r = 4 \times 16$ micro-kernel for an x86 ISA with AVX512 vector instructions (on the right). In both cases, for brevity, we only include the instructions that are comprised by the reduction loop of the micro-kernel. The codes were obtained using the generator for the optimized B3A2C0 variant depicted in Figure 9, selecting the target as `llvm -device=arm_cpu -mattr=+v8.2a,+fp-armv8,+neon` for the ARMv8a architecture, and `llvm -mcpu=icelake-server` for the x86 architecture. The codes share a similar structure, but present subtle differences in terms of syntax and vectorization strategy: The ARMV8a codes rely on vector fused multiply-add instructions with a single element of B as a source (e.g. `fmla v3.4s,v5.4s,v4.s[0]`); in contrast, the x86 counterpart operates on a broadcast operation and vector fused multiply-add strategy (`vbroadcastss %xmm4,%zmm8` followed by `vfmadd213ps %zmm3,%zmm6,%zmm8`).

7 OTHER MEMBERS IN THE FAMILY OF BLOCKED ALGORITHMS FOR GEMM

One of the advantages of TVM lies in that producing code for other blocked algorithms of the GEMM family only requires small changes in the generator routines in order to accommodate the proper loop ordering, blocking and packing schemes. This is described in this section.

7.1 Blocking and packing for variants with C in the L2 cache

Transforming the TVM generator for the baseline algorithm, with a micro-tile of Resident C, (left column in Figure 1) into that with a block of C in the L2 cache and a micro-tile of Resident B (A3C2B0, right column in Figure 2) requires a number of changes in the TVM generator, shown in Figure 7, resulting in the variant displayed in Figure 11:

Part P0. Parameter list: This generator includes `kr` as a parameter (instead of `mr`).

```python
def opt_GEMM_B3A2C0_ukernel(m, n, k, mc, nc, kc, mr, nr, lanesize, dtype):
  # P1), P2) P3) P4) P5) as in packed_GEMM_B3A2C0_ukernel
  #     Omitted for brevity

  # P6) Apply fine-grain optimizations

  #     6.1. Vectorization of the microkernel
  jto,jt = sched[C].split(jt, factor=lanesize)
  sched[C].unroll(jto)
  sched[C].vectorize(jt)

  #     6.2. Vectorization of the packing of A
  x,y,z,l = Ac.op.axis
  l,ll = sched[Ac].split(l, factor=lanesize)
  sched[Ac].unroll(l)
  sched[Ac].vectorize(ll)

  #     6.3. Vectorization of the packing of B
  x,y,z,l = Bc.op.axis
  l,ll = sched[Bc].split(l, factor=lanesize)
  sched[Bc].unroll(l)
  sched[Bc].vectorize(ll)

  #     6.4. Prefetching
  ac = sched.cache_read(Ac, "global", readers=[C])
  bc = sched.cache_read(Bc, "global", readers=[C])

  sched[ac].compute_at(sched[C],pr)
  x,y,z,l = ac.op.axis
  l,ll = sched[ac].split(l, factor=lanesize)
  sched[ac].unroll(l)
  sched[ac].vectorize(ll)

  sched[bc].compute_at(sched[C],pr)
  x,y,z,l = bc.op.axis
  l,ll = sched[bc].split(l, factor=lanesize)
  sched[bc].unroll(l)
  sched[bc].vectorize(ll)

  # P7) Loop-level parallelization on ic
  sched[C].parallel(ic)

  # P8) Generate code with LLVM backend
  return tvm.build(sched, [A, B, C],
                   target="llvm")
```

Fig. 9. TVM generator for GEMM mimicking (the blocking and packing of) the baseline algorithm, integrating both the optimized micro-kernel with Resident C and *fine-grain optimizations and a loop-level parallelization of loop* ic.

Part P2. Definition of the operation and packing schemes for A ***and*** B***:*** Lines 15–19 modify the dimensions of the Bc tensor (this version moves B to registers) so that the size of each submatrix (micro-panel) in Bc is $k_r \times n_r$.

Part P3. Preparation of the schedule: Lines 34–37 apply tiling over C in preparation for the placement of the packing into the buffer Cc. In addition, line 39 creates a buffer for reading and writing the matrix C. From that point, the scheduler operates with the object Cc. This change is crucial for the algorithms where C resides in either the L2 or L3 caches. Concretely, these variants leverage `cache_write` to induce a copy from matrix C to the object Cc as well as to restore the result of the micro-kernel, temporarily in Cc, back into C.

Part P4. Specification of the loop ordering: Line 48 specifies a loop order that matches that of the algorithm A3C2B0.

```
.LBB1_5:
  add   x11, x9, x10
  add   x12, x8, x10
  ldr   q4, [x11, #15952]
  ldr   q5, [x12, #15968]
  ldr   q6, [x11, #15968]
  ldr   q7, [x12, #15984]
  ldr   q16, [x11, #15984]
  ldr   q17, [x12, #16000]
  fmla v3.4s, v5.4s, v4.s[0]
  fmla v2.4s, v5.4s, v4.s[1]
  fmla v1.4s, v5.4s, v4.s[2]
  fmla v0.4s, v5.4s, v4.s[3]
  ldr   q4, [x11, #16000]
  ldr   q5, [x12, #16016]
  fmla v3.4s, v7.4s, v6.s[0]
  fmla v2.4s, v7.4s, v6.s[1]
  fmla v1.4s, v7.4s, v6.s[2]
  fmla v0.4s, v7.4s, v6.s[3]
  fmla v3.4s, v17.4s, v16.s[0]
  fmla v2.4s, v17.4s, v16.s[1]
  fmla v1.4s, v17.4s, v16.s[2]
  fmla v0.4s, v17.4s, v16.s[3]
  adds x10, x10, #64
  fmla v3.4s, v5.4s, v4.s[0]
  fmla v2.4s, v5.4s, v4.s[1]
  fmla v1.4s, v5.4s, v4.s[2]
  fmla v0.4s, v5.4s, v4.s[3]
  b.ne .LBB1_5
```

```
.LBB5_12:
  vmovaps        -16(%rdx,%r13), %xmm4
  vmovaps        (%rdx,%r13), %xmm5
  vmovups        -64(%rsi,%r13,4), %zmm6
  vmovups        (%rsi,%r13,4), %zmm7
  vbroadcastss   %xmm4, %zmm8
  vfmadd213ps    %zmm3, %zmm6, %zmm8
  vmovshdup      %xmm4, %xmm3
  vbroadcastsd   %xmm3, %zmm9
  vfmadd213ps    %zmm2, %zmm6, %zmm9
  vpermilps      $234, %xmm4, %xmm2
  vbroadcastsd   %xmm2, %zmm10
  vfmadd213ps    %zmm1, %zmm6, %zmm10
  vpermilps      $239, %xmm4, %xmm1
  vbroadcastsd   %xmm1, %zmm4
  vfmadd213ps    %zmm0, %zmm6, %zmm4
  vbroadcastss   %xmm5, %zmm3
  vfmadd213ps    %zmm8, %zmm7, %zmm3
  vmovshdup      %xmm5, %xmm0
  vbroadcastsd   %xmm0, %zmm2
  vfmadd213ps    %zmm9, %zmm7, %zmm2
  vpermilps      $234, %xmm5, %xmm0
  vbroadcastsd   %xmm0, %zmm1
  vfmadd213ps    %zmm10, %zmm7, %zmm1
  vpermilps      $239, %xmm5, %xmm0
  vbroadcastsd   %xmm0, %zmm0
  vfmadd213ps    %zmm4, %zmm7, %zmm0
  addq           $32, %r13
  cmpq           $2048, %r13
  jne            .LBB5_12
```

Fig. 10. Assembly codes for the reduction loop (traversing k_c) of micro-kernels with Resident C automatically generated by TVM for single precision. Left: ARMv8a assembly code with NEON vector instructions for a 4×4 micro-kernel with a loop unrolling technique with a factor of 4. Right: x86 assembly code with AVX512 vector instructions for a 4×16 micro-kernel.

- ***Part P5. Placement of the packings:*** Lines 51–53 set the buffer for `Cc` and the `Ac` and `Bc` tensors into the appropriate loops.

The TVM generator in Figure 11 does not include fine-grain optimizations, yet identical techniques as those introduced in Section 6.2 apply for this member of the family. Also, the TVM generator to obtain variant B3C2A0 can be easily derived following the same principles.

7.2 Blocking and packing for variants with C in the L3 cache

The remaining two blocked algorithmic variants for GEMM in the family store a block of matrix C in the L3 cache throughout the computation. Starting from the TVM generator for variant A3C2B0 in Figure 11, the following changes are introduced to obtain the missing TVM generator, for variant C3A2B0, displayed in Figure 12:

- ***Part P4. Specification of the loop ordering:*** Line 12 enforces a loop ordering which mimics that of the algorithm in Figure 2 (right).
- ***Part P5. Placement of the packings:*** The buffer C_c does not need to be bound to any loop because it belongs to the outer structure. The packings of `Ac` and `Bc` are placed in the same loops as in the version with C in the L2 cache (see lines 15–16).

Obtaining the last variant, C3B2A0, is direct and, therefore, its discussion is omitted for brevity.

8 EXPERIMENTAL RESULTS

In this section, we provide strong experimental evidence that the TVM-based approach to automatically generate routines for the matrix multiplication paves an almost effortless road toward

```
def packed_GEMM_A3C2B0(m, n, k, mc, nc, kc, kr, nr, dtype):
  # P1) Declare the input operands
  #     As in packed_GEMM_B3_A2C0. Omitted for brevity

  # P2) Define the operation with blocking and packing

  #     Ac: 4D tensor with packed version of A
  Ac = te.compute((math.ceil(m/mc),
                   math.ceil(k/kr), mc, kr),
                   lambda i, j, q, r:
                   A[i * mc + q, j * kr + r],
                   name="Ac")

  #     Bc: 4D tensor with packed version of B
  Bc = te.compute((math.ceil(k/kr),
                   math.ceil(n/nr), kr, nr),
                   lambda i, j, q, r:
                   B[i * kr + q, j * nr + r],
                   name="Bc")

  p = te.reduce_axis((0, k), "p")
  C = te.compute((m, n), lambda m, n:
         te.sum(Ac[m // mc, p // kr,
                   tvm.tir.indexmod(m, mc),
                   tvm.tir.indexmod(p, kr)] *
                Bc[p // kr, n // nr,
                   tvm.tir.indexmod(p, kr),
                   tvm.tir.indexmod(n, nr)],
                axis=p), name="C")

  # P3) Prepare the schedule with blocking
  sched = te.create_schedule(C.op)

  cic, cjc, \
  cir, cpr = sched[C].tile(C.op.axis[0],
                           C.op.axis[1],
                           mc, nc)

  Cc = sched.cache_write(C, "global")

  ic, jc, \
  ir, jr = sched[Cc].tile(Cc.op.axis[0],
                          Cc.op.axis[1],
                          mc, nc)
  pc, pr = sched[Cc].split(p, factor=kc)

  # P4) Specify loop ordering (schedule as in A3C2B0)
  sched[Cc].reorder(ic, pc, jc, pr, jr, ir)

  # P5) Define packing placement
  sched[Cc].compute_at(sched[C], cic)
  sched[Ac].compute_at(sched[Cc], pc)
  sched[Bc].compute_at(sched[Cc], ir)

  # P6) Apply fine-grain optimizations
  #     No fine-grain optimizations applied.

  # P7) Loop-level parallelization
  #     No parallelization in this version

  # P8) Code generation
  return tvm.build(sched, [A, B, C], target="llvm")
```

Fig. 11. TVM generator for GEMM mimicking the blocking and packing schemes of the A3C2B0 algorithm.

experimenting with a rich variety of algorithms, micro-kernels and parallelization/optimization options that offer fair performance in a number of scenarios.

```python
def packed_GEMM_C3A2B0(m, n, k, mc, nc, kc, kr, nr, dtype):
  # P1) Declare the input operands
  #     As in packed_GEMM_A3C2B0. Omitted for brevity

  # P2) Define the operation with blocking and packing
  #     As in packed_GEMM_A3C2B0. Omitted for brevity

  # P3) Prepare the schedule with blocking
  #     As in packed_GEMM_A3C2B0. Omitted for brevity

  # P4) Specify loop ordering (schedule as in C3A2B0)
  sched[Cc].reorder(jc, ic, pc, jr, pr, ir)

  # P5) Emplace the packings
  sched[Ac].compute_at(sched[Cc], pc)
  sched[Bc].compute_at(sched[Cc], ir)

  # P6) Apply fine-grain optimizations
  #     No fine-grain optimizations applied.

  # P7) Loop-level parallelization
  #     No parallelization in this version

  # P8) Code generation
  return tvm.build(sched, [A, B, C], target="llvm")
```

Fig. 12. TVM generator for GEMM mimicking the blocking and packing schemes of the C3A2B0 algorithm.

8.1 General setup

Unless otherwise explicitly stated, the experiments in this section were carried out using a single core of the NVIDIA Carmel processor (ARM v8.2) embedded on an NVIDIA Jetson AGX Xavier board, using IEEE 32-bit floating point arithmetic (FP32). In order to reduce variability, the processor frequency was fixed to 2.3 GHz, the threads were bound to the hardware cores, and the experiments were repeated a large number of times reporting in the following the average results for each experiment. In general, performance is measured in terms of billions of arithmetic operations per second, abbreviated as GFLOPS when operating with floating point arithmetic and GIOPS for integer arithmetic.

Also, unless explicitly stated, we target the baseline algorithm for GEMM with a micro-kernel that operates with Resident C. For reference, when possible we include in the evaluation the results obtained from up-to-date realizations of the GEMM kernel in libraries such as BLIS (version v0.8.1), OpenBLAS (version v0.3.19), and the ARM Performance Libraries (ARMPL, version v21.1).

The dataset for the experimentation includes two types of GEMM problems: large square matrices versus highly "rectangular" problems. Given the current interest in DL inference, the dimensions of the latter are selected as those that result from applying the IM2COL transform [6] to cast the convolution layers in the ResNet50 v1.5 deep neural network (DNN) model in terms of a GEMM. The "batch" size for the inference scenario is set to 128 samples. As some layers share the same parameters, resulting in GEMM problems of the same dimensions, we report the results for those only once (per layer type); see Table 1. In the following, we will use "layer" to refer to "layer type", since we are only interested in the "layer dimensions".

8.2 Cache configuration parameters

The performance of the GotoBLAS2-like realizations of GEMM as well as our routines obtained with TVM is strongly dictated by the optimization level of the micro-kernel and a proper selection of the cache configuration parameters m_c, n_c, k_c. The optimal values for the latter three parameters depend on hardware features such as number of cache levels, size, set associativity, etc., and the

Table 1. Dimensions of the GEMM resulting from applying the IM2COL transform to the layers of the ResNet50 v1.5 DNN model with a batch size of 128 samples.

Layer type id.	Layer numbers in ResNet50 v1.5	m	n	k	Layer type id.	Layer numbers in ResNet50 v1.5	m	n	k
1	001	1,605,632	64	147	11	080	100,352	256	512
2	006	401,408	64	64	12	083/095/105/115/125/135	25,088	256	2,304
3	009/021/031	401,408	64	576	13	086/098/108/118/128/138	25,088	1,024	256
4	012/014/024/034	401,408	256	64	14	088	25,088	1,024	512
5	018/028	401,408	64	256	15	092/102/112/122/132	25,088	256	1,024
6	038	401,408	128	256	16	142	25,088	512	1,024
7	041/053/063/073	100,352	128	1,152	17	145/157/167	6,272	512	4,608
8	044/056/066/076	100,352	512	128	18	148/160/170	6,272	2,048	512
9	046	100,352	512	256	19	150	6,272	2,048	1,024
10	050/060/070	100,352	128	512	20	154/164	6,272	512	2,048

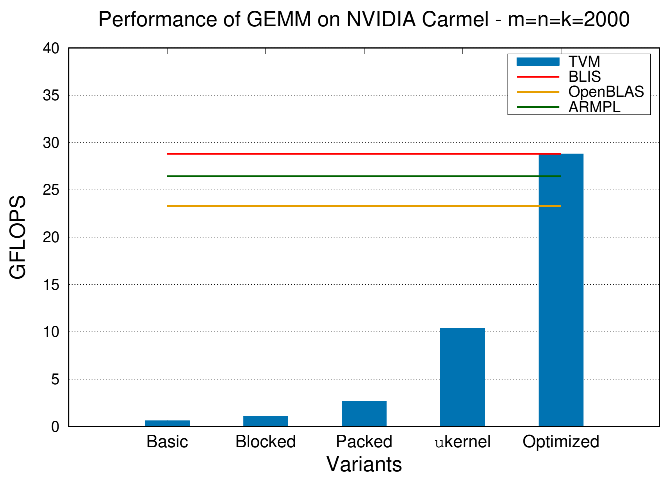


Fig. 13. Performance comparison of distinct TVM generators on a single NVIDIA Carmel core for square matrices.

specific dimensions of the micro-kernel (given by $m_r \times n_r$ for a Resident C case). Determining these values via brute force experimentation involves an expensive search across a large 3D space, possibly for each micro-kernel dimension.

Alternatively, one can use the analytical model in [21] to select the optimal values for the cache configuration parameters. The advantage of this analytical approach in our particular case will be exposed in the next subsection, when we automatically generate, explore and evaluate a variety of micro-kernels, of different dimensions, using the TVM routine.

8.3 Comparison of the TVM generators

We open the experimental evaluation by illustrating the performance boost that is obtained by incrementally integrating the blocking, packing, and fine-grained optimizations described with the successive TVM generators. For simplicity, in this initial analysis we only consider a square GEMM problem with dimensions given by $m = n = k = 2{,}000$.

Figure 13 shows the results for this first experiment. The five blue bars there report the performance attained by the GEMM routines automatically obtained with the five TVM generators described in Sections 5 and 6. Concretely, the labels Basic, Blocked, Packed, μkernel, and Optimized in the x-axis respectively refer to the TVM generators `basic_GEMM`, `block_GEMM_B3A2C0`, `packed_GEMM_B3A2C0`, `packed_GEMM_B3A2C0_ukernel`, and `opt_GEMM_B3A2C0_ukernel` in Figures 5, 6, 7, 8, and 9. For the latter, parallelization is disabled (line 41 of Figure 9). Hence, the reported performance results correspond to a sequential GEMM execution. For reference, the figure also displays the performance attained with the realizations of the GEMM kernel in BLIS (red line), OpenBLAS (yellow line), and ARMPL (green line) for this particular problem size (on the NVIDIA Carmel core).

For this first experiment, we observe the notable performance raise attained with the integration of the micro-kernel and the fine-grain optimizations (rightmost two bars) as well as the fact that, for this problem dimension (and target processor core), the best TVM generator delivers a code for GEMM whose performance exactly matches that of the BLIS kernel for this operation. This is not totally surprising since, for this initial experiment, we forced TVM to generate a routine with exactly the same cache configuration parameters and micro-kernel dimensions as those used by BLIS. Hereafter, all our TVM results correspond to the version of GEMM obtained with the optimized TVM generator `opt_GEMM_B3A2C0_ukernel`.

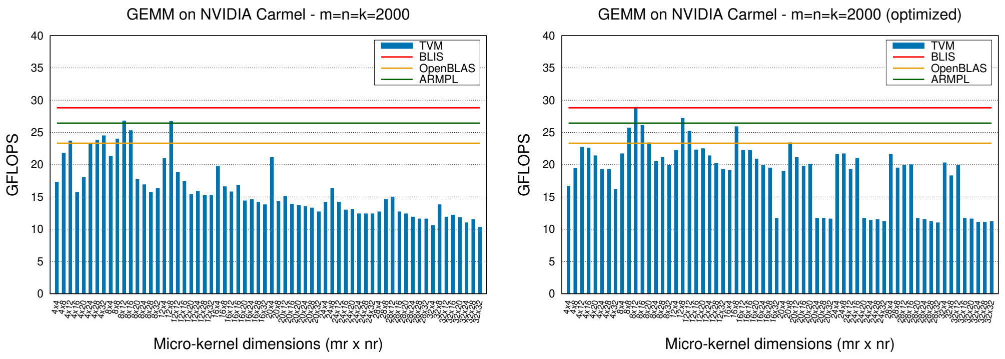


Fig. 14. Performance evaluation of the TVM generator on a single NVIDIA Carmel core for square matrices, without and with fixed lane size/prefetching (left and right, respectively).

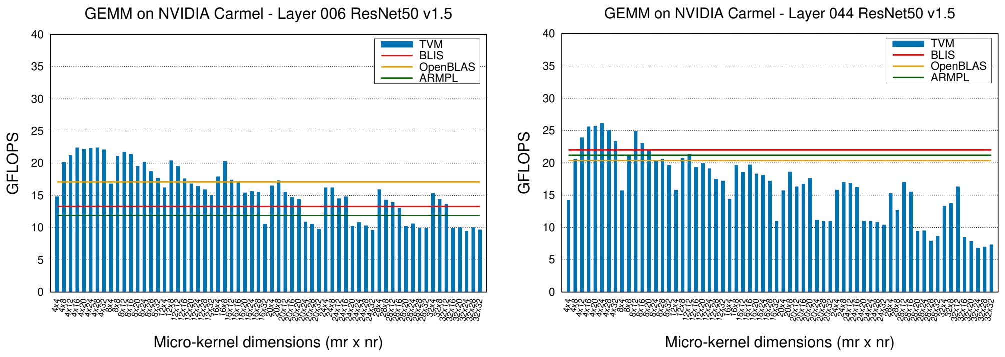


Fig. 15. Performance evaluation of the TVM generator on a single NVIDIA Carmel core for layers 2 (left) and 8 (right) of ResNet50 v1.5.

8.4 In search of the best micro-kernel

In this subsection we demonstrate the benefits of being able to automatically generate and subsequently evaluate a variety of micro-kernels for a particular problem dimension. (In comparison, a manual development is a costly process that requires a high level of expertise.)

8.4.1 Effect of the micro-kernel. We unfold the analysis of the micro-kernels by assessing the performance of the TVM-generated routines for the baseline algorithm that integrate micro-kernels of different dimensions: For a square problem with $m = n = k = 2{,}000$ in Figure 14, and for the GEMM operations arising in two layers of the ResNet model in Figure 15. We clarify here that the only difference between these micro-kernels is their dimensions. Thus, other optimization possibilities, such as varying the loop unrolling factor, selecting the loop that to vectorize, etc. were not evaluated in this experiment and were kept constant across all experiments. With respect to the two plots in Figure 14, they evaluate the impact of two low-level optimizations included in the TVM generator in Figure 9:

- *Lane size:* The innermost loop of GEMM is further split using a factor that is an integer multiple of the number of elements that fit into one vector register (e.g., 8 for FP16, 4 for FP32, or 2 for FP64 in the NVIDIA Carmel processor, for which the vector registers are 128-bit wide).
- *Prefetching:* The TVM method `cache_read` is invoked to induce the scheduler to include an automatic *prefetching* when possible.

The left plot in Figure 14 was obtained by removing Part 6.4 (lines 24 to 38) in the TVM generator in Figure 9, while the right plot was obtained with these lines included. In both cases, the performance results correspond to sequential executions.

As a result from these additional optimizations, the performance of the TVM routine displayed in the right plot of Figure 14, when setting the micro-kernel dimension to $m_r \times n_r = 8 \times 12$, improves from 16 to 18.7 GFLOPS with respect to its counterpart without these low-level optimizations. Therefore, from now on we will only report results for the TVM-generated routines with these two optimizations in place.

The right plot in Figure 14 demonstrates that a careful selection of the micro-kernel is critical to attain high performance. Concretely, the performance achieved by using micro-kernels of different dimensions varies between 10.3 GFLOPS (worst case, with $m_r \times n_r = 32 \times 32$) and 26.8 GFLOPS (best case, with $m_r \times n_r = 8 \times 12$). The two performance plots in Figure 15 also contribute to show that the best micro-kernel is largely dependent on the problem dimension. Specifically, the highest GFLOPS rates are observed for the micro-kernels $m_r \times n_r = 4 \times 16$ and 4×28 for layer 2, compared with the micro-kernel $m_r \times n_r = 4 \times 24$ for layer 8.

A direct comparison between the realizations of GEMM in "hand-encoded" libraries and the TVM routine *using the best micro-kernel* for each problem case reveals that, for square matrices, the TVM routine delivers GFLOPS rates that are similar to those attained with BLIS (28.9 GFLOPS for the former versus 28.8 GFLOPS for latter), and superior with respect to OpenBLAS (23.3 GFLOPS) and ARMPL (26.4 GFLOPS). The scenario is different for the ResNet50 problems: For layer 2, the TVM routine (with the best micro-kernel) delivers 22.4 GFLOPS versus 13.3 GFLOPS for BLIS, 17.1 GFLOPS for OpenBLAS, and 11.9 for ARMPL. In addition, for layer 8, the TVM routine (with the best micro-kernel) achieves 26.1 GFLOPS versus 22.0 GFLOPS for BLIS (best library-based option). From this point, we will always report the results for the TVM routine with the best micro-kernel for each problem dimension.

The evaluation of the GEMM operations associated with all the layers in the ResNet50 model shows a variety of scenarios. Concretely, Figure 16 illustrates that the TVM routine outperforms the BLIS realization by a large margin for layer 1–12 and by a visible difference for layers 15–17 (40

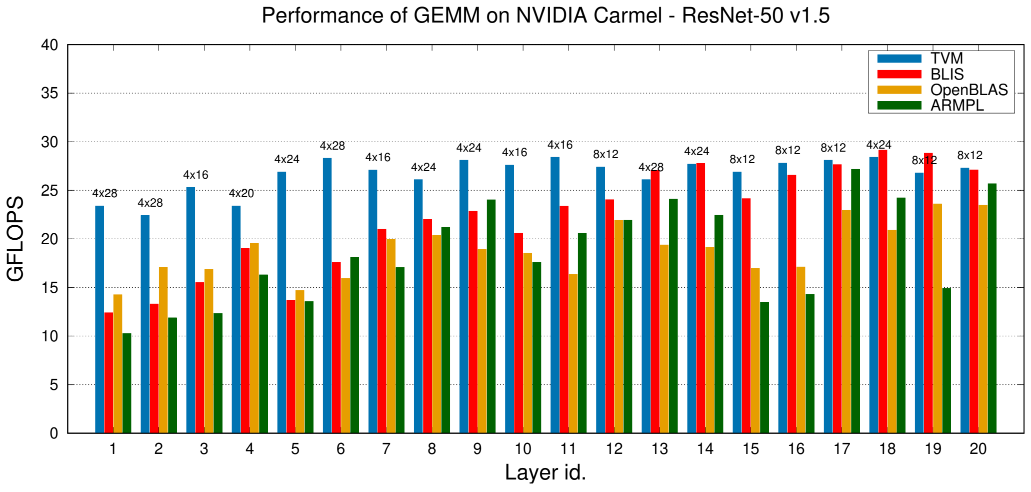


Fig. 16. Performance evaluation of the TVM generator on a single NVIDIA Carmel core for ResNet50 v1.5. The number on top of each blue bar represents the dimension $m_r \times n_r$ of the best micro-kernel for that layer.

cases out of the total 53 convolution layers in the model, see Table 1); it is competitive for layer 14 and 20 (3 cases); and it is suboptimal for layer 13, 18, 19 (10 cases). Compared with OpenBLAS and ARMPL, the TVM routine is consistently better. An analysis of the results taking into account the operands' dimensions shows that the TVM routine delivers higher performance for "rectangular" cases, with m in the range 100,352–1,605,632, and it is competitive when m=25,088. In contrast, BLIS is better choice for "square" problems, with m in the range of 6,000.[4]

8.4.2 Why is TVM better? The superiority of the TVM routine is rooted in the fact that, by (automatically) generating the micro-kernels of different dimensions, we can easily explore the space and select the one that is better suited to a particular problem dimension. This is illustrated in Figure 16, which reports the best micro-kernel for each problem dimension/DNN layer. Compared with that, BLIS, OpenBLAS and ARMPL each integrate a single, manually-encoded micro-kernel, which is therefore the only option for any problem dimension. The reason for this limitation of the libraries is that manually implementing different micro-kernels is a time-consuming task, requiring significant experience in high performance computing, computer architecture, and assembly coding. In addition, the logic of selecting the appropriate micro-kernel dimension based on problem dimensions is usually not supported in commercial or academic libraries.

Delving further into the matter, the theoretical reasons behind this behavior can be explained using the analytical model in [21]. According to that, for problems with a reduced dimension k, which in practice limits the effective value for the cache parameter k_c, the micro-panel of B_c that is stored in the L1 cache (B_r, of dimensions $k_c \times n_r$) does not attain the optimal occupation of that level, explaining the performance penalty.

Let us illustrate the problem for layer 2 and 8 of ResNet v1.5, whose experimental performance for different micro-kernel dimensions was exposed in Figure 15. We complement that figure with the values in Table 2, illustrating the cache parameters m_c, n_c, k_c and the theoretical occupation

[4]As a side note, the actual processing cost of the Resnet50 v1.5 model is concentrated in those cases where m is in the range 100,352–1,605,632 (47.8% of the total time), followed by m=25,088 (35.6% of the total time). In terms of absolute cost, this implies that the execution of all layers employing the TVM routine would require 39.1 s compared with 48.0 s when using BLIS (and higher for OpenBLAS and ARMPL).

Table 2. Detailed L1 cache occupation for layers 2 and 8 of ResNet50 v1.5 for different micro-kernel dimensions.

Layer type id.	m_c	n_c	k_c	m_r	n_r	L1 (%)	Max (%)	Layer type id.	m_c	n_c	k_c	m_r	n_r	L1 (%)	Max (%)
2	7,168	64	64	4	4	1.56	25	8	3,584	256	128	4	4	3.12	25
2	6,656	64	64	4	8	3.12	50	8	3,584	256	128	4	8	6.25	50
2	6,144	64	64	4	12	4.69	50	8	3,328	275	128	4	12	9.38	50
2	6,144	64	64	4	16	6.25	50	8	3,328	275	128	4	16	12.50	50
2	5,632	64	64	4	20	7.81	50	8	3,072	298	128	4	20	15.60	50
2	5,120	64	64	4	24	9.38	50	8	3,072	298	128	4	24	18.80	50
2	5,120	64	64	4	28	10.90	50	8	3,072	298	128	4	28	21.90	50
2	7,168	64	64	8	4	1.56	25	8	3,584	256	128	8	4	3.12	25
2	6,656	64	64	8	8	3.12	25	8	3,584	256	128	8	8	6.25	25
2	6,144	64	64	8	12	4.69	50	8	3,328	275	128	8	12	9.38	50
2	7,168	64	64	12	4	1.56	25	8	3,584	256	128	12	4	3.12	25
2	6,656	64	64	12	8	3.12	25	8	3,584	256	128	12	8	6.25	25
2	7,168	64	64	16	4	1.56	25	8	3,584	256	128	16	4	3.12	25
2	7,168	64	64	20	4	1.56	25	8	3,584	256	128	20	4	3.12	25
2	7,168	64	64	24	4	1.56	25	8	3,584	256	128	24	4	3.12	25
2	7,168	64	64	28	4	1.56	25	8	3,584	256	128	28	4	3.12	25

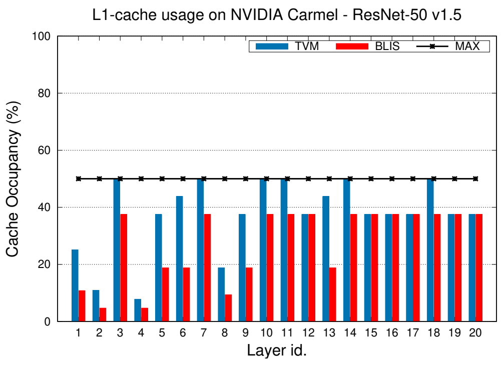
of the L1 cache by B_r, for different micro-kernel dimensions, determined using the analytical model in [21][5]. When executed on the Carmel processor, the analytical model indicates that, for the $m_r \times n_r = 8 \times 12$ micro-kernel that is integrated BLIS for that particular architecture, the micro-panel B_r targeting the L1 cache only occupies 4.69% of the L1 cache for layer 2; and 9.38% for layer 8. In contrast, this micro-kernel should have occupied up to 50% with B_r for both layers, reserving the rest of the L1 cache for entries from the A, C operands.

The only way to address this under-utilization of the L1 cache is by increasing n_r, and hence developing a new micro-kernel. Let us support this observation with a specific example. Consider a micro-kernel of dimension $m_r \times n_r = 4 \times 28$. In this case, the occupancy of the L1 cache by B_r grows to 10.90% for layer 2 and up to 21.90% for layer 8, which is clearly superior to that observed for the (BLIS) 8×12 micro-kernel. In summary, there is a clear theoretical benefit from adopting an $m_r \times n_r = 4 \times 28$ micro-kernel for these particular layers, which is conformal with the experimental advantage that was reported in Figure 16.

Figure 17 reports the utilization rates of the L1 and L2 cache levels by B_r and A_c, respectively, and all the layers. The results show that, especially for the L1 cache, the occupation compared with BLIS explains the overall differences in performance between the BLIS micro-kernel and the optimal alternative automatically generated with TVM reported in Figure 16. For reference, the figure includes the theoretical maximum occupation rate for each cache memory (black line) for the corresponding operands, dictated by the analytical model.

To wrap up this discussion, the under-utilization of the L1 and L2 cache levels resulting from the small values of k and m in non-square problems forces the developer to trade it off with larger register block dimensions (m_r and/or n_r), and hence with the development a family of micro-kernels that, in practice, are invoked intelligently depending on the problem dimensions. Using a tool like TVM to automatically generate micro-kernels, together with an analytical model for cache and register blocking parameters, alleviates this programmability burden not only to obtain dimension-agnostic performance optimization on a single architecture, but also across different architectures.

[5]In the table, we limit the study to those values of m_r and n_r that do not cause register spilling, as this effect would yield a performance penalty beyond that introduced by the negative effect of L1 under-occupation.



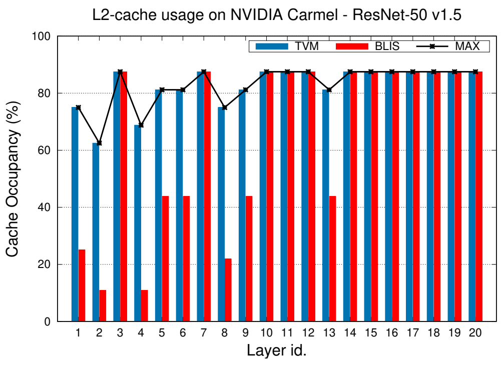


Fig. 17. Occupation for the L1 and L2 caches (left and right, resp.) on the Carmel platform for the best TVM-generated micro-kernel compared with the BLIS micro-kernel.

8.4.3 Optimization of the micro-kernel. To close this discussion on the impact of the micro-kernel, we note that, when searching for the optimal dimension of the micro-kernel, the exploratory space is basically bi-dimensional, as we only need to select the values for m_r, n_r. Now, the optimal value for at least one of these two parameters is tied to the architecture lane size and, in addition, the hardware imposes strict limits on the number of vector registers that can be used from inside the micro-kernel, which in turn constrains the practical values for m_r, n_r. As a consequence, selecting the best option for the TVM-generated routines only requires a few dozens of experiments, and the whole process can be also automatized, providing significant benefits in terms of reduced programming and optimization efforts for the library developer.

In the next subsections *we examine the programming benefits of the TVM-based approach, from the viewpoints of performance, maintainability and portability,* by exposing how TVM allows us to easily generate routines for different data types, explore distinct packing schemes, evaluate alternative parallelization options and, finally, instantiate the full family of matrix multiplication algorithms. Finally, we close the experimental section with an evaluation on AMD and Intel architectures.

8.5 Maintainability: Generating codes for different data types

Generating a routine for a specific data type with TVM only requires adjusting the `dtype` argument in the TVM generator, and modifying accordingly the lane size (see subsection 8.4.1). Compared with this, producing manually a GotoBLAS-2 like routine, for a particular data type, requires a careful re-design of the micro-kernel, usually in assembly, as well as the adaptation of the packing functions.

In Figure 18 we report the performance of the routines generated with TVM for five data types: FP16 (IEEE 16-bit floating point), FP32 (IEEE 32-bit floating point), FP64 (IEEE 64-bit floating point), INT16 (integer 16-bit), and INT32 (integer 32-bit). This figure shows acceleration factors which are conformal with the use of other precision formats.

8.6 Performance: Packing costs

For some problem dimensions, it may be beneficial to skip the packing of any of the matrix operands, or both of them, into the buffers A_c, B_c. As described in subsection 5.3, eliminating the packing scheme is straight-forward with TVM. In contrast, introducing this modification into a conventional GotoBLAS2-like routine implies rewriting the micro-kernel, as this piece of code assumes that the

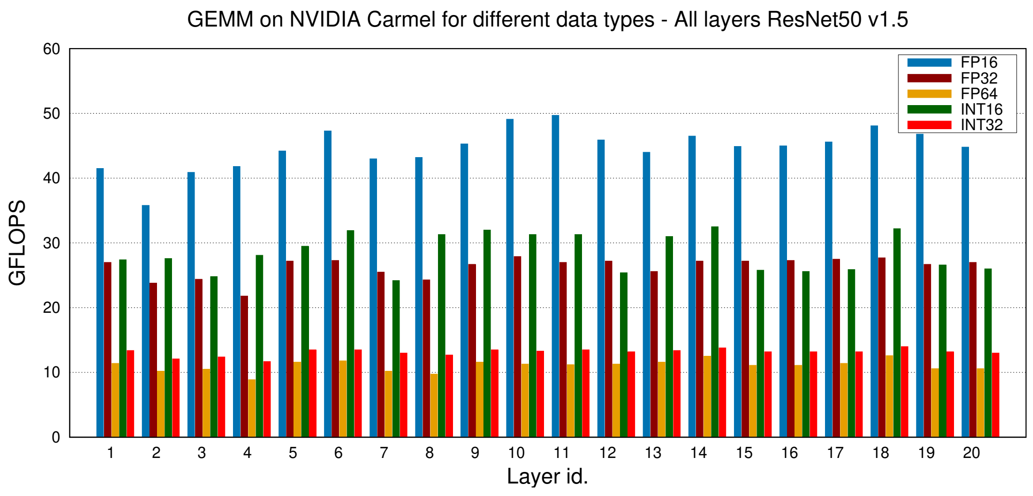


Fig. 18. Performance of the TVM generator for distinct data types on a single NVIDIA Carmel core for ResNet-50 v1.5.

matrix operands are disposed/packed into the buffers in a certain manner; see Figure 4. Given that the micro-kernel is in general encoded in assembly, this is a non-trivial task.

Figure 19 evaluates the packing possibilities using TVM to generate modified versions of the baseline algorithm. As shown in the top chart, for small square matrices, the cost of the re-arranging the data due to packing is not compensated with a "sufficient" acceleration of the micro-kernel. For the problems associated with the convolutional layers in ResNet-50 v1.5 model, the dimensions are large enough and this effect is not present. Nonetheless, the irregular sizes of these operands dictate that, for layers 1, 2, 5, 7 and 18, the performance of the baseline algorithm generated with TVM and a variant which packs only one of the matrix operands are close.

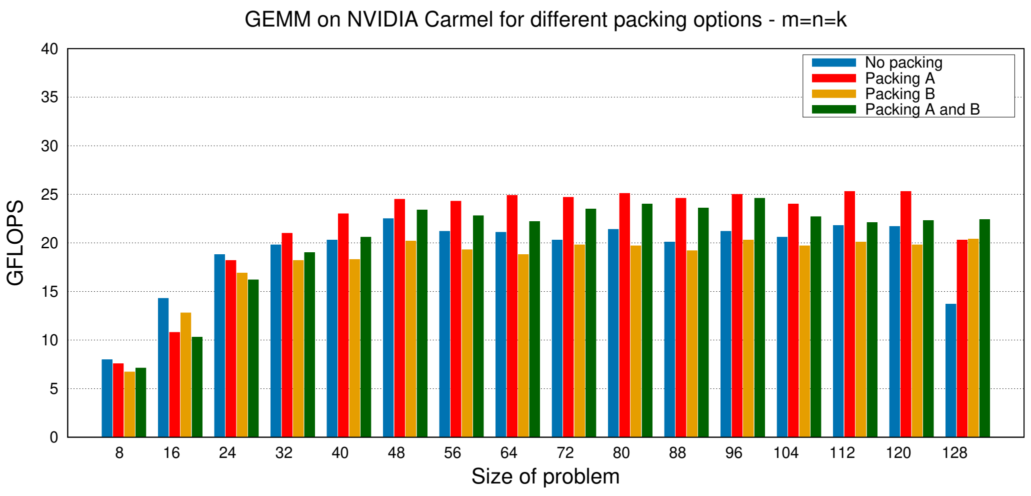
8.7 Performance: Parallelization options

The TVM generator can be leveraged to assess distinct options to orchestrate a multi-threaded execution on a multicore processor. In Figure 20, we target the 8 cores in the NVIDIA Carmel processor, evaluating four parallelization options for the baseline algorithm (see Figure 1, left) that differ in the loop which is parallelized: j_c, i_c, j_r, or i_r. (Loop p_c cannot be parallelized as this would yield a race condition.) This shows that the best choice varies slightly between two of the options, depending on the problem dimensions. However, investigating the best parallelization option adds a dimension to the complexity to the optimization effort that is out-of-scope for this work.

At this point we recognize that an analysis of the parallelization options for a conventional GotoBLAS-2-like routine is also straight-forward. Furthermore, in some cases, it may be more convenient to parallelize multiple loops to expose sufficient thread-level parallelism [31]. Unfortunately, when instructed to parallelize two or more loops, currently TVM extracts parallelism from the outermost loop only.

8.8 Performance, maintainability and portability: The complete family of algorithms

In Section 7, we argued that producing code with TVM for other blocked algorithmic variants of the GEMM family only requires small changes into the generator routines. In this subsection we analyze the practical impact of this on the sequential and parallel performance.



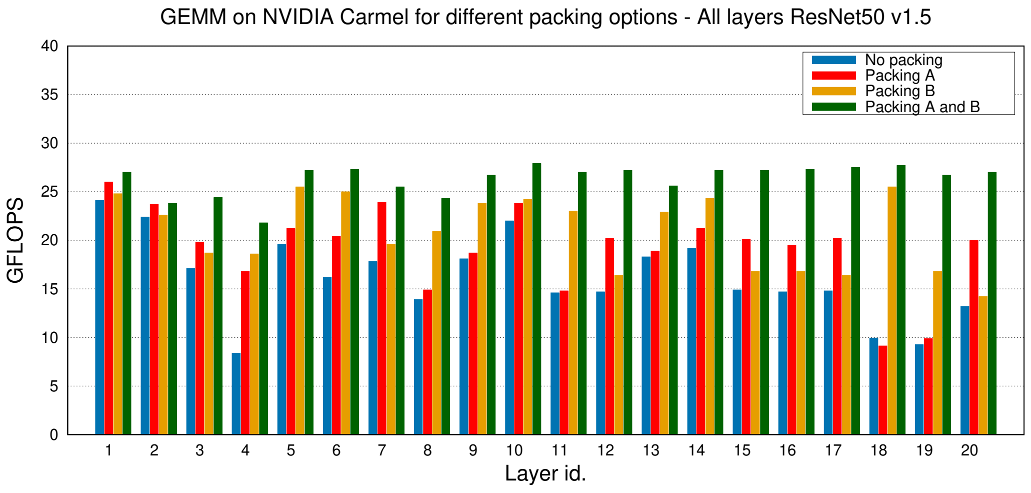


Fig. 19. Performance of the TVM generator for four distinct packing configurations using a single NVIDIA Carmel core for small-square matrices and ResNet-50 v1.5 (top and bottom, respectively).

Figure 21 shows a clear superiority of the variants that maintain C in the processor registers over their counterparts that operate with B in the registers. (We omit the variants with Resident A because they present a symmetric role with the respect to the variants with Resident B.) The reason is that, in the variants with Resident C (i.e., B3A2C0 and A3B2C0), the elements of C are housed in the processor registers during the full execution of the micro-kernel loop and, therefore, there are no writes to memory as part of its innermost loop (L6) of the algorithm. In contrast, the two other variants, A3C2B0 and C3A2B0, integrate a micro-kernel that, at each iteration of the innermost loop (L6), performs several writes on C while this operand resides in a certain level of the cache hierarchy. This behavior lies at the core of the algorithms/variants, and is preserved by TVM which simply follows the programmer's directives. With respect to the results, we observe small differences in performance between the two versions that maintain C in the processor registers, in favor of either one or another depending on the specific layer.

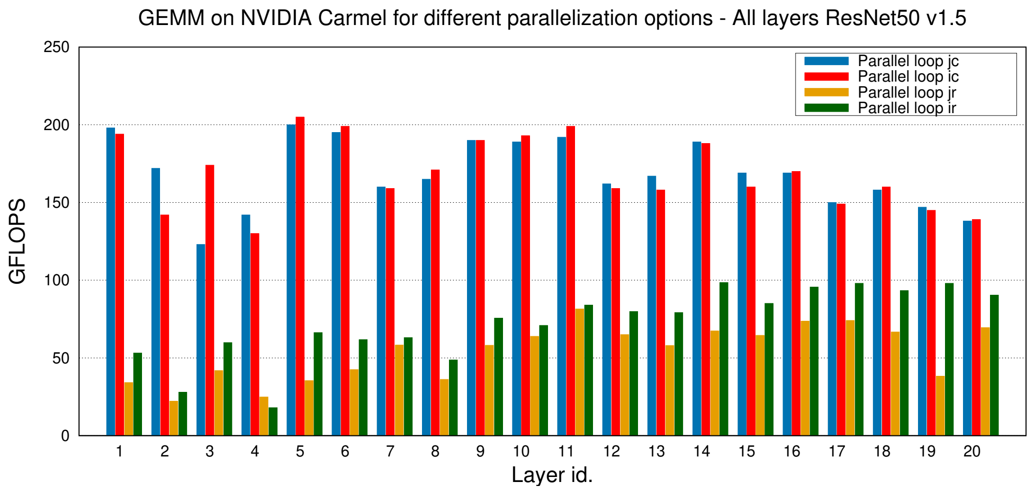


Fig. 20. Performance of the TVM generator for four distinct loop parallelization options on 8 NVIDIA Carmel cores for ResNet-50 v1.5.

Table 3. Size (in bytes) of the GEMM realization for each solution.

Solution	GEMM
ARMPL	29,145,688
BLIS	1,350,456
OpenBLAS	90,944
TVM	532,976

8.9 Memory performance: footprint

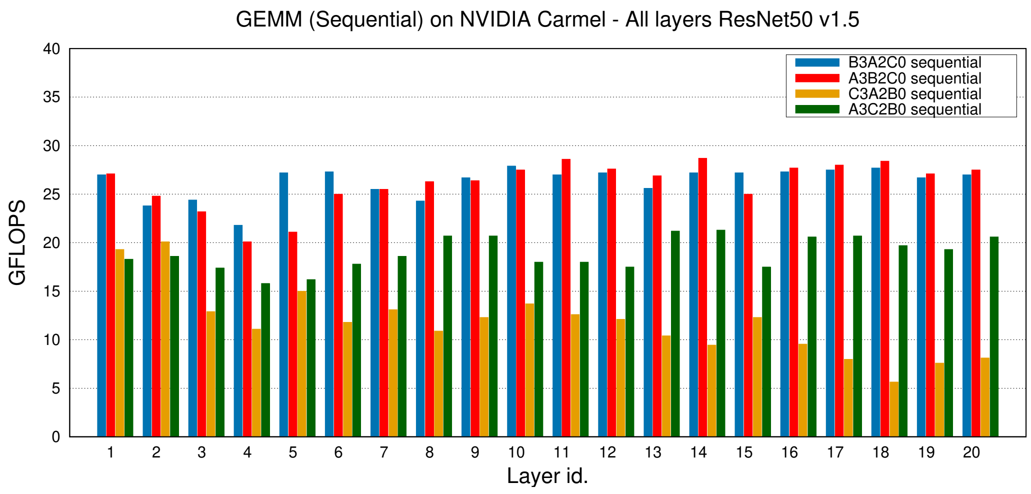
We next investigate the memory requirements of the automatically-generated codes for GEMM. Table 3 reports the size for a bare GEMM test driver routine statically linked with each library (column labeled as "GEMM"). The memory allocation for the matrices and the necessary packing buffers are not included, as their space requirements should be similar in all cases. Furthermore, the test driver is the same in all cases to allow a fair comparison.

The lowest memory footprint for the GEMM executable is offered by OpenBLAS with close to 89 KiB, followed by TVM with 520 KiB. BLIS and ARMPL need a total amount of 1.3 MiB and 27.8 MiB, respectively.

8.10 Portability: Experiments in other architectures

We close the experimental section by demonstrating the performance portability of the automatic-generation approach for GEMM using a pair of AMD and Intel processor architectures. For this purpose, we compare the (sequential) routine generated by TVM, combined with the best micro-kernel, against that of the realization of this kernel in BLIS (version 0.9.0) on an AMD EPYC 7282 (Rome) processor[6]; and the tuned implementation of the same kernel in Intel MKL (version 2021.0) on an Intel Xeon Platinum 8358 (Icelake) processor. In order to generate code specifically tuned for these two architectures, we only had to set the appropriate target in Part P8 of the TVM

[6]We note that the realization of GEMM in AMD's native library (AOCL) is basically BLIS in disguise; see https://developer.amd.com/amd-aocl/blas-library/.



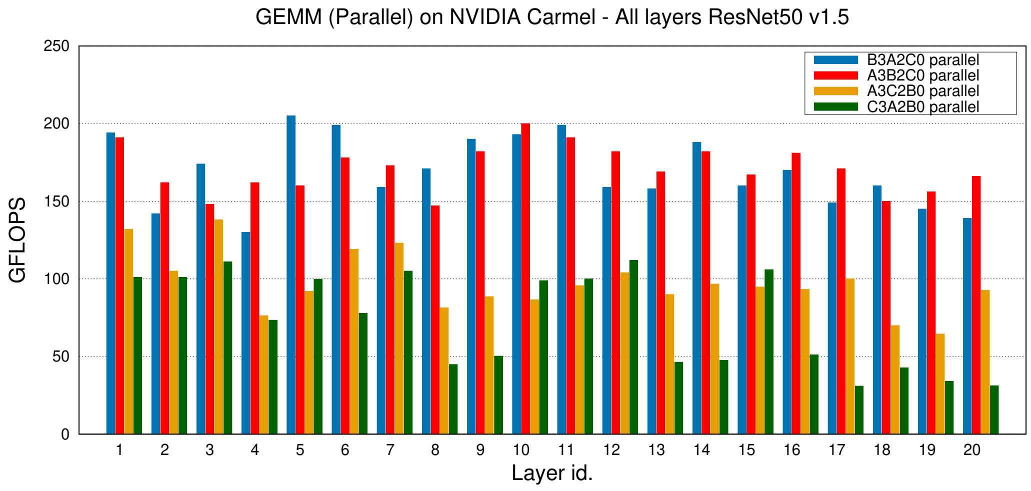


Fig. 21. Performance of the TVM generators for four variants of the GEMM family of algorithms on a single core and 8 cores of the NVIDIA Carmel processor (top and bottom, respectively), for ResNet-50 v1.5. Note the different scales of the y axis in the two plots.

generator. Concretely, lines 35–38 in Figure 5 specify the backends selected for these two processor architectures.

Figure 22 reports the results for this final experiment using the layers in the ResNet50 model as testbed. The two charts there show a similar trend, which was already present in the results for the NVIDIA Carmel core in Figure 16. Concretely, the TVM routine outperforms the library realizations for the "rectangular" cases, but it is suboptimal for "square" problems. In order to investigate this behavior in more detail, given that the library evaluated in the case of the AMD architecture is BLIS, we inspected the internals of the micro-kernel integrated in the library for that particular processor, in order to compare it with the micro-kernel generated by TVM.

Note that the BLIS and TVM solutions rely on the baseline algorithm for GEMM and, therefore, on a micro-kernel with Resident C. Specifically, for the AMD Rome, BLIS hand-encodes a micro-kernel of dimension $m_r \times n_r = 6 \times 16$, unrolling loop L6 by a factor of 4. Given that the AMD Rome

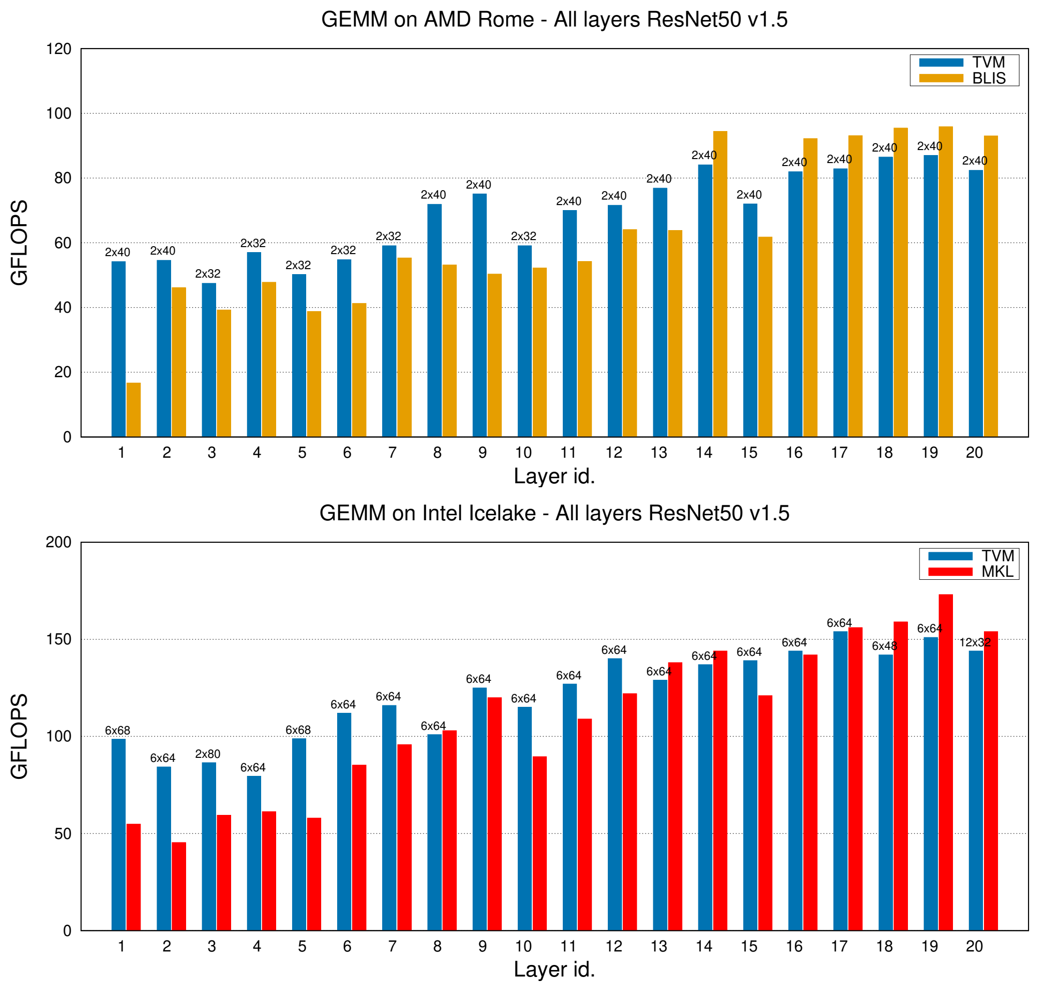


Fig. 22. Performance of the TVM generator for ResNet-50 v1.5 for AMD and Intel (top and bottom, respectively).

features 16 vector registers, and that the SIMD width is 256 bits (that is, 8 FP32 per vector register) for AVX2, the BLIS micro-kernel dedicates $6 \times 2 = 12$ vector registers to maintain the micro-tile of C. Furthermore, at each iteration of loop L6, the micro-kernel utilizes two vector registers to load a row of the micro-panel B_r (of dimension $k_c \times 16$ micro-panel) plus and single vector register to broadcast one-by-one the six entries of a column of the micro-panel A_r (of dimension $6 \times k_c$) prior to operating with each. In total, the micro-kernel thus occupies 15 out of the 16 available vector registers. In addition, the code for the micro-kernel features a notable number of (assembly) prefetching instructions.

With the above-described configuration for the micro-kernel, the realization of GEMM in BLIS delivers 95.3 GFLOPS for square GEMM problems of dimension $m = n = k$ =2,000 while, by disabling the prefetching instructions, the performance drops to 78.5 GFLOPS. Compared with that, the best micro-kernel generated with TVM for that problem dimension corresponds to $m_r \times n_r = 2 \times 40$, which delivers 88.2 GFLOPS. The flop throughtput rate for TVM is thus somewhere between the two rates for BLIS (i.e., with and without prefetching) which, on the one hand, is not totally surprising as TVM cannot exploit (assembly-level) prefetching instructions. On the other hand, the dimensions of the best micro-kernel selected by TVM are quite surprising. To further investigate this, we instructed TVM to generate a GEMM routine using a BLIS-like micro-kernel, that is, with $m_r \times n_r = 6 \times 16$ and an unrolling factor of 4. Interestingly, the TVM routine that integrated this micro-kernel reported 39.60 GFLOPS only. The reason is that, for this particular micro-kernel dimension and unrolling factor, TVM produced a micro-kernel that did not maintain the micro-tile of C into the processor registers, incurring into register spilling during the execution of the micro-kernel loop and considerably degrading performance! Whether we can enforce TVM to avoid this effect is currently under investigation as, to a good extent, it depends on internal behavior of TVM.

9 CONCLUDING REMARKS

We have presented an integral TVM-based solution to automatically obtain high performance realizations of GEMM. On the one hand, our solution departs from conventional library-based realizations of GEMM in that the full code is automatically generated, including both the blocked routines for the family of GEMM algorithms extending the ideas of GotoBLAS2 as well as the processor-specific micro-kernels. On the other hand, compared with other JIT compilation frameworks, we mimic the techniques in the GotoBLAS2/BLIS/OpenBLAS2 algorithms for GEMM to obtain blocked algorithms that attain an efficient utilization of the cache memories, considerably trimming the cost of exploring the optimization search space. TVM can also generate competitive code for GPUs that perform close to NVIDIA cuBLAS library. However, the schedule for the GPU-specific GEMM differs from that in the CPU version described in this work. Explaining the differences out of scope for this work.

Our work exposes the programming advantages, from the points of view of performance, maintainability, and portability, of the TVM-automatized approach, which can be leveraged, among others, to seamlessly generate routines for different data types, explore distinct packing schemes, evaluate alternative parallelization options, and instantiate the entire family of matrix multiplication algorithms.

ACKNOWLEDGMENTS

This work was supported by the research projects PID2020-113656RB-C22 (MCIN/AEI/10.13039/501100011033), and PID2021-126576NB-I00; and CM via Multiannual Agreement with Complutense University in the line Program to Stimulate Research for Young Doctors in the context of the V PRICIT under projects PR65/19-22445 and CM S2018/TCS-4423. A. Castelló is a FJC2019-039222-I fellow supported by MCIN/AEI/10.13039/ 501100011033. H. Martínez is a postdoctoral fellow supported by the *Consejería de Transformación Económica, Industria, Conocimiento y Universidades de la Junta de Andalucía*. This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558 (eFlows4HPC project). The JU receives support from the European Union's Horizon 2020 research and innovation programme, and Spain, Germany, France, Italy, Poland, Switzerland, and Norway.

REFERENCES

[1] Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V. Le. 2017. Neural Optimizer Search with Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 459–468. http://proceedings.mlr.press/v70/bello17a.html

[2] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis. *ACM Comput. Surv.* 52, 4, Article 65 (Aug. 2019), 43 pages. https://doi.org/10.1145/3320060

[3] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. *J. Mach. Learn. Res.* 13, 1 (feb 2012), 281–305.

[4] Uday Bondhugula. 2020. High Performance Code Generation in MLIR: An Early Case Study with GEMM. *CoRR* abs/2003.00532 (2020). arXiv:2003.00532 https://arxiv.org/abs/2003.00532

[5] A. Castelló, F. D. Igual, and E. S. Quintana-Ortí. 2022. Anatomy of the BLIS family of algorithms for matrix multiplication. In *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 92–99. https://doi.org/10.1109/PDP55904.2022.00023

[6] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High Performance convolutional neural networks for document processing. In *International Workshop on Frontiers in Handwriting Recognition*.

[7] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi (Eds.). ACM, 785–794. https://doi.org/10.1145/2939672.2939785

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018). arXiv:1802.04799 http://arxiv.org/abs/1802.04799

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[11] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. *CoRR* abs/1805.08166 (2018). arXiv:1805.08166 http://arxiv.org/abs/1805.08166

[12] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759

[13] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 1 (2001), 3–35. https://doi.org/10.1016/S0167-8191(00)00087-9 New Trends in High Performance Computing.

[14] Kevin Dowd and Charles R. Severance. 1998. *High Performance Computing* (2nd ed.). O'Reilly.

[15] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of a High-Performance Matrix Multiplication. *ACM Trans. on Mathematical Software* 34, 3 (May 2008), 12:1–12:25.

[16] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. 2004. A Family of High-Performance Matrix Multiplication Algorithms. In *Proc. 7th Int. Conf. on Applied Parallel Computing: State of the Art in Scientific Computing* (Lyngby, Denmark) *(PARA'04)*. 256–265.

[17] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabás Póczos, and Eric P. Xing. 2018. Neural Architecture Search with Bayesian Optimisation and Optimal Transport. *CoRR* abs/1802.07191 (2018). arXiv:1802.07191 http://arxiv.org/abs/1802.07191

[18] Duseok Kang, Euiseok Kim, Inpyo Bae, Bernhard Egger, and Soonhoi Ha. 2018. C-GOOD: C-Code Generation Framework for Optimized on-Device Deep Learning. In *Proceedings of the International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '18)*. Association for Computing Machinery, New York, NY, USA, Article 105, 8 pages. https://doi.org/10.1145/3240765.3240786

[19] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. *CoRR* abs/2002.11054 (2020). arXiv:2002.11054 https://arxiv.org/abs/2002.11054

[20] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, and Depei Qian. 2020. The Deep Learning Compiler: A Comprehensive Survey. *CoRR* abs/2002.03794 (2020). arXiv:2002.03794 https://arxiv.org/abs/2002.03794

[21] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Ortí. 2016. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Trans. on Mathematical Software* 43, 2, Article 12 (Aug. 2016), 18 pages.

[22] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*

(Proceedings of Machine Learning Research, Vol. 70), Doina Precup and Yee Whye Teh (Eds.). PMLR, 2430–2439. http://proceedings.mlr.press/v70/mirhoseini17a.html

[23] Thierry Moreau, Tianqi Chen, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. VTA: An Open Hardware-Software Stack for Deep Learning. *CoRR* abs/1807.04188 (2018). arXiv:1807.04188 http://arxiv.org/abs/1807.04188

[24] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[25] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. 2017. Searching for Activation Functions. *CoRR* abs/1710.05941 (2017). arXiv:1710.05941 http://arxiv.org/abs/1710.05941

[26] Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.

[27] Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. 2014. Anatomy of High-Performance Many-Threaded Matrix Multiplication. In *Proc. IEEE 28th Int. Parallel and Distributed Processing Symp. (IPDPS'14)*. 1049–1059.

[28] Tyler M. Smith and Robert A. van de Geijn. 2019. The MOMMS Family of Matrix Multiplication Algorithms. *CoRR* abs/1904.05717 (2019). arXiv:1904.05717 http://arxiv.org/abs/1904.05717

[29] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* 105, 12 (Dec 2017), 2295–2329. https://doi.org/10.1109/JPROC.2017.2761740

[30] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. on Mathematical Software* 41, 3 (2015), 14:1–14:33.

[31] Field G. Van Zee and Robert A. van de Geijn. 2016. The BLIS Framework: Experiments in Portability. *ACM Trans. on Mathematical Software* 42, 2, Article 12 (June 2016), 19 pages.

[32] Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, and Jürgen Schmidhuber. 2011. Natural Evolution Strategies. arXiv:1106.4487 [stat.ML]

[33] Zhang Xianyi, Wang Qian, and Zhang Yunquan. 2012. Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*.

[34] Kamen Yotov, Xiaoming Li, María Jesús Garzarán, David Padua, Keshav Pingali, and Paul Stodghill. 2005. Is Search Really Necessary to Generate High-Performance BLAS? *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"* 93, 2 (2005).

[35] Huaqing Zhang, Xiaolin Cheng, Hui Zang, and Dae Hoon Park. 2019. Compiler-Level Matrix Multiplication Optimization for Deep Learning. *CoRR* abs/1909.10616 (2019). arXiv:1909.10616 http://arxiv.org/abs/1909.10616

[36] Y. Zhang. 2009. *Parallel solution of integral equation-based EM problems in the frequency domain.* IEEE Press.

[37] Barret Zoph and Quoc V. Le. 2016. Neural Architecture Search with Reinforcement Learning. *CoRR* abs/1611.01578 (2016). arXiv:1611.01578 http://arxiv.org/abs/1611.01578