

Convolution Operators for Deep Learning Inference: Libraries or Automatic Generation?

Guillermo Alaejos, Adrián Castelló, Pedro Alonso-Jordá, Enrique S. Quintana-Ortí

Universitat Politècnica de València, Spain

{galalop,palonso}@upv.es, {adcastel,quintana}@disca.upv.es

Francisco D. Igual

Universidad Complutense de Madrid, Spain

figual@ucm.es

Abstract—Convolutional deep neural networks often leverage the so-called lowering (or IM2COL) approach to transform their convolution operators into large, cache-friendly general matrix-matrix multiplications (GEMM). This path offers high flexibility, does not perturb the numerical accuracy of the result, and benefits from the existence of high-performance realizations of GEMM in numerical linear algebra libraries. However, the GEMM kernels in these libraries present a series of performance-related issues, in the particular case of deep learning inference, which turns automatic generation into an appealing alternative.

While automatic generation combined with a brute force search of the optimization space is possible, in our work we advocate for a hybrid solution, with the code generation guided by standard techniques embedded in linear algebra libraries, and complemented with analytical models for parameter selection. The code generation effort is reduced to the automatic generation of a small component of the operation, known as the micro-kernel. As a result, this solution can be rapidly and efficiently tailored to different data types, processor architectures, and convolution operator shapes.

Index Terms—Convolution, Deep learning, Matrix multiplication, Apache TVM, SIMD vectorization.

I. INTRODUCTION

A. Libraries

Matrix multiplication (GEMM) is a key operation for deep learning (DL), leveraged by transformers for natural language processing and convolutional deep neural networks (DNNs) for signal processing and computer vision [1], [2]. GEMM is also a crucial computational kernel upon which linear algebra (LA) libraries are built. Therefore, it is natural that, over the past decades, there has been a continuous effort toward developing high performance realizations of GEMM for a variety of architectures, resulting in a fair collection of commercial as well as open source products, such as Intel oneMKL, AMD AOCL, IBM ESSL, ARMPL, NVIDIA cuBLAS, GotoBLAS2 [3], OpenBLAS [4], and BLIS [5].

Unfortunately, when the objective is deploying software for DL inference on low-power processors that operate close to the edge [6], these LA libraries present several problems:

1. The memory footprint of conventional libraries is in the order of Mbytes, which may be excessive for embedded devices for inference such as micro-controller units. Here the problem comes from the fact that LA libraries integrate functionality far beyond what is needed in DL

inference (e.g., real/complex arithmetic, single/double precision, a large variety of routines for LA operations). Even worse, they *miss some relevant cases* as, for example, support for reduced (i.e., 16-bit) floating point precision or mixed precision.

2. The implementation of GEMM in these libraries is sub-optimal under certain circumstances. The reason is that they are usually tuned for the general, large-scale case while, at the same time, they are designed to work for any problem dimension. Thus, these libraries employ blocking parameters that are statically fixed during the installation, and require recompilation in case they turn out to be sub-optimal for a particular problem dimension. *The caveat is that the problems that usually arise in DL inference tasks are far from being “big and square”.*
3. These libraries are hardware-specific. This is obviously the case for Intel, AMD, IBM, ARM and NVIDIA’s packages. To a certain extent, it also applies to OpenBLAS and BLIS, which require a hardware-specific micro-kernel [5]. *In contrast, edge computing is highly heterogeneous, encompassing hundreds of distinct deployed micro-architectures.*

B. Automatic generation

Many common DL frameworks and compiler frameworks, including TFLite, XLA, MLIR and TVM, rely on JIT (just-in-time) compilation. Armed with hardware-agnostic IRs (Intermediate Representations), these software infrastructures effectively decouple schedules and computation, and enable the automatic exploration of the scheduling and configuration spaces by means of auto-tuning techniques.

A brute force optimization scheme, such as that performed in AutoTVM [7], 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 have been enhanced with Random Search, Bayesian optimization, Genetic algorithms, and Deep Reinforcement Learning. While

all these efforts reduce the cost of the hyperparameter search, they are still computationally expensive and, in many cases, yield solutions that are difficult to explain and reproduce for developers or to port to embedded architectures or systems with reduced compute capabilities.

C. Hybridization

In this paper we address the limitations of current LA libraries and pure automatic generation techniques by demonstrating that it is possible to automatically generate a BLIS-like algorithm and a collection of micro-kernels for GEMM, using Apache TVM [8], offering an alternative solution with the following advantages:

1. By adjusting the algorithm and micro-kernel to the problem dimensions, it is possible to outperform high performance realizations of GEMM in commercial as well as academic libraries.
2. The optimization process for each problem dimension is largely seamless, boiling down to the evaluation of a reduced number of micro-kernels.
3. The generation/optimization tools can be easily specialized for any data type, *enhancing the portability and maintainability of the solution*.
4. The entire framework library is very small.
5. The ideas extend to the implementation of convolution operators for DL via the IM2COL approach or the direct convolution [9].

II. THE BLIS ALGORITHM FOR GEMM

A. Blocking (and packing) for the cache

Consider the matrix multiplication $C = C + AB$, abbreviated as $C += AB$, 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 BLIS realization of this LA kernel (as well as that in other libraries such as OpenBLAS, Intel oneMKL, and AMD AOCL –actually based on BLIS–) follows the basic ideas of GotoBLAS2 to decompose the computation into five nested loops, traversing the m, n, k dimensions of the problem in a certain order. Inside these loops, two packing routines copy parts of the input operands A, B into two special buffers, $A_c \rightarrow m_c \times k_c$, $B_c \rightarrow k_c \times n_c$, in order to ensure 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, inside the fifth loop there is a micro-kernel that is often vectorized to exploit the SIMD FPUs (single-instruction, multiple-data floating point units) in current multicore processors [5].

The orchestration of the data movements across the memory hierarchy in the BLIS algorithm for GEMM is favored by the specific nesting of the algorithm loops, in combination with a careful choice of the loop strides and the sizes of the buffers [10]. The operand partitionings induced by the loops, and the target level of the memory hierarchy for each operand block are illustrated in Figure 1.

B. SIMD micro-kernels

The BLIS algorithm casts its innermost computation in terms of a micro-kernel that computes the smaller GEMM $C_r += A_r B_r$, where $A_r \rightarrow m_r \times k_c$, $B_r \rightarrow k_c \times n_r$ respectively denote micro-panels of the buffers A_c, B_c , while $C_r \rightarrow m_r \times n_r$ is a small micro-tile. (For simplicity, hereafter we assume that m_c and n_c are respectively integer multiples of m_r and n_r .) This corresponds to the operation performed in the innermost loop of the BLIS algorithm (labeled as L5 there), with

$$\begin{aligned} A_r &= A_c(ir:ir+mr-1, 0:kc-1), \\ B_r &= B_c(0:kc-1, jr:jr+nr-1), \text{ and} \\ C_r &= C(ic+ir:ic+ir+mr-1, jc+jr:jc+jr+nr-1). \end{aligned}$$

Inside loop L6, the micro-kernel iterates across the k_c dimension of the problem, 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 2.

For high performance, the data in A_c/B_c are carefully packed to ensure access with unit stride to the columns/rows of A_r/B_r from within the micro-kernel; see Figure 3. This reduces cache evictions during these accesses as well as accommodates the use of efficient SIMD instructions to load their elements into vector registers.

There are a few rules of thumb that guide the design of a high performance realization of the micro-kernel [10]:

- Considering the k_c successive updates of the micro-tile C_r occurring in loop L6, 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 principles suggest maximizing the values for m_r, n_r as part of a “large” micro-kernel. In practice, the limited number of vector registers in current FPUs constrain the practical values of m_r, n_r for conventional, manually-developed realizations of the micro-kernels within a couple of dozens.

The actual implementation of these micro-kernels 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.

III. AUTOMATIC GENERATION OF A BLIS-LIKE ROUTINE FOR GEMM

Apache TVM is an open source compiler framework that allows to generate, optimize, and execute machine learning kernels on multicore processors, GPUs (graphics processing units), and other accelerator backends [11]. In our effort toward the automatic generation of a BLIS-like algorithm, we build upon the instructions in the basic tutorial for TVM.¹

¹<https://tvm.apache.org/docs/tutorials/>

Loop | BLIS algorithm for GEMM

```

L1 | for ( jc=0; jc<n; jc+=nc )
L2 |   for ( pc=0; pc<k; pc+=kc ) {
L3 |     Bc := B(pc:pc+kc-1, jc:jc+nc-1);
L4 |     for ( ic=0; ic<m; ic+=mc ) {
L5 |       Ac := A(ic:ic+mc-1, pc:pc+kc-1);
L6 |       for ( jr=0; jr<nc; jr+=nr )
L7 |         for ( ir=0; ir<mc; ir+=mr )
L8 |           C(ic+ir:ic+ir+mr-1, jc+jr:jc+jr+nr-1)
L9 |             += Ac(ir:ir+mr-1, 0:kc-1)
L10 |              * Bc(0:kc-1, jr:jr+nr-1);
L11 |     }
L12 |   }
L13 | }

```

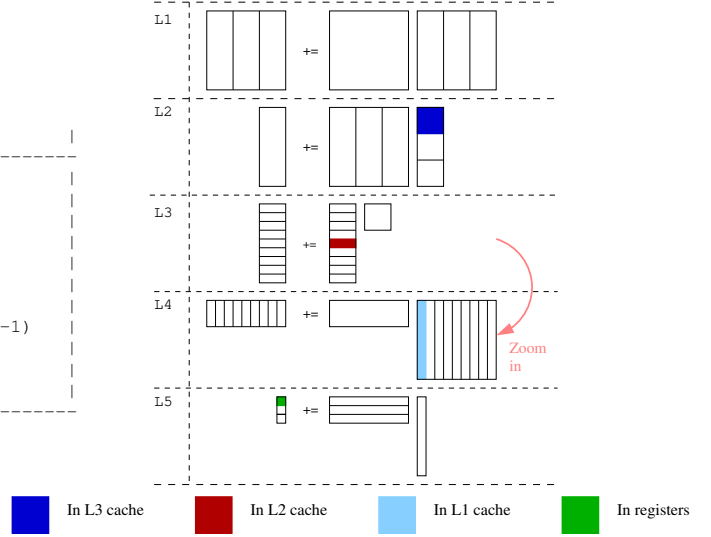


Fig. 1: BLIS algorithm for GEMM, with C , A and B respectively streamed from the main memory, L2 cache and L3/L1 cache into the processor registers.

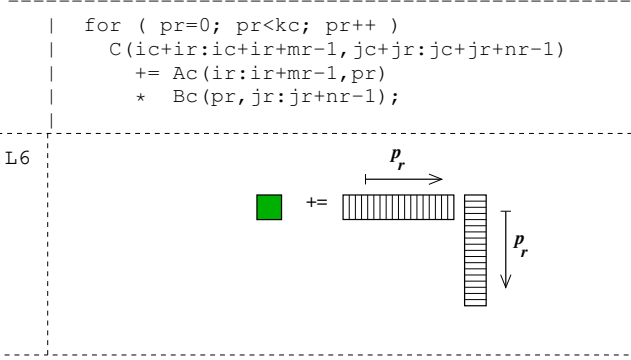


Fig. 2: Micro-kernel with C resident in the processor registers.

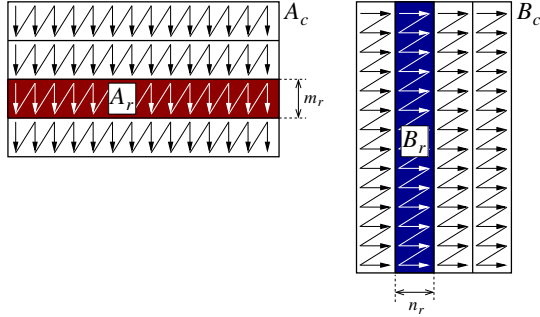


Fig. 3: Packing in the BLIS algorithm.

A. Blocking and packing

Figure 4 provides a TVM generation script that produces a code with the same organization as that proposed in the BLIS algorithm. We highlight the following aspects in the script:

- Lines 3–4 define two “virtual” operands, of the appropriate dimensions, for A and B .
- Lines 8–12 declare a 4D TVM tensor, A_c , which acts as a “view” into the A operand. The lambda function in Line 10 induces a data copy from operand A to A_c . The four variables (i, j, q, r) of the function are translated by TVM

into four loops, traversing the corresponding dimensions of the A_c tensor (e.g., $0 < q < k_c$). These variables are then used in the subsequent expression to indicate the correspondence between the entries of A and A_c .

- Lines 13–17, involving the “view” B_c and the B operand, play an analogous role to that described in the previous item for A_c and A .
- Lines 19–27 define the operation to be computed in terms of the A_c and B_c views. Here A_c is transposed, which is necessary at this point to ensure that TVM generates a code that accesses the entries of A_c with unit stride.
- Lines 30–35 create a schedule, extract the loop indices, and then instruct TVM on how to nest them.
- Lines 38–39 place the packings for B_c and A_c at the desired points of the loop nesting.
- Finally, lines 45–46 induce TVM to generate the code, in this case, for an LLVM backend.

B. Automatic Generation SIMD micro-kernels

The next stage in our journey to obtain a high performance realization of GEMM has the objective of generating high performance micro-kernels which can then be integrated within the GEMM BLIS-like TVM-based algorithm. For this purpose, starting from the TVM code in Figure 4, we introduce two major changes, with the result shown in Figure 5:

- The m_c, n_c dimensions are further partitioned in order to expose the loops that will operate with the individual elements of the micro-panel C_r ; see lines 6–7.
- Instructions for vectorization together with loop unrolling are passed to TVM in lines 14–19.
- TVM also allows to exploit multi-threading. (Omitted for brevity.)

C. Lowering

Consider a convolution operator receiving a 4D tensor I , composed of t inputs of dimension $c_i \times h_i \times w_i$ each, where c_i

```

1 def packed_GEMM(m, n, k, mc, nc, kc, mr, nr):
2     # P1) Define operation
3     A = te.placeholder((m, k), name="A")
4     B = te.placeholder((k, n), name="B")
5
6     # 4D view into A and B to induce creation
7     # of buffer by TVM
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     Bc = te.compute((math.ceil(k/kc),
14                     math.ceil(n/nr), kc, nr),
15                    lambda i, j, q, r:
16                      B[i * kc + q, j * nr + r],
17                    name="Bc")
18
19     p = te.reduce_axis((0, k), "p")
20     C = te.compute((m, n), lambda i, j:
21                   te.sum(Ac[p//kc, i//mr,
22                          tvml.tir.indexmod(p, kc),
23                          tvml.tir.indexmod(i, mr)] *
24                          Bc[p//kc, j//nr,
25                          tvml.tir.indexmod(p, kc),
26                          tvml.tir.indexmod(j, nr)],
27                   axis=p), name="C")
28
29     # P2) Prepare schedule
30     sched = te.create_schedule(C.op)
31     ic, jc, \
32     ir, jr = sched[C].tile(C.op.axis[0],
33                          C.op.axis[1],
34                          mc, nc)
35     pc, pr = sched[C].split(p, factor=kc)
36
37     # P3) Place Ac, Bc in the desired loops
38     sched[Bc].compute_at(sched[C], pc)
39     sched[Ac].compute_at(sched[C], ic)
40
41     # P4) Loop schedule as in B3A2C0
42     sched[C].reorder(jc, pc, ic, jr, ir, pr)
43
44     # P5) Generate code with LLVM backend
45     return tvml.build(sched, [A, B, C],
46                      target="llvm")

```

Fig. 4: TVM generator for GEMM mimicking the blocking and packing schemes of the BLIS algorithm.

stands for the number of input channels and $h_i \times w_i$ denote the size of the input image (height \times width). Furthermore, assume the convolution kernel applies a 4D tensor F consisting of c_o filters of dimension $c_i \times h_f \times w_f$ each, where $h_f \times w_f$ specify the size of each 2D filter. The convolution

$$O = \text{CONV}(F, I), \quad (1)$$

then computes a 4D output tensor O , with t outputs of size $c_o \times h_o \times w_o$ each. Here, each of the c_o individual filters in this layer combines a (sub)tensor of the inputs, with the same dimension as the filter, to produce a single scalar value (entry) in one of the c_o outputs. By repeatedly applying the filter to the whole input, in a sliding window manner (and with certain height/width strides h_s and w_s), the convolution operator produces the complete entries of this single output; see [1]. Assuming height/width paddings given by h_p and w_p , the output dimensions become $h_o = \lfloor (h_i - h_f + 2h_p) / h_s + 1 \rfloor$ and $w_o = \lfloor (w_i - w_f + 2w_p) / w_s + 1 \rfloor$.

```

1 def opt_GEMM(m, n, k, mc, nc, kc, mr, nr):
2     # P1), P2), P3) as in packed_GEMM
3     # Omitted for brevity
4
5     # P4) Expose loops inside micro-kernel
6     ir, it = sched[C].split(ir, factor=mr)
7     jr, jt = sched[C].split(jr, factor=nr)
8
9     # P5) Loop schedule as in B3A2C0
10    sched[C].reorder(jc, pc, ic,
11                   jr, ir, pr, it, jt)
12
13    # P6) Unroll+vectorize micro-kernel loops
14    sched[C].unroll(it)
15    sched[C].vectorize(jt)
16    i, j = Bc.op.axis
17    sched[Bc].vectorize(j)
18    i, j, q, r = Ac.op.axis
19    sched[Ac].vectorize(r)
20
21    # P7) Generate code with LLVM backend
22    return tvml.build(sched, [A, B, C],
23                     target="llvm")

```

Fig. 5: TVM generator for GEMM mimicking the blocking and packing schemes of the BLIS algorithm, and integrating the optimized micro-kernel.

Among the different methods to realize the convolution operator, the IM2COL approach [9] is a popular option due to its superior flexibility and fair performance. Concretely, this approach transforms the input tensor I into an augmented matrix B so that (1) can be obtained from the GEMM:

$$C = A \cdot B = A \cdot \text{IM2COL}(I),$$

where $C \equiv O \rightarrow c_o \times (t \cdot h_o \cdot w_o)$ is the output tensor viewed as an $m \times n$ matrix, with $m = c_o$ and $n = h_o \cdot w_o \cdot t$; $A \equiv F \rightarrow c_o \times (c_i \cdot h_f \cdot w_f) = m \times k$ contains the filters; and $B \rightarrow (c_i \cdot h_f \cdot w_f) \times (t \cdot h_o \cdot w_o) = k \times n$ results from applying the IM2COL transform to the input tensor I according to the filter dimensions and strides (h_f, w_f, h_s, w_s).

The IM2COL transform can be easily obtained using a lambda function in Python, as shown in Figure 6. For the IM2COL transformation with TVM:

- The dimensions of the I operand are specified in line 3.
- Lines 5, 6 calculate the dimensions of the result.
- The operation is defined in line 8–19, employing several conditions to ensure that data accesses remain within the boundaries of the matrix.
- Lines 22 and 25–26 respectively define the schedule and build the function.

IV. EXPERIMENTAL RESULTS

In this section we evaluate the performance of the GEMM BLIS-like routine and micro-kernels automatically generated using TVM. All the experiments employ IEEE 32-bit floating point arithmetic (FP32).

A. Setup

The experiments were carried out in an NVIDIA Jetson AGX Xavier board equipped with an ARM Carmel processor.

```

1 def im2col(t, ci, hi, wi, hf, wf, hp, wp, hs,
2   ws):
3     # P1) Define operation
4     I = te.placeholder((t, ci, h, w), name="I")
5
6     ho = (hi - hf + 2 * hp) // hs + 1
7     wo = (wi - wf + 2 * wp) // ws + 1
8
9     B = te.compute(
10        (ci, hf, wf, t, ho, wo),
11        lambda jci, jhf, jwf, jt, jh, jw:
12        te.if_then_else(te.any(
13            (jh*hs-hp)>=hi,
14            (jh*hs-hp)<0,
15            (jw*ws-wp)>=wi,
16            (jw*ws-wp)<0)),
17            0, I[jt, jci,
18                jh*hs-hp,
19                jw*ws-wp]),
20        name="B")
21
22     # P2) Prepare schedule
23     sched = te.create_schedule(B.op)
24
25     # P3) Generate code with LLVM backend
26     return tvn.build(sched, [I, B],
27                      target="llvm")

```

Fig. 6: TVM generator for the IM2COL transform.

For reference, we include in the evaluation the high performance realizations of this kernel in up-to-date releases of BLIS (version v0.8.1), OpenBLAS (version v0.3.19), and ARM Performance Libraries (ARMPL, version v21.1). Selecting the optimal loop to parallelize is out-of-scope for this work, so we perform all our experiments using a single core. In order to reduce variability, the processor frequency is fixed to 2.3 GHz, the process is bound to the same core, and the experiments are repeated a large number of times reporting next average results. Performance is measured in terms of billions of FP32 floating point operations per second, abbreviated as GFLOPS.

Given our interest in edge deep learning inference, the specific dimensions of the problems are selected as those that result from applying the IM2COL-approach to the convolution layers in the ResNet50 v1.5 DNN model with a batch size of 128 samples. As some layers share the same GEMM shapes, we report the results for them only once.

B. Cache configuration parameters

The performance of the BLIS realization of GEMM is strongly dictated by that of the micro-kernel and an appropriate selection of the cache configuration parameters: m_c, n_c, k_c . The optimal values for these three parameters depend on cache hardware features such as number of levels, size, set associativity, etc., as well as the micro-kernel dimension. Determining these values via brute force experimentation involves an expensive search across a large 3D space. For the ARM Carmel processor, BLIS employs a micro-kernel of size $m_r \times n_r = 8 \times 12$ and sets $m_c, n_c, k_c = 120, 640, 3072$.

Alternatively, one can use the analytical model in [10] 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 integrate a fair

number of automatically generated micro-kernels, of different dimensions, within the GEMM TVM-based routine.

C. Performance

Figure 7 details the impact of the TVM-generated micro-kernels on the performance of the GEMM BLIS-like TVM-based routine for the matrix multiplications with the dimensions corresponding to two selected layers of the ResNet model. These results show that a careful selection of the micro-kernel is critical to attain high performance for the GEMM TVM-based routine.

The two plots in the figure also show that the optimal micro-kernel is highly dependent on the problem dimension: the best rates are observed for the 4×16 and 4×28 micro-kernels for layer 006 and the 4×24 micro-kernel for layer 044. At this point we emphasize that not only the generation of the micro-kernels is automatized thanks to TVM, but the best micro-kernel for each problem case can be determined automatically via a few exploratory executions in a very reduced search space. Furthermore, as the optimal micro-kernel also depends on the specific data type, an approach that leverages TVM to automatically generate the micro-kernels provides additional significant advantages for the programmer.

Inspecting the realizations of GEMM in other libraries, in left plot of Figure 7, the TVM-based routine achieves 22.4 GFLOPS versus 13.3 GFLOPS for BLIS, 17.1 GFLOPS for OpenBLAS, and 11.9 for ARMPL. In the right plot of the same figure, the TVM-based routine achieves 26.1 GFLOPS versus 22.0 GFLOPS for BLIS (best library-based option).

The evaluation of the complete ResNet model, in Figure 8, shows a variety of results, with the TVM-based routine outperforming the BLIS realization by a large margin for layers 1–12, and 15–17 (40 cases out of the total 53 convolution layers in the model; it is competitive for layers 14 and 20 (3 cases); and it shows inferior performance for layers 13, 18, 19 (10 cases). Compared with OpenBLAS and ARMPL, the TVM-based solution is consistently better.

A closer analysis of the results, taking into account the GEMM operands' shapes determined by the corresponding layer reveals that the TVM-based routines deliver higher performance for highly rectangular cases, with m in the range 100352–1605632, and it is competitive when $m = 25088$, while BLIS is better choice for “squarish” problems, with m in the range of 6000. At this point we note that the actual processing cost of the Resnet50 v1.5 model is concentrated in those cases where m is in the range 100352–1605632 (47.8% of the total time), followed by $m = 25088$ (35.6% of the total time). In terms of absolute cost, this implies that the execution of all layers employing the TVM-generated routines requires 39.1 s compared with 48.0 s when using BLIS (and higher for OpenBLAS and ARMPL).

V. CONCLUDING REMARKS

We have presented a TVM-based solution to automatically obtain high performance realizations of GEMM and the convolution operator, especially tailored for inference tasks in edge devices, with reduced memory footprint, flexibility to adapt

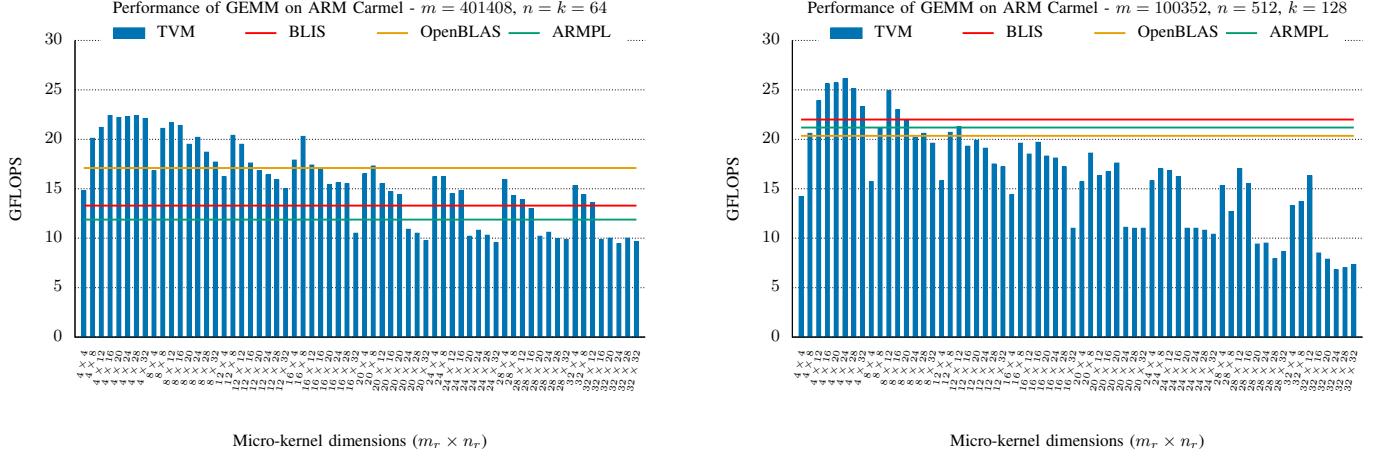


Fig. 7: Performance evaluation on ARM Carmel for layers 006 (left) and 044 (right) of ResNet50 v1.5.
Performance of GEMM on ARM Carmel - ResNet-50 v1.5

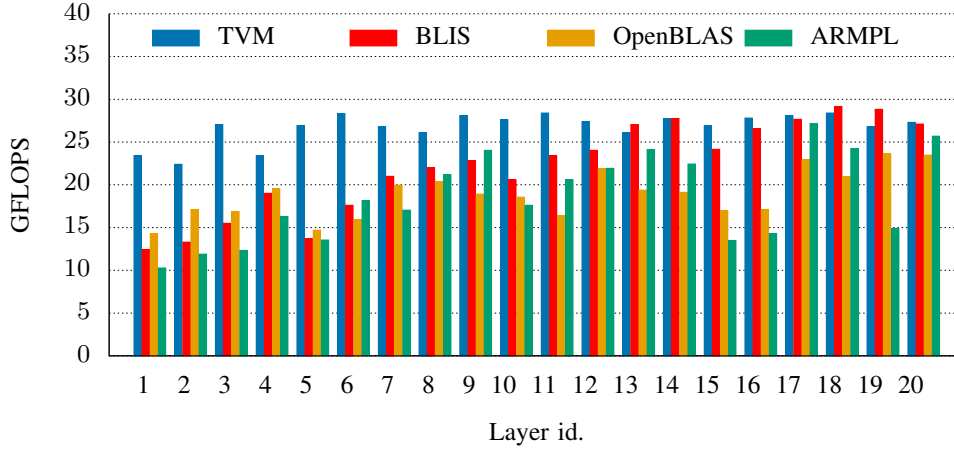


Fig. 8: Performance evaluation on ARM Carmel for all layers of ResNet50 v1.5.

the solution to distinct data types, enhanced maintainability, and fair portability to different processor architectures. Our solution departs from conventional library-based realizations of GEMM in that the full code is automatically generated and, therefore, completely portable. Compared with other JIT compilation frameworks, we mimic the techniques in the GotoBLAS2/BLIS/OpenBLAS2 algorithms for GEMM to obtain a blocked algorithm that performs an efficient utilization of the cache memories.

The experiments reveal that the TVM-based routine for GEMM outperforms the realizations of this operator in high performance libraries by a large margin for highly rectangular cases while being competitive for more squarish ones. At this point, we believe there is still a margin to optimize the TVM-based routine, while maintaining the automatic generation process, by exploring other variants of the BLIS family of algorithms and/or packing for other layers of the cache hierarchy. This is part of our future research plan.

REFERENCES

- [1] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.
- [2] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surv.*, vol. 52, no. 4, pp. 65:1–65:43, 2019.
- [3] K. Goto and R. A. van de Geijn, "Anatomy of a high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 12:1–12:25, 2008.
- [4] Z. Xianyi, W. Qian, and Z. Yunquan, "Model-driven level 3 BLAS performance optimization on Loongson 3A processor," in *2012 IEEE 18th Int. Conf. Parallel and Distributed Systems (ICPADS)*, 2012.
- [5] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, pp. 14:1–14:33, 2015.
- [6] D. L. Dutta and S. Bharali, "TinyML meets IoT: A comprehensive survey," *Internet of Things*, vol. 16, p. 100461, 2021.
- [7] T. Chen, L. Zheng, E. Q. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," *CoRR*, vol. abs/1805.08166, 2018. [Online]. Available: <http://arxiv.org/abs/1805.08166>
- [8] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: end-to-end optimization stack for deep learning," *CoRR*, vol. abs/1802.04799, 2018. [Online]. Available: <http://arxiv.org/abs/1802.04799>
- [9] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *International Workshop on Frontiers in Handwriting Recognition*, 2006.
- [10] T. M. Low *et al.*, "Analytical modeling is enough for high-performance BLIS," *ACM Trans. Math. Softw.*, vol. 43, no. 2, pp. 12:1–12:18, 2016.
- [11] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," 2018.