The Journal of Supercomputing (2025) 81:214
https://doi.org/10.1007/s11227-024-06720-7

Experience-guided, mixed-precision matrix multiplication with apache TVM for ARM processors

Adrián Castelló[1] · Héctor Martínez[2] · Sandra Catalán[3] · Francisco D. Igual[4] · Enrique S. Quintana-Ortí[1]

Accepted: 12 November 2024
© The Author(s) 2024

Abstract

Deep learning (DL) generates new computational tasks that are different from those encountered in classical scientific applications. In particular, DL training and inference require general matrix multiplications (GEMM) with matrix operands that are far from large and square as in other scientific fields. In addition, DL models gain arithmetic/storage complexity, and as a result, reduced precision via quantization is now mainstream for inferring DL models in edge devices. Automatic code generation addresses these new types of GEMM by (1) improving portability between different hardware with only one base code; (2) supporting mixed and reduced precision; and (3) enabling auto-tuning methods that, given a base operation, perform a (costly) optimization search for the best schedule. In this paper, we rely on Apache TVM to generate an experience-guided GEMM that provides performance competitive with the TVM auto-scheduler, while reducing tuning time by a factor of 48×.

Keywords Matrix multiplication · Apache TVM · Auto-scheduling · ARM NEON · High performance

✉ Adrián Castelló
adcastel@disca.upv.es

Héctor Martínez
el2mapeh@uco.es

Sandra Catalán
catalans@uji.es

Francisco D. Igual
figual@ucm.es

Enrique S. Quintana-Ortí
quintana@disca.upv.es

[1] Universitat Politècnica de València, València, Spain

[2] Universidad de Córdoba, Córdoba, Spain

[3] Universitat Jaume I de Castelló, Castelló de la Plana, Spain

[4] Universidad Complutense de Madrid, Madrid, Spain

Published online: 26 November 2024

1 Introduction

The thirst to apply artificial intelligence (AI) to almost every scientific problem has fueled the proliferation of deep neural networks (DNNs), which are now running on a wide variety of computing devices, from powerful server nodes to resource-constrained Internet of Things (IoT) devices. In this scenario, inferring complex deep learning (DL) models involves costly computations and consumes large amounts of memory. While these requirements are rarely problematic for server nodes, in the case of power- and memory-constrained devices, they are typically addressed through compression techniques such as quantization [1].

General matrix multiplication (GEMM) is a flagship operation for linear algebra (LA) problems and scientific applications. This kernel is also critical for DL models that use transformers for natural language processing [2] as well as convolutional deep neural networks (CNNs) for signal processing and computer vision tasks [3, 4].

Well-known LA libraries, such as Intel MKL, AMD AOCL, ARM PL, NVIDIA cuBLAS, GotoBLAS2 [5], OpenBLAS [6], BLIS [7], oneDNN, and ATLAS [8], provide high-performance instances of GEMM. In general, these libraries implement an architecture-agnostic blocked algorithm for GEMM that embeds a small, hardware-specific piece of code known as a micro-kernel. To reduce development effort, most of these libraries provide a single, highly optimized micro-kernel per processor architecture, resulting in a solution that sacrifices arithmetic throughput for productivity.

The performance of GEMM is highly dependent on the problem dimensions of the matrix operands. LA libraries address this limitation by providing specialized code paths to handle the most common cases in scientific computing, but still limit the number of different micro-kernels to one per processor architecture. In comparison, the types of GEMMs that arise in DL present some special dimensions that differ from those typically found in scientific computing, and benefit from a customized design of the micro-kernel. In addition, DL inference with quantized models requires new data types, such as 16-bit float, integer, or mixed precision, which are not supported by current LA libraries, with the notable exception of oneDNN.[1]

Recent compiler frameworks such as Halide [9], Apache TVM (Tensor Virtual Machine) [10], Exo [11], JAX [12], TensorFlow XLA [13], and Tiramisu [14] fill this niche by (semi-)automatically generating optimized code. In rough detail, these frameworks take an operation, encoded in a domain-specific language (DSL) or a high-level language such as Python, with some optional hints for the preferred operation plan. To improve code portability and reduce optimization effort, the frameworks then generate an architecture-agnostic intermediate representation (IR) of the operation, which is finally transformed and compiled for the specific target architecture.

TVM is a prominent code generation tool that enjoys a large supporting community of vendors, academics, and individuals who continuously contribute new

[1] https://oneapi-src.github.io/oneDNN/dev_guide_matmul.html#data-type.

hardware backends and functionality. A major reason for this is TVM's auto-scheduling tool, which, given a basic operation definition, explores a variety of optimization approaches (as many as the user specifies) and returns the best-performing solution among them. On the negative side, this mechanism is time-consuming. Moreover, if the performance of the resulting implementation is not satisfactory, the problem dimensions vary, or the data type combinations are different, the auto-tuning has to be repeated, which increases the optimization cost.

In this paper, we present an experience-guided (EG), TVM implementation of GEMM for DL scenarios that significantly reduces the tuning effort while delivering performance on par with that of the best solution produced by an expensive search of the TVM auto-scheduler. Specifically, our work makes the following specific contributions: (1) we overcome the missing functionality in common LA libraries for reduced and mixed-precision implementations of GEMM; (2) we provide a step-by-step guide for auto-scheduling a high-performance implementation of GEMM using TVM; and (3) we demonstrate that our EG GEMM significantly reduces the GEMM tuning cost while maintaining or improving performance on two different ARM platforms.

The rest of the paper is organized as follows: Section 2 provides an overview of the state-of-the-art automatic code generators for high-performance codes. In Sect. 3, we review the high-performance realization of GEMM, highlighting the critical features for achieving the best performance. Section 4 shows how to use TVM scheduling to generate GEMM with reduced and mixed precision. In addition, we focus there on the TVM auto-scheduling method and explain our EG-based approach. In Sect. 5, we evaluate the performance of the resulting solution by comparing it to the TVM auto-scheduling approach on NVIDIA Jetson AGX Xavier and nano boards for different data types. Finally, in Sect. 6, we summarize the main results of this work.

2 Related work

Automatic code generation has gained interest in recent years as a way to achieve performance portability across architectures with minimal programmer intervention [15]. Languages and compiler frameworks such as Halide [9], Exo [11], and Apache TVM [10] propose a dual approach that decouples the definition of the operation from its scheduling (or optimization). This simplifies the development and enhances the portability of operators across target architectures [16], since once the operator is defined, optimization techniques such as operator fusions and transformations combined with hardware-specific optimizations are applied.

From a technical point of view, automatic generation frameworks can be broadly classified as JIT (*Just-in-time*) or AOT (*Ahead-of-time*). JIT compilers generate executable code on the fly. Thus, they can use extended runtime knowledge to fine-tune the final executable at the cost of some overhead due to code generation. Examples

of JIT compilers are TFLite ,[2] XLA, MLIR [17], and TVM. In contrast, AOT compilers generate executable code a priori and execute common versions at runtime without further modification. The advantage of AOT is that it facilitates the development of cross-compilation schemes, such as Exo, for embedded architectures [18] or remote execution-only architectures [19].

Unfortunately, fine-tuning in both JIT and AOT approaches is an unpredictable, time-consuming process because it involves exploring and measuring the performance of a wide variety of search spaces. Scheduling time has also been optimized, and a clear example is Ansor (or auto-scheduling) [20], the TVM auto-scheduling method that replaces AutoTVM [21]. This new scheduler supersedes template-based tuning with a more sophisticated search algorithm, reducing the search space and thus the tuning time.

Our approach combines the advantages of existing JIT compiler frameworks, such as TVM, to easily derive high-performance codes for GEMM-based DL primitives, with analytical models to avoid expensive auto-tuning.

Our work differs from [22], which uses MLIR to describe early experiences exclusively with GEMM; from [23], which uses Exo to generate small hardware-oriented pieces of code; and from [24], which proposes advanced auto-tuning schemes for this primitive. This work also differs from our previous work [25], in which we develop an entire family of GEMM algorithms. All of them employ 32- and 64-bit floating-point matrix operands, while our focus now is the more challenging mixed-precision case. Specifically, we guide TVM to generate mixed- and/or reduced-precision GEMMs with minimal tuning time overhead, and compare the process and the generated code with TVM's auto-scheduling.

3 Baseline implementation of GEMM

Consider hereafter the GEMM $C = C + AB$, where A, B, and C are matrices of dimensions $m \times k$, $k \times n$, and $m \times n$, respectively. Modern high-performance instances of this operation, for conventional processor architectures with deep memory hierarchies, follow GotoBLAS [5] to encode it as five nested loops, two packing routines, and a micro-kernel. Furthermore, for processors with SIMD vector units, the latter consists of an additional loop that performs an outer product per iteration. Figure 1 (top) displays the baseline algorithm for the blocked GEMM, consisting of the six loops, the two packing routines, and the micro-kernel. Portable realizations of GEMM encode the five outermost loops and the two packing routines in a high-level programming language such as C. In contrast, for high performance, the architecture-specific micro-kernel is usually encoded in assembly or in C with vector intrinsics.

The three outermost loops of the baseline algorithm for GEMM partition the matrix operands conformal to the processor cache hierarchy. This specific nesting of the loops, together with a proper packing of A, B (see the bottom right plot in Fig. 1) plus a careful selection of the cache configuration parameters (CCPs) m_c, n_c, k_c [26],

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

```
for (jc=0; jc<n; jc+=nc)                    // Loop L1
  for (pc=0; pc<k; pc+=kc) {                // Loop L2
    // Pack B
    Bc := B(pc:pc+kc-1,jc:jc+nc-1);
    for (ic=0; ic<m; ic+=mc) {              // Loop L3
      // Pack A
      Ac := A(ic:ic+mc-1,pc:pc+kc-1);
      for (jr=0; jr<nc; jr+=nr)             // Loop L4
        for (ir=0; ir<mc; ir+=mr)           // Loop L5
          // Micro-kernel
          for (kr=0; kr<kc; kr++)           // Loop L6
            C(ic+ir:ic+ir+mr-1,jc+jr:jc+jr+nr-1)
              += Ac(ir:ir+mr-1,kr) * Bc(kr,jr:jr+nr-1);
  }}
```

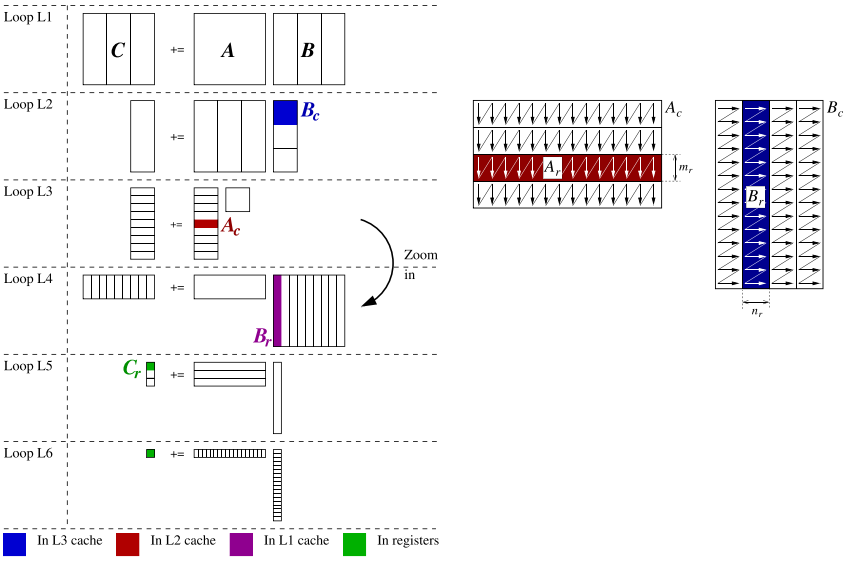


Fig. 1 Baseline algorithm for GEMM (top), data transfers across the memory hierarchy (bottom left), and packing (bottom right)

favor that, during the execution of the micro-kernel, the buffers A_c, B_c respectively remain in the L2 and L3 cache memories. Inside the micro-kernel, the code streams an $m_r \times n_r$ micro-tile C_r of C from the main memory into the processor registers; an $m_r \times k_c$ micro-panel A_r of A_c from the L2 cache; and a $k_c \times n_r$ micro-panel B_r of B_c from the L1 cache; see the bottom left plot in Fig. 1.

4 Implementing GEMM with TVM

4.1 Scheduling in TVM

Apache TVM is a compiler framework that provides developers with a tool to program, optimize, and execute computations on a variety of hardware backends. TVM

can be used to generate optimized code for an entire DL model (including isolated operator optimizations or general optimizations such as layer fusion, operator fusion, or layout transformations) or for a specific operator within a model.

Typically, the TVM workflow starts with a model developed in a high-level framework such as Tensorflow, PyTorch, or ONNX. From this input, TVM performs a series of successive transformations that reduce the abstraction level of the model prior to final code generation, namely:

Stage 1 (Translation to Relay). Relay is the common high-level language used by TVM to describe and unify models imported from high-level frameworks, regardless of their source and format. After applying high-level optimizations to the relay representation, the framework partitions the model into subgraphs and proceeds to the next transformation step.

Stage 2 (Lowering to TE). Lowering is the process of transforming a high-level representation of a computation to a lower level, closer to the hardware representation. TE (*Tensor Expression*) is a DSL for tensor computations. Optionally, TE can provide mechanisms for specifying low-level optimizations such as *unrolling*, *tiling*, *vectorization*, *parallelization*, or *loop fusion*. Note that manual development of operators (such as GEMM) can start at this stage, as is our case for the EG approach.

Stage 3 (Schedule Optimization). A *schedule* is the mechanism proposed by TVM to specify the loop-level optimizations for a given operator or subgraph within a model defined via TE. TVM builds a schedule by incrementally applying basic transformations (schedule primitives) that preserve the logical equivalence of the program. There are two strategies for selecting these primitives in TVM:

1. *Automatic*. Auto-tuning modules in TVM proceed by inspecting the scheduling space, using cost models or actual time measurements as a reference for selection. Currently, there are two different mechanisms for auto-tuning in TVM:
   - AutoTVM [21]. Based on *schedule templates*, AutoTVM requires the developer to write extensive code templates that define the degrees of freedom (in

terms of schedules) that the optimizer needs to explore during the search process. These templates are not only verbose but also require a deep understanding of the potential low-level optimizations and thus the underlying hardware.
- Auto-scheduler [20] does not require the explicit definition of scheduling templates, but only a high-level description of the tensor operation. Armed with a more sophisticated search algorithm, this allows the exploration of more optimization combinations at the cost of additional exploration time.

2. *Manual.* As said, TE allows not only the definition of different low-level optimizations but also their order of application in the general schedule [10]. This gives the developer full control over the optimization path and avoids the auto-tuning cost, but requires a deep understanding of the underlying hardware and potential optimizations.

STAGE 4 (LOWERING TO TENSOR IR). Tensor IR is the low-level TVM intermediate representation into which the optimal schedule of operations is translated and low-level optimizations are applied before the representation is lowered to the underlying compiler (e.g., LLVM) for code generation.

STAGE 5 (TARGET ARCHITECTURE COMPILATION) This stage applies compiler and architecture-specific optimization flags.

In the following, we focus on STAGE 3, and the potential of the auto-scheduler and the manual (expert-guided) approach to GEMM generation.

4.2 TVM auto-scheduling

In this subsection, we describe the process to optimize GEMM by leveraging the TVM auto-scheduler. Concretely, we first present the operation definition and then explain how to set up the scheduling.

Figure 2 defines a basic GEMM realization using TVM TEs. Line 1 indicates that the function is suitable for auto-scheduling and line 2 declares the dimensions and data types of the matrix operands. Lines 5–6 associate these dimensions and data types with the operands A, B. Line 9 specifies the "reduction axis" for the computation. Lines 12–16 specify how to compute the output from the GEMM inputs, including castings (in case they are necessary) from the original data types for A, B to the destination data type for C. The parameter `attrs` in line 15 informs the auto-scheduler that it is permitted to transform the tensor B to a more suitable form for

```
@auto_scheduler.register_workload
def gemm(m, n, k, dtypeA, dtypeB, dtypeC):

    # Definition of buffers
    A = te.placeholder((m, k), name="A", dtype=dtypeA)
    B = te.placeholder((k, n), name="B", dtype=dtypeB)

    # Reduction axis declaration
    p = te.reduce_axis((0, k), name="p")

    # Computation of GEMM
    C = te.compute((m, n),
      lambda i, j: te.sum(
        (A[i, p].astype(dtypeC) * B[p, j].astype(dtypeC)),
        axis=p), name="C", attrs={"layout_free_placeholders": [B]}
    )

    return [A, B, C]
```

Fig. 2 TVM definition for the basic GEMM with the auto-scheduling annotation and suitable for mixed precision

```
# Define the computation for optimization
task = tvm.auto_scheduler.SearchTask(
           func=gemm, args=(m, n, k, dtypeA, dtypeB, dtypeC),
           target="llvm -device=arm_cpu -mattr=+v8.2a+fp16fml")

# Name of the record file
log_file = "gemm.json"

# Configure the auto-tuning method
tune_option = auto_scheduler.TuningOptions(
          num_measure_trials=trials,
          measure_callbacks=[auto_scheduler.RecordToFile(log_file)])

# Run auto-tuning (search)
task.tune(tune_option)

# Apply the best schedule
sch, args = task.apply_best(log_file)

# Build the optimized code
func = tvm.build(sch, args, target)
```

Fig. 3 TVM auto-schedule configuration for GEMM

the operation. This optimization hint is extracted from the TVM tutorial.[3] In this concrete case, as A and B are stored in row-major order, and the GEMM multiplies one row of A with one column of B, the auto-scheduler will transpose B so that all its elements are encountered in consecutive positions in memory during the computation of C.

[3] https://tvm.apache.org/docs/tutorial/auto_scheduler_matmul_x86.html.

Figure 3 shows the instructions that run the Auto-scheduler. Lines 2–4 create the scheduling task with the GEMM function and arguments, and specify the target hardware: an ARMv8 CPU in this case.

Lines 10–12 configure the search space for the auto-scheduler. There we indicate the number of the schedules to be evaluated and the file where the results will be recorded. In this search, the `early-stopping` argument is not set so that the scheduler inspects all the possibilities (at the expense of a very high tuning cost). Line 15 runs the scheduling optimization, while line 18 identifies the highest performing solution. Finally, line 21 builds the code for the specific target.

Figure 4 reports the best schedule identified by the TVM auto-scheduler when applied to the GEMM, with 32-bit single precision matrix operands, associated with the first layer of the ResNet-50 v1.5 model ($m = 12544, n = 64, k = 147$). The ad hoc auto-generated solution extracts the axes in lines 2–3. Lines 6–12 split these axes, applying block tiling to generate a new loop structure; and lines 14–15 reorder the loops. Finally, lines 18–22 apply some additional optimizations: unroll and parallelize the outermost loop, and vectorize the innermost one.

Using the information from the auto-scheduler, TVM next generates the IR in Fig. 5. Lines 6–8 there allocate a new buffer, of dimension $k \times n = 147 \times 64 = 9408$, for the transposed B operand (as we declare it in the computation definition), and lines 11–16 apply the transform. Lines 19–32 then compute the GEMM operation. Concretely, line 23 initializes the result to zero; line 24 sets a pointer to the A operand, of dimension $m \times k = 12544 \times 64 = 1843968$; and finally lines 26–32 perform the computation. (For brevity, we omit repeated lines in the code.)

Although the auto-scheduler is a very useful tool for generating highly optimized code, its main drawback is the time it takes the framework to find a solution. For example, when instructed to apply 1,000 trials per operation, the TVM

```
1  # Getting GEMM loops
2  C_i, C_j = C.op.axis
3  C_p = C.op.reduce_axis
4
5  # GEMM structure
6  C_i_o_i, C_i_i       = s[C].split(C_i, factor=1)
7  C_i_o_o_i, C_i_o_i   = s[C].split(C_i_o_i, factor=1)
8  C_i_o_o_o, C_i_o_o_i = s[C].split(C_i_o_o_i, factor=1)
9  C_j_o_i, C_j_i       = s[C].split(C_j, factor=64)
10 C_j_o_o_i, C_j_o_i   = s[C].split(C_j_o_i, factor=1)
11 C_j_o_o_o, C_j_o_o_i = s[C].split(C_j_o_o_i, factor=1)
12 C_p_o, C_p_i         = s[C].split(C_p, factor=21)
13
14 s[C].reorder(C_i_o_o_o, C_j_o_o_o, C_i_o_o_i, C_j_o_o_i, C_p_o,
15              C_i_o_i, C_j_o_i, C_p_i, C_i_i, C_j_i)
16
17 # GEMM optimizations
18 C_i_o_o_o = s[C].fuse(C_i_o_o_o)
19 s[C].parallel(C_i_o_o_o)
20 s[C].pragma(C_i_o_o_o, "auto_unroll_max_step", 512)
21 s[C].pragma(C_i_o_o_o, "unroll_explicit", True)
22 s[C].vectorize(C_j_i)
```

Fig. 4 Optimal schedule determined by the TVM auto-scheduling for the GEMM associated with first layer of ResNet-50 v1.5 and FP32 matrices

```
@I.ir_module
@T.prim_func
def gemm(A: T.Buffer((12544, 147), "float32"),
         B: T.Buffer((147, 64), "float32"),
         C: T.Buffer((12544, 64), "float32")):
  auto_layout_transform = T.allocate([9408], "float32")
  auto_layout_transform_1 = T.Buffer((9408,),
                            data=auto_layout_transform)

  # B tensor transform
  for ax0_ax1_fused_ax2_fused in T.parallel(7):
    for ax4, ax5 in T.grid(21, 64):
      var_1: T.int32 =
        ax0_ax1_fused_ax2_fused * 1344 + ax4 * 64 + ax5
      B_1 = T.Buffer((9408,), data=B.data)
      auto_layout_transform_1[var_1] = B_1[var_1]

  # GEMM computation
  for i_o_o_o in T.parallel(12544):
    var_3: T.int32 = i_o_o_o * 64
    var_2: T.int32 = i_o_o_o * 147
    C_1 = T.Buffer((802816,), data=C.data)
    C_1[var_3:var_3 + 64] = T.Broadcast(T.float32(0), 64)
    A_1 = T.Buffer((1843968,), data=A.data)

    C_1[var_3:var_3 + 64] += T.Broadcast(A_1[var_2], 64) *
                             auto_layout_transform_1[0:64]
    C_1[var_3:var_3 + 64] += T.Broadcast(A_1[var_2 + 1], 64) *
                             auto_layout_transform_1[64:128]
    ... # Omitted for brevity
    C_1[var_3:var_3 + 64] += T.Broadcast(A_1[var_2 + 146], 64) *
                             auto_layout_transform_1[9344:9408]
```

Fig. 5 Resulting IR of the TVM auto-scheduling for the first layer of ResNet-50 v1.5 and FP32

auto-scheduler takes 20 min to optimize a single layer of the ResNet-50 v1.5 model, for a single combination of data types, on the NVIDIA Jetson AGX Xavier. Obviously, the tuning time grows linearly with the number of layers and data type combinations. In other words, changing a single dimension of the problem or one of the matrix data types requires repeating the entire process. In addition, it is possible that the solution found by the auto-scheduler is still suboptimal because the number of trials was too small or the best solution was in an unexplored direction.

4.3 Experience-guided scheduling

In this section, we present our EG realization of GEMM using TVM. In contrast with the previous approach, in this case, we generate a specific schedule that mimics the GotoBLAS2 formulation of GEMM presented in Sect. 3.

Concretely, Fig. 6 defines the EG GEMM operation in TVM. Line 1 defines the function, with parameters that now include the tiling/blocking arguments (m_c,n_c, k_c) and the micro-kernel dimensions (m_r, n_r) associated with the GotoBLAS2 GEMM-like structure. Lines 3–4 define the dimensions of the input matrix operands A, B. Line 7 indicates the reduction axis. Lines 10–15 define the packing buffers for A_c, B_c, the correspondence between the buffer entries and those of A, B, and the data type

```
def eg_gemm(m, n, k, mc, nc, kc, mr, nr, dtypeA, dtypeB, dtypeC):
  # Buffers definition
  A = te.placeholder((m, k), name="A", dtype=dtypeA)
  B = te.placeholder((k, n), name="B", dtype=dtypeB)

  # Reduction axis declaration
  p = te.reduce_axis((0, k), name="p")

  # Buffers for packing A and B with casting
  Ac = te.compute(( k//kc, m//mr, kc, mr),
    lambda am, an, az, al:
      A[an * mr + al, am * kc + az].astype(dtypeC), name='Ac')
  Bc = te.compute(( k//kc, n//nr, kc, nr),
    lambda bm, bn, bz, bl:
      B[bm * kc + bz, bn * nr + bl].astype(dtypeC), name='Bc')

  # GEMM computation
  C = te.compute((m, n),
    lambda i, j: te.sum(
      Ac[k//kc, i//mr, p%kc, i%mr] * Bc[k//kc, j//nr, p%kc, j%nr],
      axis=p), name="C",
  )

  # Scheduler creation
  return te.create_schedule(C.op)
```

Fig. 6 TVM definition for the EG GEMM

```
def eg_gemm_scheduling( mc, nc, kc, mr, nr, s):

  #BLIS-like GEMM structure
  ic, jc, icin, jcin = s[C].tile(C.op.axis[0], C.op.axis[1], mc, nc)

  ir, it = s[C].split(icin, factor=mr)
  jr, jt = s[C].split(jcin, factor=nr)

  p, = s[C].op.reduce_axis
  pc, pr = s[C].split(p, factor=kc)

  s[C].reorder(jc, pc, ic, jr, ir, pr, it, jt)

  # Micro-kernel optimization
  s[C].unroll(it);
  s[C].vectorize(jt);

  # Packings location
  s[Ac].compute_at(s[C],ic)
  s[Bc].compute_at(s[C],pc)

  # Packings optimizations, X is used instead of
  # A and B for avoiding duplicated-code
  s[Xc].vectorize(Xc.op.axis[3])

  # Code generation
  f = tvm.lower(s, [A, B, C], name="gemm")
  return tvm.build(f, target="llvm -device=arm_cpu
                                   -mattr=+v8.2a+fp16fml")
```

Fig. 7 Experience-guided schedule configuration for GEMM

casting. Lines 18–22 define the GEMM operation and line 25 creates the scheduler for the C operand.

Figure 7 defines our schedule for the EG GEMM. Lines 4–12 generate the loop structure. Concretely, line 4 splits the loops i, j with factors m_c, n_c, respectively; and lines 6–7 split the latest loops according to the micro-kernel dimensions m_r, n_r. Lines 9–10 split the k loop with a factor k_c. Line 12 reorders the loops in the desired way. Lines 15–16 vectorize the innermost loop. Lines 19–20 move the packings to the desired loop so that the buffer for A_c is placed inside the third loop while that for B_c is placed inside the second loop. Line 24 vectorizes the loads from A, B to A_c, B_c. Finally, lines 27–29 build the code for the desired target architecture.

Let us now focus on the parameters that have to be tuned in our EG GEMM. First, the micro-kernel dimensions ($m_r \times n_r$) dictate the number of vector registers that will be necessary to store the entries of the micro-tile C_r. Therefore, selecting a combination that maximizes this number without exceeding the hardware capacity is fundamental to attain high performance. In addition, the micro-kernel dimensions also affect the cache memory usage, interacting with the optimal choice of the CCPs m_c, n_c, k_c. Tuning our EG GEMM thus consists of building and evaluating the GEMM code for different micro-kernel dimensions. With respect to the CCPs, for each micro-kernel we leverage a variation of the cache memory model in [26] which, given the problem dimensions m, n, k and cache level associativity, returns optimal values for the CCPs m_c, n_c, k_c. Finally, the best-performing configuration is selected and returned.

A simplified version of the IR generated by our scheduler for the first layer of the ResNet-50 v1.5 model and FP32 is shown in Fig. 8. Lines 7–10 allocate buffers for the packings and the data from A_r, B_r inside the micro-kernel. Lines 14–19 initialize the output operand to zero. Lines 22–26 pack B into B_c. In contrast with the auto-scheduling solution, as B is stored in row-major order and we need to pack it by columns, we cannot vectorize the loads from this matrix. Conversely, the packing of A in lines 29–35 is vectorized. Lines 38–52 perform the micro-kernel computation. Line 43 and lines 45–47 respectively load data from A_c and B_c to vector registers; and lines 50–52 perform the computation.

5 Experimental evaluation

5.1 Problem cases in DL for computer vision

The convolution operator is found in well-known neural networks for signal processing (including computer vision) and bears most of the computational cost of model execution. For example, in [27], we report that the convolution layers in the ResNet-50 v1.5 model combined with ImageNet consume between 45% and 87% of the inference time, depending on the optimization level of the model. For high performance, convolutional operators are usually cast in terms of GEMM kernels by applying the IM2COL (or IM2ROW) [28] transform to the input activation matrix. In the following, we focus our study on the IM2COL-transformed GEMMs associated with the convolutional layers of the ResNet-50 v1.5 model combined

```
@I.ir_module
@T.prim_func
def eg_gemm(A: T.Buffer((12544, 147), "float32"),
            B: T.Buffer((147, 64), "float32"),
            C: T.Buffer((12544, 64), "float32"), zero: T.float32):

  Bc = T.allocate([10584], "float32", "global")
  Ac = T.allocate([100254], "float32x4", "global")
  ac = T.allocate([4], "float32", "global")
  bc = T.allocate([6], "float32x4", "global")
  C_1 = T.Buffer((802816,), data=C.data)

  # C initialization
  for i_o_i in range(4):
    for j_i_o_i, i_i_o_i in T.grid(2, 682):
      var_1: T.int32 = i_o_i * 174592 + i_i_o_i * 256 + j_i_o_i * 24
      C_1[var_1:var_1 + 4] = T.Broadcast(T.float32(0), 4)
      ... # Omitted for brevity
      C_1[var_1 + 196:var_1 + 196 +4] = T.Broadcast(T.float32(0), 4)

  # Pack B
  Bc_1 = T.Buffer((10584,), data=Bc)
  for bn, bz in T.grid(3, 147):
    B_1 = T.Buffer((9408,), data=B.data)
    Bc_1[bn * 3528 + bz * 24 + bl_inner_s + 12] =
                              B_1[bz * 64 + bn * 24 + bl_inner_s + 12]

  A_1 = T.Buffer((1843968,), data=A.data)
  for i_o in range(4):
    # Pack A
    Ac_1 = T.Buffer((100254,), "float32x4", data=Ac)
    for an, az in T.grid(682, 147):
      var_6 = i_o
      Ac_1[an * 147 + az] = A_1[var_6 * 401016 + an * 588 + az:
                               var_6 * 401016 + an * 588 + az + 588:147]

      # Micro-kernel
      for j_i_o, i_i_o, k_i in T.grid(2, 682, 147):
        var_31: T.int32 = j_i_o * 3528 + k_i * 24
        ... # Omitted for brevity
        var_7: T.int32 = var_30 + 12
        ac_1 = T.Buffer((4,), data=ac, align=16)
        ac_1[0:4] = Ac_1[i_i_o * 147 + k_i]
        bc_1 = T.Buffer((6,), "float32x4", data=bc)
        bc_1[0] = Bc_1[var_31:var_31 + 4]
        ... # Omitted for brevity
        bc_1[5] = Bc_1[var_31 + 20:var_31 + 20 + 4]

        # GEMM computation
        C_1[var_30:var_30 + 4] += T.Broadcast(ac_1[0], 4) * bc_1[0]
        ... # Omitted for brevity
        C_1[var_21:var_21 + 4] += T.Broadcast(ac_1[3], 4) * bc_1[5]
```

Fig. 8 Resulting IR of the EG scheduler for the first layer of ResNet-50 v1.5 and FP32

with the ImageNet dataset [29]. In the experiments, the batch size b is set to 1 sample, reflecting a latency-oriented scenario.

The dimensions of the GEMM cases targeted in our experiments are shown in Table 1. As some of the layers share their dimensions (e.g., layers 009, 021, and 031), we only list those once in the table and in the performance plots.

Table 1 GEMM dimensions of the convolutional layers in the ResNet-50 v1.5 model with the ImageNet dataset and batch size $b = 1$

Id	Layers	m	n	k
1	001	12,544	64	147
2	006	3136	64	64
3	009/021/031	3136	64	576
4	012/014/024/034	3136	256	64
5	018/028	3136	64	256
6	038	3136	128	256
7	041/053/063/073	784	128	1152
8	044/056/066/076	784	512	128
9	046	784	512	256
10	050/060/070	784	128	512
11	080	784	256	512
12	083/095/105/115/125/135	196	256	2304
13	086/098/108/118/128/138	196	1024	256
14	088	196	1024	512
15	092/102/112/122/132	196	256	1024
16	142	196	512	1024
17	145/157/167	49	512	4608
18	148/160/170	49	2048	512
19	150	49	2048	1024
20	154/164	49	512	2048

5.2 Hardware and software setup

In the evaluation, we target the following two ARM-based development platforms:

- An NVIDIA Carmel processor in the NVIDIA Jetson AGX Xavier platform, with a 64-KB L1 data cache, a 2-MB L2 cache, a 4-MB L3 cache, and a 16-GB LPDDR4x memory. The processor frequency in this platform is fixed to 2.3 GHz.
- An NVIDIA Cortex A57 processor, in the NVIDIA Jetson Nano board, with a 32-KB L1 data cache, a 2-MB L2 cache, and a 4-GB LPDDR4 memory. The processor frequency in this platform is fixed to 1.48 GHz.

These target systems are representative of some of the equipment used today to infer DL models for computer vision.

A single core is employed in the two architectures because adding parallelism benefits all the approaches. All the experiments are repeated a large number of times, reporting the average results. Performance is measured in terms of billions of floating-point operations per second (GFLOPS) when the resulting operand is floating point and billions of integer operations per second (GIOPS) when the resulting operand is integer. In addition, the results are expressed in seconds in the case of aggregated time and the auto-scheduling time.

For Apache TVM, we have used version 0.17, the last stable release. The code of our EG environment is available at https://github.com/adcastel/TVM_quantized.

5.3 Performance evaluation

In this subsection, we evaluate the combinations of data types for A, B, C displayed in Table 2, which also reports the reduction memory factor per operand with respect to the FP32 approach.

The bars in the following performance plots labeled as `TVM-nnn` correspond to the TVM auto-scheduling method, where the `nnn` specifies the number of trials that the auto-scheduler was allowed to explore when searching for the optimal solution. The bars with the label `EG-TVM` display the results from our EG scheduling. We only report the results once for those convolutional layers with the same dimensions. In contrast, when measuring the total execution time, we consider the cost of all layers.

Figure 9 shows the performance when using only floating-point operands on the NVIDIA Jetson AGX Xavier (single NVIDIA Carmel core). It is clear that when there is no need for casting (`FP32|FP32` and `FP16|FP16`), leveraging the TVM auto-scheduling with 1000 trials is the best option for almost every layer. In fact, this approach attains a performance rate that is close to the peak of the machine: 36.8 GFLOPS for FP32 and 73.6 GFLOPS for FP16. The results of the two approaches are closer when casting from FP16 to FP32 is needed. In that scenario, our EG GEMM outperforms the auto-scheduling in 12 of the 20 layers. The difference in performance comes not only from the schedule but also from the place where the casting is done. In EG GEMM, the casting is done in the packing routine, so when the data are loaded in the micro-kernel, it is already in the destination operand data type. Conversely, for auto-scheduling, this casting can only be declared in the computation and, therefore, it is performed inside the micro-kernel.

It is noticeable that, in some layers, increasing the number of auto-tuning trials from 500 to 1,000 does not add any extra performance. In some cases, the solution found by the former outperforms the more complete search. This situation is caused because of the nondeterministic starting exploration point.

When operands A and B are INT8 integers, casting to a data type with a wider range is in general required to avoid overflow/underflow errors. Figure 10 reveals the performance difference among GEMMs when A and B are 8-bits integers and C is stored in INT32, FP32, or FP16. In the former case, the winner is auto-scheduling, which dominates on 17 of 20 layers. In contrast, when casting to any of the two floating-point data types, our EG GEMM outperforms the auto-scheduling variants,

Table 2 Combinations of the data types and memory reduction with respect to FP32

Combination	*A*	*B*	*C*	*A* Factor	*B* Factor	*C* Factor
`FP32|FP32`	FP32	FP32	FP32	1×	1×	1×
`FP16|FP16`	FP16	FP16	FP16	2×	2×	2×
`FP16|FP32`	FP16	FP16	FP32	2×	2×	1×
`I8|I32`	INT8	INT8	INT32	4×	4×	1×
`I8|FP32`	INT8	INT8	FP32	4×	4×	1×
`I8|FP16`	INT8	INT8	FP16	4×	4×	2×

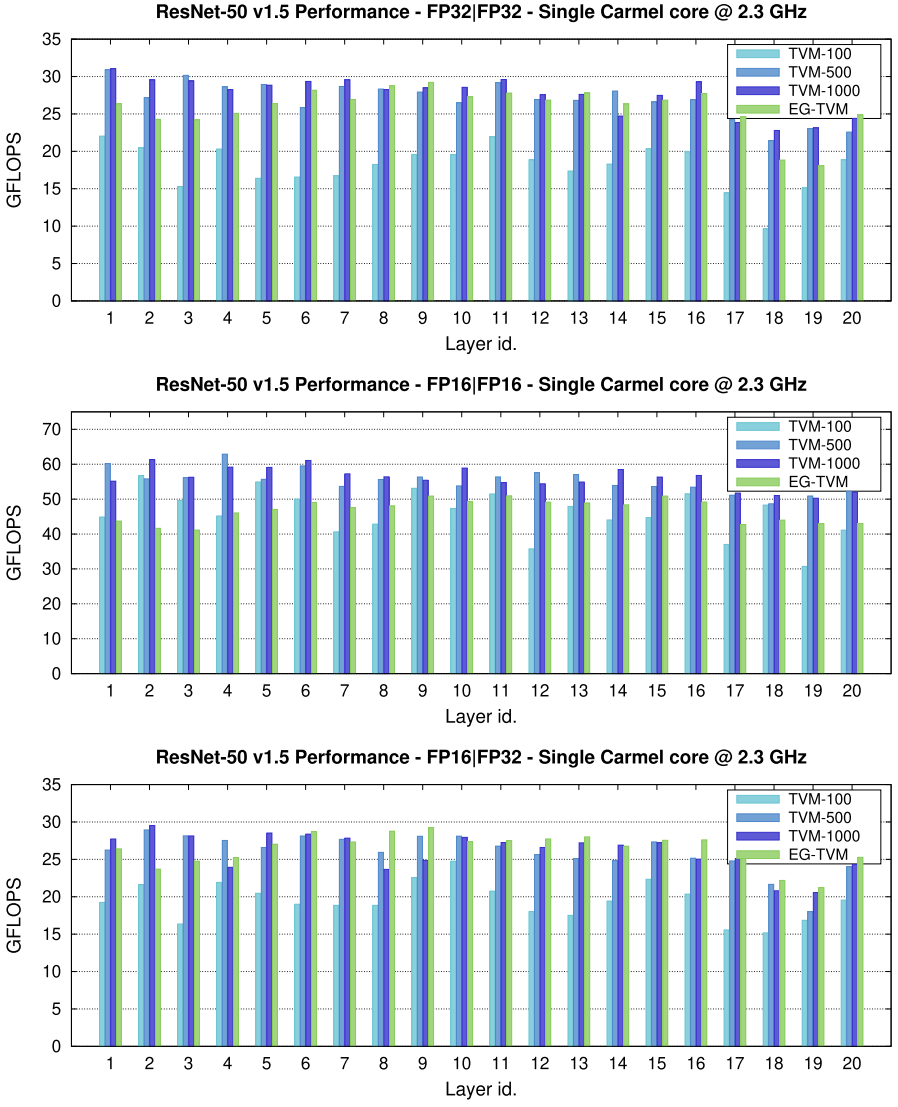


Fig. 9 Performance in GFLOPS of the distinct convolutional layers in ResNet-50 v1.5+ImageNet on a single core of the NVIDIA Jetson AGX Xavier when using `FP32|FP32` (top), `FP16|FP16` (middle), and `FP16|FP32` (bottom)

being the top choice in 14 of 20 layers, both for the `I8|FP32` and `I32|FP16` scenarios.

Figure 11 compares the total execution time of the ResNet-50 v1.5 model for the distinct combinations of data types. Thanks to the special support of FP16 arithmetic in this device, the fastest execution time corresponds to auto-scheduling with pure FP16 operands. This approach reduces memory consumption by a factor 2× with

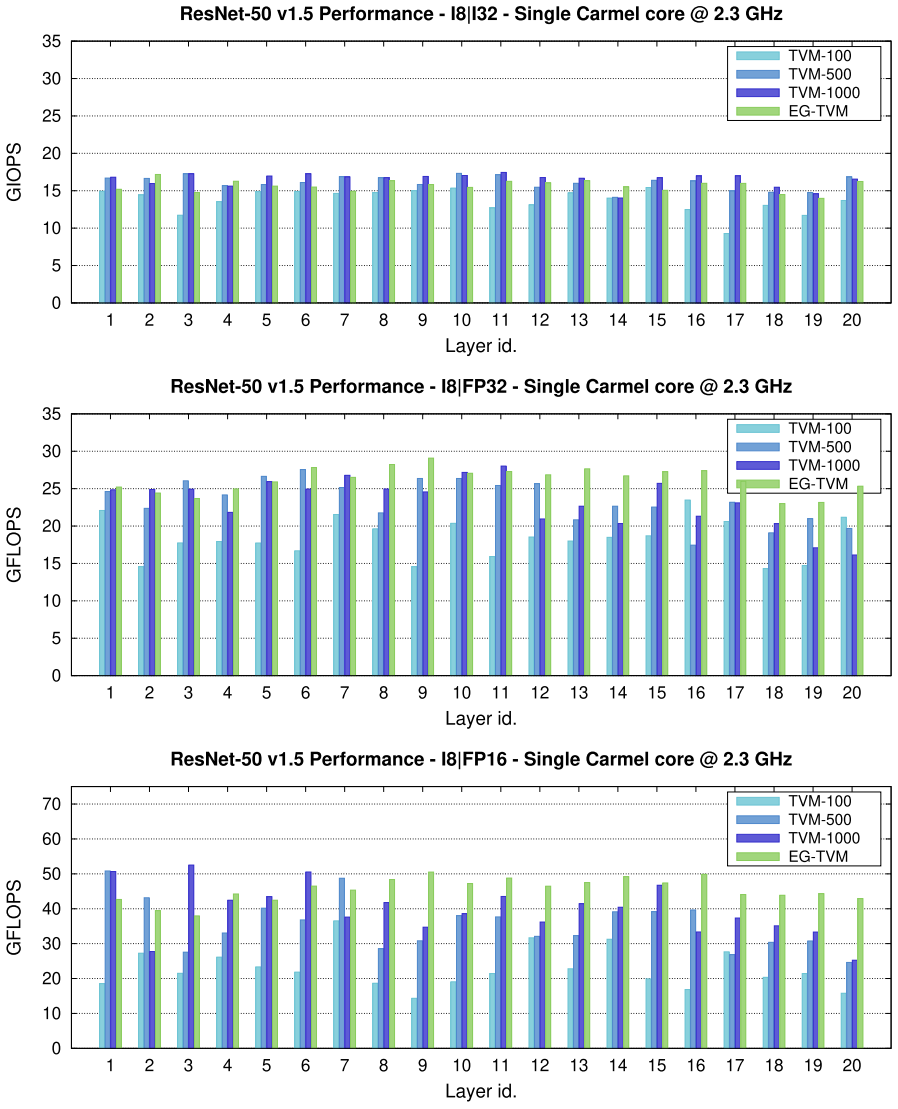


Fig. 10 Performance in GIOPS (top) and GFLOPS (middle and bottom) of the distinct convolutional layers in ResNet-50 v1.5+ImageNet on a single core of the NVIDIA Jetson AGX Xavier when using `I8|I32` (top), `I8|FP32` (middle), and `I8|FP16` (bottom)

respect to its FP32 counterpart. When A, B are stored as INT8 numbers, the memory consumption for A, B is reduced by a factor 4×. Among the six combinations, auto-scheduling is the best option in three scenarios: `FP32|FP32`, `FP16|FP16`, and `I8|I32`, while in the remaining three cases, EG GEMM is the best choice.

We next focus on the NVIDIA Jetson AGX Nano. As this platform does not support the FP16 fused multiply-add in hardware, we remove those scenarios

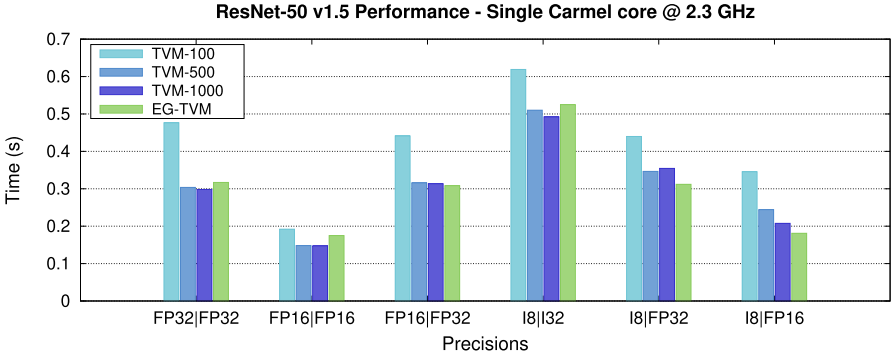


Fig. 11 Total execution time (in seconds) for ResNet-50 v1.5+ImageNet on a single core of the NVIDIA Jetson AGX Xavier using different combinations of data types

involving that type of arithmetic from the following study. Figure 12 reports the GFLOPS for `FP32|FP32` and `FP16|FP32`. When no casting is needed, auto-scheduling is the best option for 11 from 20 layers. However, the outcome varies when the data is stored initially in FP16 and cast to FP32. In that case, our EG

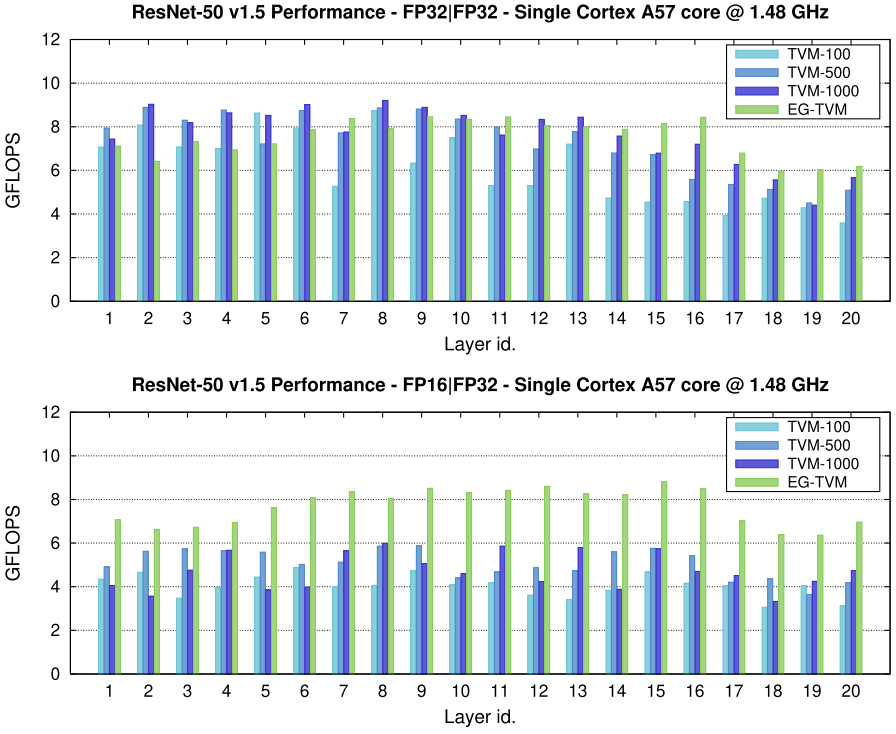


Fig. 12 Performance in GFLOPS of the distinct convolutional layers in ResNet-50 v1.5+ImageNet on a single core of the NVIDIA Jetson Nano when using `FP32|FP32` (top) and `FP16|FP32` (bottom)

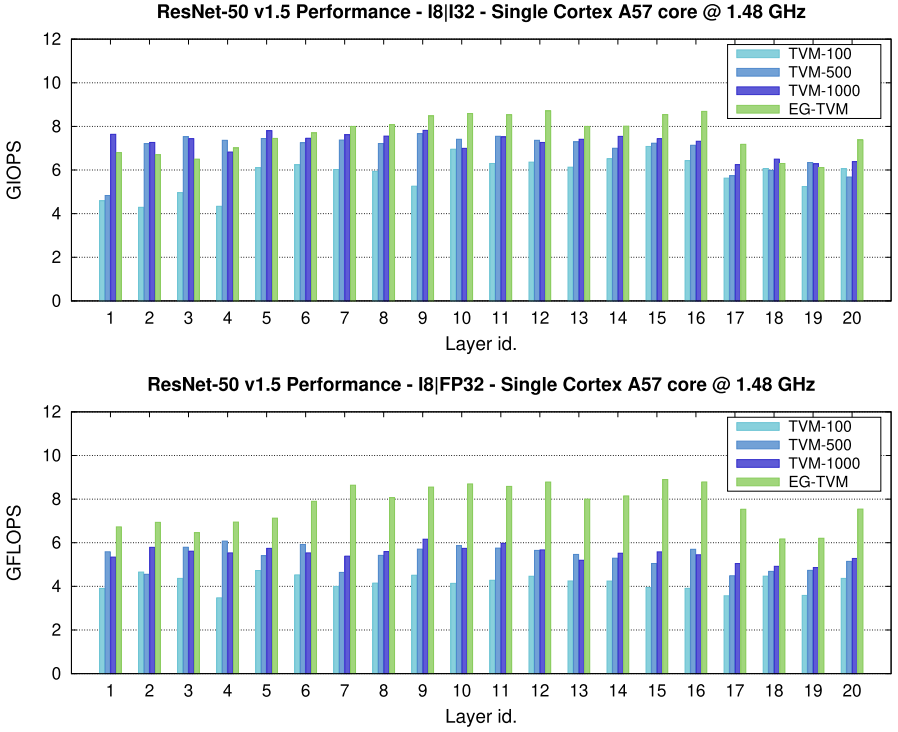


Fig. 13 Performance in GIOPS (top) and GFLOPS (bottom) of the distinct convolutional layers in ResNet-50 v1.5+ImageNet on a single core of the NVIDIA Jetson Nano when using `INT8|INT32` (top) and `INT8|FP32` (bottom)

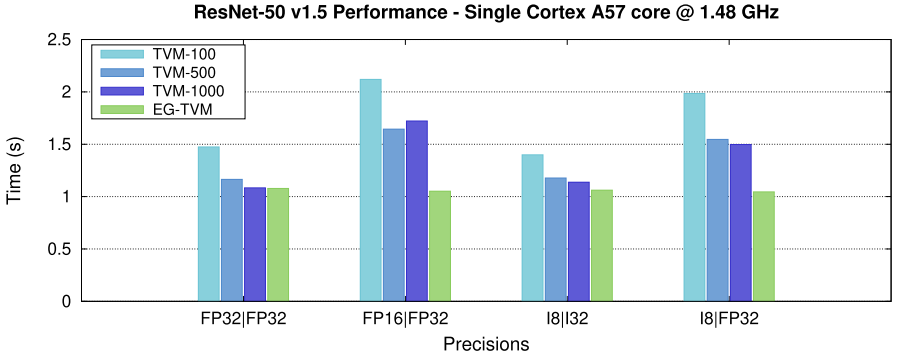


Fig. 14 Total execution time (in seconds) for ResNet-50 v1.5+ImageNet on a single core of the NVIDIA Jetson Nano using different combinations of data types

GEMM fully dominates the performance. Figure 13 shows the performance when A, B operands are stored as INT8 matrices. In both cases, EG GEMM is the clear winner for almost the range of convolutions.

Table 3 Total execution time (in seconds) of the tuning processes for the ResNet-50 v1.5+ImageNet with batch size $b = 1$

	TVM-100	TVM-500	TVM-1000	EG-TVM
NVIDIA Jetson AGX Xavier	3,600	14,400	28,800	600
NVIDIA Jetson Nano	8,400	39,600	82,800	1,800

Figure 14 reports the total execution time of the entire ResNet-50 v1.5 model. In this platform, our EG GEMM is clearly the best option.

5.4 Scheduling time

A relevant aspect is the time that is required by the auto-scheduler to identify an optimal configuration. To illustrate this, we have measured the total time of tuning the 20 distinct layers of the ResNet-50 v1.5+ImageNet model with $b = 1$. Table 3 summarizes the time (in seconds) spent on the tuning process for the NVIDIA Jetson AGX Xavier. The reported times there correspond to a unique combination of data types. In order to tune the schedule for the six data types used in this paper, these costs have to be multiplied by six. If the batch size b is increased, or we want to use a dataset different from ImageNet, the full optimization process has to be repeated. The optimal solutions identified by the auto-scheduler are hardware-specific. Moving to a different processor involves the repetition of the full optimization process.

Incrementing the number of trials from 100 to 1,000 for the auto-scheduler increases the total tuning time by 8× and almost 10× for NVIDIA Jetson AGX Xavier and NVIDIA Jetson Nano, respectively. These numbers also highlight that using our EG tuning method involves a tuning time that is lower in a factor of 48× and 46× with respect to the `TVM-1000` option in the NVIDIA Jetson AGX Xavier and NVIDIA Jetson Nano, respectively.

6 Conclusion

We have compared two tuning methodologies for the homogeneous precision GEMM as well as several mixed-precision counterparts which are usually necessary in quantized DL. Specifically, we have compared the TVM auto-scheduling tool with our EG scheduling for multiple combinations of data types on two low-power ARM-based processors. We have demonstrated that, when no type casting of the matrix operands is required, the TVM auto-scheduler identifies the best option. However, when the multiplication involves matrices of different data types, the EG GEMM delivers the best solution and, furthermore, requires a considerably lower tuning cost. This paper thus demonstrates that using our EG environment performs close to or even better than the solution found by the TVM auto-scheduler for quantized DL scenarios.

Acknowledgements This work received funding from projects PID2020-113656RB and PID2021-126576NB-I00 of MCIN/AEI/ 10.13039/501100011033; PROMETEO 2023-CI PROM/2022/20. H. Martínez is a POSTDOC_21_00025 postdoctoral fellow supported by *Junta de Andalucía*. S. Catalán is supported by the grant RYC2021-033973-I, funded by MCIN/AEI/10. 13039/501100011033 and the "NextGenerationEU"/ PRTR, and UJI-2023-04, funded by Universitat Jaume I.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Jacob B *et al.* (2017) Quantization and training of neural networks for efficient integer-arithmetic-only inference
2. Pati S, Aga S, Jayasena N, Sinclair MD (2022) Demystifying bert: system design implications, In: IEEE International Symposium on Workload Characterization (IISWC) 2022:296–309
3. Sze V, Chen Y-H, Yang T-J, Emer JS (2017) Efficient processing of deep neural networks: a tutorial and survey. Proc. IEEE 105(12):2295–2329
4. Ben-Nun T, Hoefler T (2019) Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. ACM Comput. Surv. 52(4):65:1-65:43
5. Goto K, van de Geijn RA (2008) Anatomy of a high-performance matrix multiplication. ACM Trans. Math. Softw. 34(3):121–1225
6. Xianyi Z, Qian W, Yunquan Z (2012) Model-driven level 3 BLAS performance optimization on Loongson 3A processor, In: 2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)
7. Van Zee FG, van de Geijn RA (2015) BLIS: a framework for rapidly instantiating BLAS functionality. ACM Trans. Math. Softw. 41(3):14:1-14:33
8. Clint Whaley R, Petitet A, Dongarra JJ (2001) Automated empirical optimizations of software and the atlas project. Parallel Comput. 27(1):3–35
9. Ragan-Kelley J, Barnes C, Adams A, Paris S, Durand F, Amarasinghe S (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, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, p. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176
10. Chen T, Moreau T, Jiang Z, Shen H, Yan EQ, Wang L, Hu Y, Ceze L, Guestrin C, Krishnamurthy A (2018) TVM: end-to-end optimization stack for deep learning, *CoRR*, vol. abs/1802.04799. [Online]. Available: http://arxiv.org/abs/1802.04799
11. Ikarashi Y, Bernstein GL, Reinking A, Genc H, Ragan-Kelley J (2022) Exocompilation for productive programming of hardware accelerators, In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 703–718
12. Bradbury J, Frostig R, Hawkins P, Johnson MJ, Leary C, Maclaurin D, Necula G, Paszke A, VanderPlas J, Wanderman-Milne S, Zhang Q (2018) JAX: composable transformations of Python+NumPy programs. [Online]. Available: http://github.com/google/jax
13. Sabne A (2020) Xla : Compiling machine learning for peak performance

14. Baghdadi R, Ray J, Romdhane MB, Sozzo ED, Akkas A, Zhang Y, Suriana P, Kamil S, Amarasinghe S (2019) Tiramisu: a polyhedral compiler for expressing fast and portable code, In: IEEE/ACM International Symposium on Code Generation and Optimization (CGO) 2019:193–205
15. Li M, Liu Y, Liu X, Sun Q, You X, Yang H, Luan Z, Qian D (2020) The deep learning compiler: a comprehensive survey, *CoRR*, vol. abs/2002.03794. [Online]. Available: https://arxiv.org/abs/2002.03794
16. Moreau T, Chen T, Jiang Z, Ceze L, Guestrin C, Krishnamurthy A (2018) VTA: an open hardware-software stack for deep learning, *CoRR*, vol. abs/1807.04188. [Online]. Available: http://arxiv.org/abs/1807.04188
17. Lattner C, Pienaar JA, Amini M, Bondhugula U, Riddle R, Cohen A, Shpeisman T, Davis A, Vasilache N, Zinenko O (2020) MLIR: a compiler infrastructure for the end of Moore's law, *CoRR*, vol. abs/2002.11054, [Online]. Available: https://arxiv.org/abs/2002.11054
18. Kang D, Kim E, Bae I, Egger B, Ha S (2018) C-good: C-code generation framework for optimized on-device deep learning, In: Proceedings of the International Conference on Computer-Aided Design, ser. ICCAD '18. New York, NY, USA: Association for Computing Machinery. [Online]. Available: https://doi.org/10.1145/3240765.3240786
19. Chen T, Moreau T, Jiang Z, Zheng L, Yan E, Shen H, Cowan M, Wang L, Hu Y, Ceze L, Guestrin C, Krishnamurthy A (Oct. 2018) TVM: An automated End-to-End optimizing compiler for deep learning, In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). Carlsbad, CA: USENIX Association, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen
20. Zheng L, Jia C, Sun M, Wu Z, Yu CH, Haj-Ali A, Wang Y, Yang J, Zhuo D, Sen K *et al.* (2020) Ansor: Generating {High-Performance} tensor programs for deep learning, In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), 863–879
21. Chen T, Zheng L, Yan EQ, Jiang Z, Moreau T, Ceze L, Guestrin C, Krishnamurthy A (2018) Learning to optimize tensor programs, *CoRR*, vol. abs/1805.08166. [Online]. Available: http://arxiv.org/abs/1805.08166
22. Bondhugula U (2020) High performance code generation in MLIR: an early case study with GEMM, *CoRR*, vol. abs/2003.00532. [Online]. Available: https://arxiv.org/abs/2003.00532
23. Castelló A, Bellavita J, Dinh G, Ikarashi Y, Martínez H (2024) Tackling the matrix multiplication micro-kernel generation with exo, In: IEEE/ACM International Symposium on Code Generation and Optimization (CGO) 2024:182–193
24. Zhang Y (2009) Parallel Solution of Integral Equation-Based EM Problems in the Frequency Domain. IEEE Press
25. Alaejos G, Castelló A, Alonso-Jordá P, Igual FD, Martínez H, Quintana-Ortí ES (2024) "Algorithm 1039: Automatic generators for a family of matrix multiplication routines with apache tvm," ACM Trans. Math. Softw., 50(1). [Online]. Available: https://doi.org/10.1145/3638532
26. Low TM, Igual FD, Smith TM, Quintana-Ortí ES (2016) Analytical modeling is enough for high-performance BLIS. ACM Trans. Math. Softw. 43(2):12:1-12:18
27. Barrachina S, Dolz MF, San-Juan P, Quintana-Ortí ES (2022) Efficient and portable GEMM-based convolution operators for deep neural network training on multicore processors. J. Parallel Distrib. Comput. 167(C):240–254
28. Chellapilla K, Puri S, Simard P (2006) High Performance Convolutional Neural Networks for Document Processing, In: International Workshop on Frontiers in Handwriting Recognition
29. He K, Zhang X, Ren S, Sun J (2016) Deep Residual Learning for Image Recognition, In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 770–778

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.