The Journal of Supercomputing
https://doi.org/10.1007/s11227-024-05955-8

Automatic generation of ARM NEON micro-kernels for matrix multiplication

Guillermo Alaejos[1] · Héctor Martínez[2] · Adrián Castelló[1] · Manuel F. Dolz[3] · Francisco D. Igual[4] · Pedro Alonso-Jordá[1] · Enrique S. Quintana-Ortí[1]

Accepted: 3 February 2024
© The Author(s) 2024

Abstract
General matrix multiplication (GEMM) is a fundamental kernel in scientific computing and current frameworks for deep learning. Modern realisations of GEMM are mostly written in C, on top of a small, highly tuned *micro-kernel* that is usually encoded in assembly. The high performance realisation of GEMM in linear algebra libraries in general include a single micro-kernel per architecture, usually implemented by an expert. In this paper, we explore a couple of paths to automatically generate GEMM micro-kernels, either using C++ templates with vector intrinsics or high-level Python scripts that directly produce assembly code. Both solutions can integrate high performance software techniques, such as loop unrolling and software pipelining, accommodate any data type, and easily generate micro-kernels of any requested dimension. The performance of this solution is tested on three ARM-based cores and compared with state-of-the-art libraries for these processors: BLIS, OpenBLAS and ArmPL. The experimental results show that the auto-generation approach is highly competitive, mainly due to the possibility of adapting the micro-kernel to the problem dimensions.

Keywords Matrix multiplication · ARM NEON · SIMD arithmetic units · High performance

1 Introduction

The general matrix multiplication (GEMM) is a key computational kernel on top of which a significant part of the *basic linear algebra subprograms* (BLAS) [1] is built [2, 3]. In addition, GEMM plays a fundamental role for convolutional (deep) neural networks that are prominent in computer vision tasks, as well as for transformers that are currently used in natural language processing [4, 5]. For these reasons, it is natural that considerable effort has been spent over the past decades to optimise GEMM in basically any computer architecture.

Extended author information available on the last page of the article

The performance of GEMM is strongly determined by the efficiency of a small architecture-specific component, known as the micro-kernel [6–8]. Most modern instances of BLAS contain a single micro-kernel per processor architecture, usually encoded in assembly by a computer architecture expert. However, the benefits of choosing among multiple micro-kernels have been illustrated for deep learning in [9] and for dense linear algebra, as well as scientific computing in general in [10].

This paper contributes towards the development of optimised versions of GEMM by presenting two methods to automatically generate competitive micro-kernels for ARM NEON (v8.2) processors equipped with single-instruction multiple-data (SIMD) vector units. This is especially interesting for processors from the same family that do not yet have a tuned instance of GEMM. In more detail, our work makes the following specific contributions:

- Initially, we generalise the initial solution proposed in [9] to leverage C++ templates in order to produce micro-kernels, based on vector intrinsics, *at compilation time*. This allows to deal more efficiently with "corner" cases that arise when the architecture cache configuration parameters are not integer multiples of the micro-kernel dimensions. In addition, the adoption of templates eases the generation of code for distinct data types using a single generator.
- Next, we take one step forward to directly produce assembly micro-kernels, using Python scripts. Compared with the previous solution, this method presents the advantage of enforcing a given order of the micro-kernel instructions that the compiler will not change.
- For three distinct ARM-based architectures and a collection of representative problem instances arising from practical deep learning applications, we demonstrate that

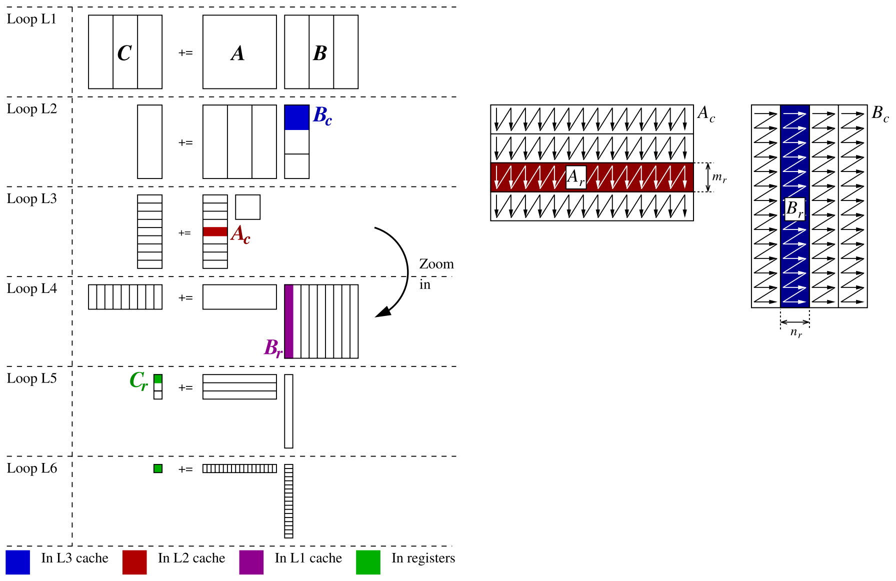


Fig. 1 Baseline algorithm for GEMM (top), data transfers across the memory hierarchy (bottom-left), and packing (bottom-right)

a GEMM routine that integrates the automatically generated micro-kernels provides competitive performance, on par or even outperforming the realisation of GEMM in highly tuned linear algebra libraries such as BLIS, OpenBLAS and ArmPL

At this point, it is worth noting that our work targets ARM NEON processors, yet the Python scripts can be easily adapted to target ARM SVE, AMD AVX2, Intel AVX512 or even the RISC-V vector extension.

The remainder of the paper is structured as follows: In Sect. 2, we briefly review the modern implementation of GEMM for current processors, with SIMD vector units and a multilayered memory hierarchy. In Sects. 3 and 4, we describe the two automatic generators introduced in this paper, respectively, relying on vector intrinsics and assembly. In Sect. 5, we evaluate the performance of the different solutions, and finally, in Sect. 6, we close the paper with the conclusion.

```
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);
  }}
```

2 Baseline implementation of GEMM

Consider 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 computational kernel, for conventional processor architectures with deep memory hierarchies, follow GotoBLAS [6] to encode it as five nested loops comprising two *packing routines* and a micro-kernel. Furthermore, for processors with SIMD vector units, the micro-kernel consists of an additional loop that performs an outer product per iteration. Figure 1 (top) displays the *baseline algorithm* for GEMM, identifying the six loops, the two packings, and the micro-kernel. Portable realisations of GEMM encode the five outermost loops and the two packing routines of the baseline algorithm in a high-level programming language such as C. In contrast, for high performance, the micro-kernel is an architecture-specific piece of code, usually encoded in assembly.

Cache hierarchy The three outermost loops of the baseline algorithm 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 m_c, n_c, k_c [11], favour that, during the execution of the micro-kernel, the buffers A_c, B_c, respectively, remain in the L2 and L3 cache memories.

Micro-kernel for SIMD processors The micro-kernel 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. Packings also ensure that the contents of buffers A_r, B_r are accessed with a unit stride during the execution of the micro-kernel, enabling the use of vector instructions to retrieve their contents. The arithmetic-to-memory access ratio (or arithmetic intensity [12]) of the micro-kernel is given by

$$\frac{2m_r n_r k_c}{(m_r + n_r) k_c} = \frac{2m_r n_r}{m_r + n_r} \quad \text{flops per memory access.}$$

Choosing large values for $m_r \approx n_r$ thus maximises this ratio. For the same reason, it is convenient to maximise the use of vector registers, without incurring into register spilling [11], by ensuring that

$$\underbrace{\left\lceil \frac{m_r}{\texttt{vl\_fp}} \right\rceil n_r}_{\text{Micro-tile of } C} + \underbrace{\left\lceil \frac{m_r}{\texttt{vl\_fp}} \right\rceil}_{\text{Column of } A_r} + \underbrace{\left\lceil \frac{n_r}{\texttt{vl\_fp}} \right\rceil}_{\text{Row of } B_r} = m_v\,(n_r + 1) + n_v \leq \texttt{vr},$$

where `vl_fp` denotes the number of floating point numbers (elements) that fit into a single vector register and `vr` is the number of vector registers.

Parallelism The multi-threaded parallelization of GEMM has been analysed for conventional multicore processors, modern many-threaded architectures, and asymmetric ARM-based processors in [7, 13, 14]. In those works, the parallelization of GEMM is approached by extracting parallelism from any of the loops `L1`, `L3`, `L4`, `L5`, or a combination of them. (Loops `L2` and `L6` are not convenient as they introduce race conditions.)

The parallelization technique is rather orthogonal to the micro-kernel: As the parallelization targets one (or more) of the five outermost loops of GEMM (`L1`–`L5`, except `L2`), and the micro-kernel only comprises the sixth loop (`L6`), any of the micro-kernels proposed in this work can be combined with the parallel approaches proposed in the literature. In consequence, and in order to keep the paper focused on the generation of the micro-kernel, we will omit the analysis of parallelism in the following.

3 ARM NEON micro-kernels for GEMM using vector intrinsics

In this section, we pursue the development of architecture-specific SIMD-based micro-kernels for ARM processors using vector intrinsics. However, instead of a manual development process, we demonstrate that it is feasible to generate the micro-kernels automatically, significantly easing this task while delivering fair performance.

For simplicity, hereafter we choose the 32-bit floating point (FP32) as the basic data type for all the routines/codes presented in the section (in C language, float). Furthermore, we target ARM NEON v8.2, for which the vector length is 128 bits (i.e. 16 bytes) so that, for FP32, vl_fp32=4 (FP32 numbers per vector register). The same ideas for automatic generation apply to other data types and SIMD-enabled processor architectures.

3.1 A simple generic micro-kernel

In [9], we took a significant step forward toward improving the portability and maintainability of the BLAS by introducing a "generic" (i.e. multiplatform) scheme that relies on C macros to abstract the vector data type and the basic vector intrinsics for load, store, and AXPY (scalar α times x plus y) update. In addition, when supported by the compiler, the generic micro-kernel can accommodate a micro-tile of any dimension $m_r \times n_r$ using arrays of vector registers.

```
#define mv      (mr/vl_fp32)
#define nv      (nr/vl_fp32)

gemm_ukernel_generic_intrinsics_mrxnr(int kc, float *Ar, float *Br,
                                      float *C,  int ldC) {
  // mr x nr micro-kernel with C resident in regs.
  // Inputs:
  //   - kc:  k-dimension of micro-kernel
  //   - Ar:  packed micro-panel of Ac, with leading dimension mr
  //   - Br:  packed micro-panel of Bc, with leading dimension nr
  //   - C:   micro-tile of C stored in column-major order
  //   - ldC: leading dimension of C

  int        iv, j, jv, kr;
  vregister Cr[mv][nr]      // Micro-tile of C
            ar[mv], br[nv]; // Single column/row of Ar/Br

  vinit();

  // Load the micro-tile of C into vector registers
  for (j=0; j<nr; j++)
    for (iv=0; iv<mv; iv++)
      vload(Cr[iv][j], &C[j*ldC+iv*vl_fp32]);

  for (kr=0; kr<kc; kr++) { // Main loop
    // Load the kr-th column/row of Ar/Br into vector registers
    for (iv=0; iv<mv; iv++) {
      vload(ar[iv], Ar); Ar += vl_fp32; }
    for (jv=0; jv<nv; jv++) {
      vload(br[jv], Br); Br += vl_fp32; }

    // Update the micro-tile with AXPYs
    for (iv=0; iv<mv; iv++)
      for (jv=0; jv<nv; jv++)
        for (j=0; j<vl_fp32; j++)
          vupdate(Cr[iv][jv*vl_fp32+j], ar[iv], br[jv], j);
  }

  // Store the micro-tile in memory
  for (j=0; j<nr; j++)
    for (iv=0; iv<mv; iv++)
      vstore(&C[j*ldC+iv*vl_fp32], Cr[iv][j]);
}
```

Listing 1: Generic micro-kernel that operates with an mr ×nr micro-tile using vector intrinsics.

```c
// Macros for ARM NEON
#define vl_fp32 4 // FP32 numbers per vector register

#define vregister float32x4_t
#define vinit()
#define vload(vreg, mem)  vreg = vld1q_f32(mem)
#define vstore(mem, vreg) vst1q_f32(mem, vreg)
#define vupdate(vreg1, vreg2, vreg3, j) \
                vreg1 = vfmaq_laneq_f32(vreg1, vreg2, vreg3, j)
```

Listing 2: C macros to customise the generic micro-kernel for ARM NEON (v8.2) and FP32.

In order to present the solution in [9], assume for simplicity that m_r, n_r are both integer multiples of `vl_fp32`. We then need $m_v\, n_r = (m_r/\texttt{vl\_fp32})\, n_r$ vector registers to store the micro-tile C_r; m_v for a single column of A_r; and $n_v = n_r/\texttt{vl\_fp32}$ for a single row of B_r. Listing 1 displays our original generic micro-kernel, where we highlight a couple of details:

- Prior to the main loop, indexed by `kr` (line 25), we load the contents of the $m_r \times n_r$ micro-tile of C into the array of vector registers `Cr` via two nested loops (lines 21–23). The transfer in the opposite direction, from the vector registers back to the memory, is carried out after the main loop (lines 40–42).
- At each iteration of the main loop, a column of m_r elements of A_r and a row of n_r elements of B_r are first loaded into the appropriate vector registers (lines 27–28 for `ar` and lines 29–30 for `br`, respectively). These elements then participate in the update of the micro-tile stored in `Cr` (lines 33–36).

The generic micro-kernel in Listing 1 is customised for ARM NEON (v8.2) and FP32 using the C macros displayed in Listing 2.

3.2 Evolving the generic micro-kernel

For the GEMM baseline algorithm, when m_c, n_c are not integer multiples of m_r, n_r, respectively, a significant benefit can be obtained by developing specialised micro-kernels which employ SIMD instructions to update the “corner” cases. To illustrate this, imagine we have adopted a 8×8 micro-kernel as the baseline. Unfortunately, we will encounter certain cases where it is necessary to process a micro-tile of smaller dimensions, for example 4×8. In that particular case, it may be more efficient to employ a 4×8 micro-kernel with vector instructions to update this smaller micro-tile than to employ a scalar (i.e. non-SIMD) routine the baseline 8×8 micro-kernel. Furthermore, the corner cases where the any or both micro-tile dimensions are not integer multiples of the vector register length (e.g. 3×7) can be dealt via a micro-kernel of dimensions immediately superior that are integer multiples (e.g. $m_r \times n_r = 4 \times 8$), exploiting the fact that the buffers A_c, B_c will accommodate this excess in the dimensions and using scalar instructions to load and store only the necessary elements of C_r.

```
// Micro-kernel using templates
template<int mr, int nr, typename T>
void gemm_ukernel_generic_intrinsics_mrxnr(int kc, T *Ar, T *Br,
                                           T *C, int ldC) {
  // mr x nr micro-kernel with C resident in regs.
  constexpr int mv = mr / vl_fp32, nv = nr / vl_fp32;
  int iv, j, jv, pr, baseA = 0, baseB = 0;
  vregister Cr[mv * nr],     // Macro-tile of C
            ar[mv], br[nv]; // Single column/row of Ar/Br
  vinit();
  // Load the micro-tile of C into vector registers
  vloadC_mv_nr<nr, mv, nr, vl_fp32>(Cr, C, ldC);

  for (pr=0; pr<kc; pr++) {
     // Load the pr-th column/row of Ar/Br into vector registers
     vload_<mv, vl_fp32>(ar, &Ar[baseA]); baseA += mr;
     vload_<nv, vl_fp32>(br, &Br[baseB]); baseB += nr;

     // Update micro-tile with AXPYs
     vupdate_vl_nv_mv<mv, nv, vl_fp32, nr>(Cr, ar, br);
  }
  // Store the micro-tile in memory
  vstoreC_mv_nr<nr, mv, nr, vl_fp32>(C, Cr, ldC);
}

// Instantiation examples for different micro-kernel sizes
gemm_ukernel_generic_intrinsics_mrxnr<4, 4>(kc, Ar, Br, C, ldC);
gemm_ukernel_generic_intrinsics_mrxnr<4, 8>(kc, Ar, Br, C, ldC);
gemm_ukernel_generic_intrinsics_mrxnr<8, 4>(kc, Ar, Br, C, ldC);
```

Listing 3: Generic micro-kernel that leverages C++17 templates to operate with any $m_r \times n_r$ micro-tile using vector intrinsics.

To address these scenarios, the top part of Listing 3 presents an enhanced version of the generic micro-kernel in Listing 1 that utilises C++ templates to facilitate the generation of a collection of micro-kernels for any combination of m_r and n_r. To achieve this, a set of auxiliary template functions are responsible for unrolling the micro-kernel loops using recursive integer template parameters and constant conditional (static-if) expressions specific to the C++17 standard (see Listing 10 in the appendix for details). During compilation, these functions are evaluated to generate the appropriate instructions within the loop body based on the values of m_r and n_r. For instance, in the main function depicted at the bottom part of Listing 3, we instantiate GEMM micro-kernels for sizes 4×4, 4×8, and 8×4.

In conclusion, as in the case of the initial generic micro-kernel, this template-based version produces customised code for any architecture. It also accommodates the generation of code for a family of micro-kernels of different sizes at compile time.

4 ARM NEON micro-kernels for GEMM using assembly

In this section, we also address automatic generation of micro-kernels, but now targeting architecture-specific SIMD-based routines for ARM NEON processors using assembly instead of vector intrinsics.

4.1 Simple micro-kernels using ARM NEON assembly

In Listing 4, we show a simple micro-kernel of dimension $m_r \times n_r = 4 \times 4$ for FP32 data and encoded using ARM NEON assembly. The routine receives the same five parameters as its counterpart with vector intrinsics in Listing 1: The dimension k_c; address pointers to A_r, B_r and C; and the leading dimension `ldC`. Note the connection between the two versions of the micro-kernel, emphasised with the use of the same comments for those blocks of the two codes that perform analogous functions. The assembly routine proceeds as follows:

- From the address pointer `C` (≡ `C00`) to the appropriate entry of the matrix C, the routine initialises the pointers `C01`, `C02`, `C03` to the remaining three columns of the 4×4 micro-tile taking into account the matrix column stride (lines 9–11).
- Prior to the main loop, four columns of C, each consisting of four FP32 numbers, are loaded into four vector registers: `Crq00`–`Crq03`, using the assembly SIMD instruction `ldr`, which has a analogous function to that of the vector intrinsics `vld1q_f32` (lines 13–16). After the loop, the contents of these vector registers are stored back into the matrix C using the assembly SIMD instruction `str`, with an analogous function to that of the vector intrinsics vst1q_f32 (lines 33–36). Note that `CrqXY` and `CrvXY` refer to the same register, but are referenced depending on whether the register is, respectively, involved in a memory (load/store) instruction or an arithmetic operation.
- At each iteration of the main loop, the routine loads one column of A_r into a vector register `arq0` (line 19), one row of B_r into a vector register `brq0` (line 21), and updates `Crv00`–`Crv03` via four AXPYs (lines 24–27) using the assembly SIMD instruction `fmla`, which performs a vector fused multiply-add (functionally equivalent to the vector intrinsic `vfmaq_laneq_f32`.

```
.text
.align       5
.global      gemm_ukernel_NEON_asm_4x4

gemm_ukernel_NEON_asm_4x4:
// Save variables to stack
// Omitted for brevity

add C01, C00, ldC           // Pointers
add C02, C01, ldC           // to columns of C,
add C03, C02, ldC           // column stride=ldC

ldr Crq00, [C00]            // Load the
ldr Crq01, [C01]            // micro-tile of C
ldr Crq02, [C02]            // into vector
ldr Crq03, [C03]            // registers

.LOOP_4x4:                  // Main loop
ldr arq0,   [Ar]            // Load the kr-th
add Ar, Ar, #shift          // column/row
ldr brq0,   [Br]            // of Ar,Br
add Br, Br, #shift

fmla Crv00, arv0, brv0[0] // Update the
fmla Crv01, arv0, brv0[1] // micro-tile
fmla Crv02, arv0, brv0[2] // with AXPYs
fmla Crv03, arv0, brv0[3]

sub kc, kc, 1               // Prepare for
cmp kc, 0                   // the next
b.ne .LOOP_4x4              // iteration

str Crq00, [C00]            // Store the
str Crq01, [C01]            // micro-tile
str Crq02, [C02]            // in memory
str Crq03, [C03]

END:
// Restore variables from stack
// Omitted for brevity
ret
```

Listing 4: Micro-kernel that operates with a 4×4micro-tile using ARM NEON assembly.

```
// Inputs
#define kc     x0
#define Ar     x1
#define Br     x2
#define C      x3
#define ldC    x4

// ARM NEON
// vl * sizeof
// datatype (FP32)
#define shift 16

// Addresses for
// remaining
// columns of C
#define C00    C
#define C01    x5
#define C02    x6
#define C03    x7


// Vector registers
// for micro-tile
// of C
#define Crq00 q0
#define Crv00 v0.4s
#define Crq01 q1
#define Crv01 v1.4s
#define Crq02 q2
#define Crv02 v2.4s
#define Crq03 q3
#define Crv03 v3.4s


// Vector registers
// for column/row
// of Ar,Br
#define arq0   q4
#define arv0   v4.4s
#define brq0   q5
#define brv0   v5.s
```

Listing 5: Macros forthe micro-kernel usingARM NEON assembly.

The 4×4 micro-kernel in Listing 4 exhibits a regular structure that is possible to generalise to many other dimensions. (This is demonstrated, for example, with the excerpt of code in Listing 11 provided in the appendix, which corresponds to the main loop of a 12×8 assembly micro-kernel for ARM NEON and FP32 data.) Concretely, we identify the following characteristics which are independent of the micro-kernel dimensions:

1. The contents of the micro-tile are retrieved from memory into vector registers prior to the loop and written from there back to memory after it.
2. The micro-kernel employs $m_v \times n_r$ vector registers to keep the contents of the $m_r \times n_r$ micro-tile.
3. At each iteration, the loop loads the contents of one column of A_r and one row of B_r, and then uses them to update the micro-tile once.

This similarity is the basis that motivates the automatic generation of assembly micro-kernels.

4.2 A python generator of micro-kernels using ARM NEON assembly

A basic generator The regularity of the basic micro-kernels can be leveraged to automatically elaborate their code using the Python routine in Listing 6. (Indeed, the ARM NEON assembly codes for the 4×4 and 12×8 micro-kernels in Listings 4, 5, and 11 were obtained using this generator.) Inspecting the instructions of the generator, we can easily identify the different parts that produce the code fragments for the load/store of C, A_r, B_r, and the arithmetic. Note that this generator assumes that C is stored in column-major order. In case C is stored in row-major order, we can still use the same micro-kernel by swapping the roles of A_r and B_r and adjusting the leading dimension of C accordingly.

```python
def generator_gemm_ukernel_NEON_asm_mrxnr(mr, nr, vl, data_type):
  # mr x nr micro-kernel with C resident in regs.
  # Inputs:
  #   - mr, nr: columns, rows of micro-tile
  #   - vl: vector length (bits of vector register/bits of data_type)
  #   - data_type: bytes of data type (e.g., 4 for FP32)

  # Macros for routine parameters, defines and header function
  # Omitted for brevity
  shift = vl*data_type
  #
  for c in range(1, nr): #Pointers to columns of C, column stride=ldC
    fout.write(f"add C0{c}, C0{c-1}, ldC\n")
  #
  for c in range(0, nr):                          # Load the micro-tile
    fout.write(f"ldr Crq0{c}, [C0{c}]\n")  # of C into vector regs.
    addr = shift
    for r in range(1, mr // vl):
      fout.write(f"ldr Crq{r}{c}, [C0{c}, #{addr}]\n")
      addr += shift
  #
  fout.write(f".LOOP_{mr}x{nr}:\n")        # Main loop
  addr = shift
  for r in range(0, mr // vl):                    # Load the pr-th
    fout.write(f"ldr arq{r},  [Ar]\n")        # column of Ar
    fout.write(f"add Ar, Ar, #{addr}\n")      # into vector regs.
  #
  for c in range(0, nr // vl):                    # Load the pr-th
    fout.write(f"ldr brq{c},  [Br]\n")        # row of Br
    fout.write(f"add Br, Br, #{addr}\n")      # into vector regs.
  #
  for r in range(0, mr // vl):                    # Update the micro-tile
    for c in range(0, nr // vl):                  # with AXPYs
      for v in range(0, vl):
        Cp = c*vl+v
        fout.write(f"fmla Crv{r}{Cp}, arv{r}, brv{c}[{v}]\n")
  #
  fout.write(f"sub kc, kc, 1\n")                   # Prepare for the
  fout.write(f"cmp kc, 0\n")                       # next iteration
  fout.write(f"b.ne .LOOP_{mr}x{nr}\n")
  #
  for c in range(0, nr):                          # Store the micro-tile
    fout.write(f"str Crq0{c}, [C0{c}]\n")  # in memory
    addr = shift
    for r in range(1, mr // vl):
      fout.write(f"str Crq{r}{c}, [C0{c}, #{addr}]\n")
      addr += shift
```

Listing 6: Micro-kernel generator that operates with an mr × nr micro-tile using ARM NEON assembly.

The simple generator in Listing 6 builds a micro-kernel that operates with an $m_r \times n_r$ micro-tile of C, assuming there are enough vector registers for this. This can result in a compilation error if the number of utilised vector registers exceeds the maximum. This would be the case, for example, of a 16×8 micro-kernel, which would require 32 vector registers for the micro-tile of C, 4 for the column of A_r, and 2 for the row of B_r, for a total of 38. In the actual generator, this type of situations are avoided with a simple logic test.

Automatic generation of advanced micro-kernels The basic generator has been extended to produce more sophisticated micro-kernels enhanced with advanced techniques such as loop unrolling and software pipelining [15]. For example, the former can be accommodated in the 4×4 micro-kernel using the macro in Fig. 7, which comprises the loads of A_r, B_r and the arithmetic in Listing 4 (lines 19–27).

```
#define BODY_LOOP_4x4(aq, bq, av, bv)      \
  ldr aq, [Ar]           // Load the pr-th \
  add Ar, Ar, #shift     // column/row     \
  ldr bq, [Br]           // of Ar,Br       \
  add Br, Br, #shift                       \
                                           \
  fmla Crv00, av, bv[0] // Update the      \
  fmla Crv01, av, bv[1] // micro-tile      \
  fmla Crv02, av, bv[2] // with AXPYs      \
  fmla Crv03, av, bv[3]
```

Listing 7: Macros for the 4×4 micro-kernel.

We can then generate the main loop of the micro-kernel with an unrolling factor of 4 by replicating the loop body that number of times, via the macro, as shown in Listing 8. For simplicity, we do not show here how to extend the code for the cases where k_c is not an integer multiple of 4.

```
.LOOP_4x4:                                // Main loop
BODY_LOOP_4x4(arq0, brq0, arv0, brv0) // Unroll basic loop body 4 times
BODY_LOOP_4x4(arq0, brq0, arv0, brv0)
BODY_LOOP_4x4(arq0, brq0, arv0, brv0)
BODY_LOOP_4x4(arq0, brq0, arv0, brv0)

sub kc, kc, 4                             // Prepare for the next iteration
cmp kc, 0
b.ne .LOOP_4x4
```

Listing 8: Main loop of the 4×4 micro-kernel unrolled with a factor 4.

A complementary technique that can be integrated in the automatic generator is *Software pipelining* is a complementary technique which, during an iteration, pre-loads the data that will be utilised in the “next” iteration of the main loop, separating these memory accesses from the arithmetic operations where the data is utilised. The excerpt of code in Listing 9 combines software pipelining and loop unrolling with a factor of 4. Again, for simplicity, we do not discuss the code required to cover the final iterations or the cases where k_c is not an integer multiple of 4.

```
ldr arq0n,   [Ar]                        // Pre-load the pr-th
add Ar, Ar, #shift                       // column/row
ldr brq0n,   [Br]                        // of AR, Br
add Br, Br, #shift

.LOOP_4x4:                               // Main Loop
BODY_LOOP_4x4(arq0,  brq0,  arv0n, brv0n)
BODY_LOOP_4x4(arq0n, brq0n, arv0,  brv0)
BODY_LOOP_4x4(arq0,  brq0,  arv0n, brv0n)
BODY_LOOP_4x4(arq0n, brq0n, arv0,  brv0)

sub kc, kc, 4                            // Prepare for the
cmp kc, 4                                // next iteration
b.ne .LOOP_4x4
```

Listing 9: Main loop of the 4×4 micro-kernel unrolled with a factor 4 and with software pipelining.

Dealing with "corner" cases Our Python generator for assembly micro-kernels is not oblivious to corner cases that were already discussed in the case of vector intrinsics. Indeed, the Python generator takes this into account and, when asked to produce an $m_r \times n_r$ micro-kernel, it actually builds the requested one plus a full collection of smaller micro-kernels to tackle other micro-tile dimensions. In addition, there is a complete logic that is integrated into the GEMM routine and invokes the micro-kernel that better matches the specific dimensions of each corner case.

5 Experimental evaluation

In this section, we assess the performance of the GEMM realisations embedding the automatically generated micro-kernels. For reference, we include in the comparison an evaluation of the GEMM in optimised instances of BLIS, OpenBLAS and ArmPL for the target platforms.

5.1 Problem cases

Much of the interest of our work lies in the fact that the matrix multiplication kernel is the backbone of the convolution operation, once the IM2COL (or IM2ROW) transform casts this operator into a GEMM. The convolution is found in well-known neural network layers for signal processing (including computer vision) and, moreover, bears most of the computational weight of model execution. For example, in [16] we report that the convolution layers in the ResNet-50 v1.5 model combined with ImageNet can consume between 45% and 87% of the inference time, depending on the optimisations that are applied. Thus, given the interest in deploying deep learning technologies, the dataset for the experimentation here includes matrix multiplications with their dimensions determined by the application of IM2COL to the convolution layers in the neural networks ResNet-50 v1.5 [17] and GoogleLeNet [18], combined with the ImageNet dataset. In the experiments, the batch size is set to 1 sample, reflecting a latency-oriented scenario [16, 19].

5.2 Hardware setup

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

- An NVIDIA Cortex-A78AE processor, embedded in the NVIDIA Jetson AGX Orin board, with a 64-KB L1 data cache, a 256-KB L2 cache, a 2-MB L3 cache, and a 32-GB LPDDR5 memory.
- 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.
- 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.

These target systems, listed from highest to lowest computational power, are representative of the type of equipment that can be used to run machine learning inference workloads.

In order to reduce variability in the experiments, the frequency of the processor cores is fixed in all cases. A single core is employed in the three architectures, with a thread bound to it. All experiments are carried out in IEEE FP32 arithmetic, and they 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) or, in the final part of the section, in execution time (s).

5.3 Software setup

We focus on the performance gains that can be obtained when leveraging specific micro-kernels for the convolution operators in the ResNet and GoogleLeNet neural networks. The goals are to show the performance obtained in each layer with our GEMM using the best micro-kernel for that layer and to demonstrate that, by choosing the appropriate computational kernel, i.e. the GEMM with the appropriate micro-kernel dimensions and optimisation techniques, it is possible to obtain performance similar, or even superior in many cases, to that offered by the implementation of GEMM in optimised libraries. Concretely, for reference, the comparison includes data for the GEMM realisations contained in BLIS (version v0.8.1) [13], OpenBLAS (version v0.3.19) [8], and ArmPL (version v21.1) [20].

5.4 Performance per layer

Figures 2, 2 and 4 report the performance of the five GEMM implementations on the three platforms, for the individual convolutional layers present in the two convolutional neural networks (CNNs). For the two most powerful platforms, NVIDIA Jetson AGX Orin and Xavier, the results are similar, with our automatically generated micro-kernels being, in a large majority of layers, among the top-3 best options; and in many cases, offering the best choice. The results are quite different for the

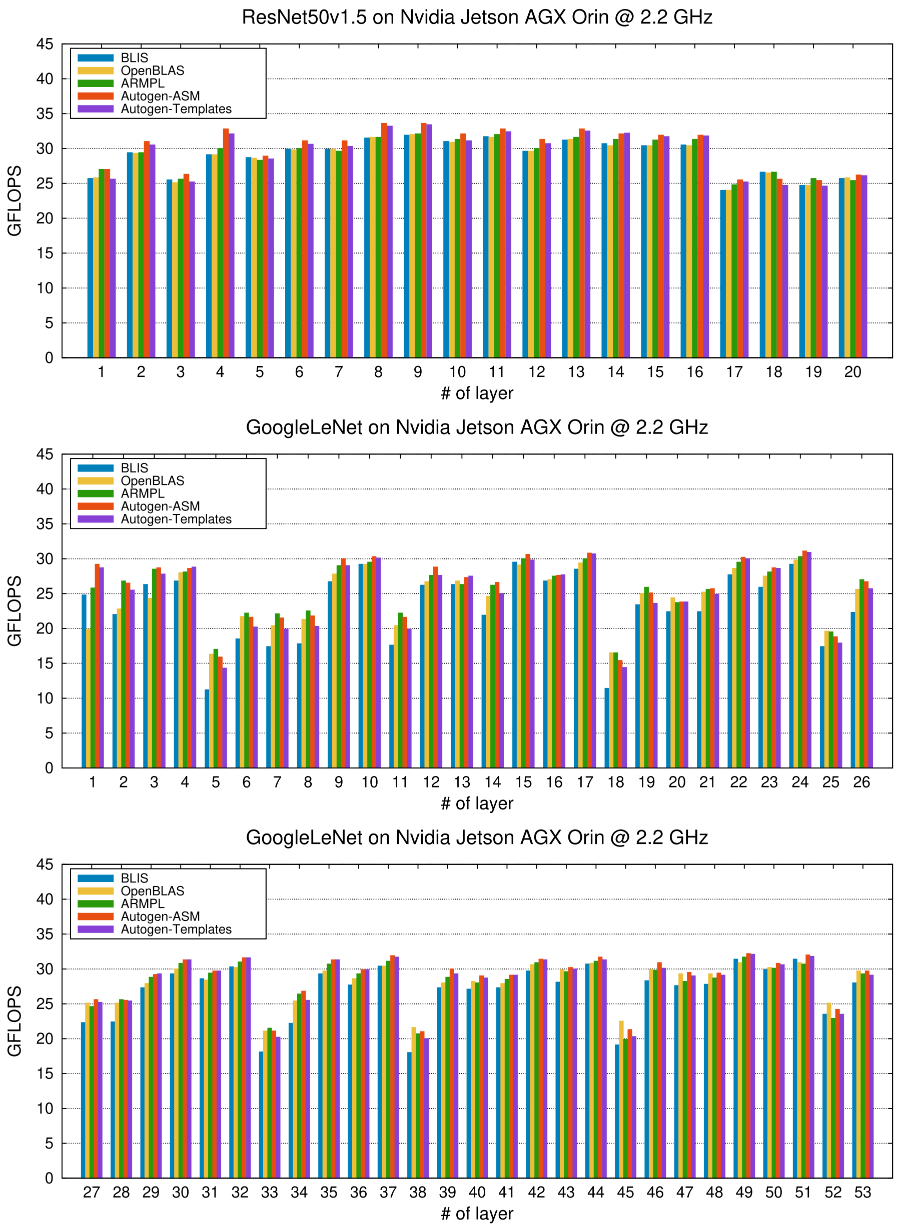


Fig. 2 Performance in GFLOPS of each representative convolutional layer of neural networks ResNet-50 v1.5 (top) and GoogleLeNet (middle and bottom) on Nvidia Jetson AGX Orin

NVIDIA Jetson Nano, where BLIS and OpenBLAS present superior performance. In consequence, we comment these two cases separately.

NVIDIA Jetson AGX Xavier/Orin As the number of results is large, we will focus our comments on one scenario that we believe is representative of the remaining

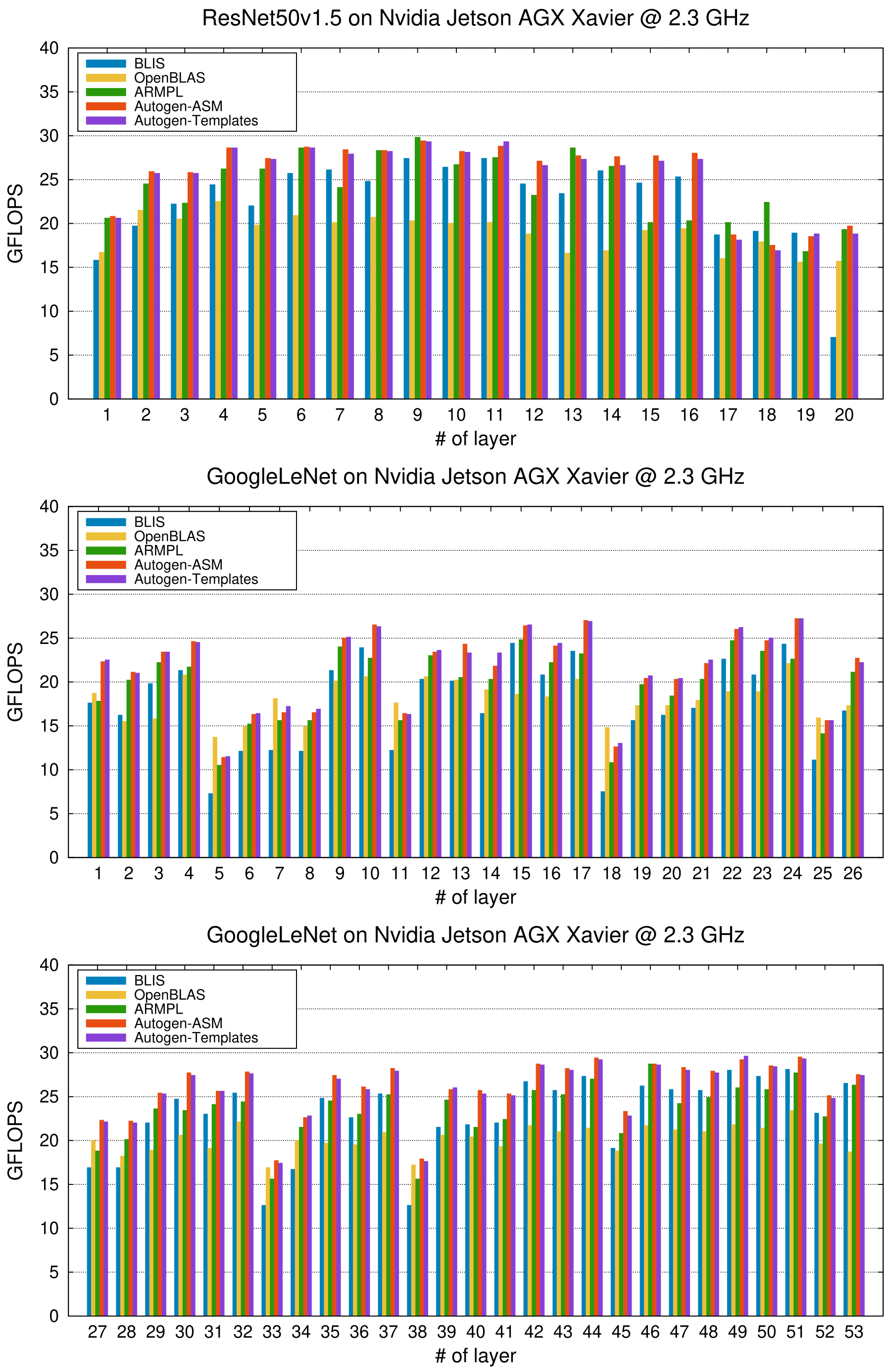


Fig. 3 Performance in GFLOPS of each representative convolutional layer of neural networks ResNet-50 v1.5 (top) and GoogleLeNet (middle and bottom) on Nvidia Jetson AGX Xavier

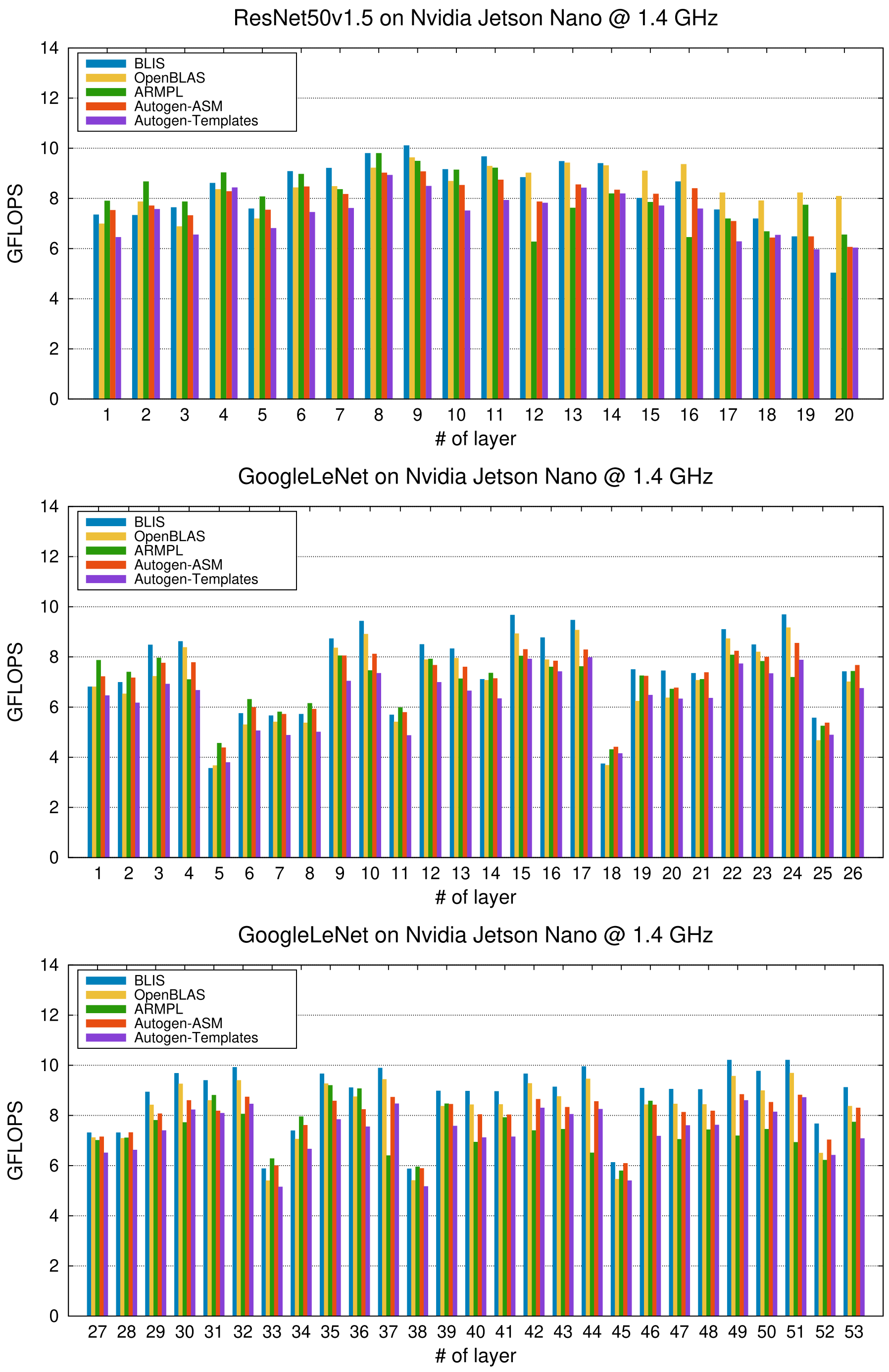


Fig. 4 Performance in GFLOPS of each representative convolutional layer of neural networks ResNet-50 v1.5 (top) and GoogleLeNet (middle and bottom) on Nvidia Jetson AGX Nano

cases on these two platforms and both CNN models. In particular, we describe in detail the outcome of the execution of the convolutional layers in ResNet-50 v1.5 on the NVIDIA Jetson AGX Xavier; see the top plot in Fig. 3. The results there show that the pair of solutions that integrate our automatically generated micro-kernels (labeled as `Autogen-ASM` and `Autogen-Templates`) outperform the library-based implementations (labeled as `BLIS`, `OpenBLAS`, and `ARMPL`) for layers #2 to #5, #7, #10 to #12, #15, #16 (10 cases out of 20, that is, 50%). In contrast, `ARMPL` is the best option for layers #13, #17, #18 (3 cases out of 20, 15%). For layers #1, #6, #8, #9, #14, #20 (6 cases, 30%), the best performance is attained by `Autogen-ASM/Templates` and `ARMPL`, with little differences between them. Finally, in one layer, #19, `Autogen-ASM/Templates` and `BLIS` offer similar performance, superior to that of the other alternatives. In summary, our automatically generated codes deliver the highest performance in all except three layers, where they are outperformed by `ARMPL`.

In order to characterise these results, we first need to link them with the dimensions of the GEMM associated with each layer; see Table 1. Depending on the value of *m*, a classification can be established into four groups of problems: large, medium, small, and tiny. This clustering offers a characterisation of performance for the two

Table 1 Dimension of the GEMM problems associated with the convolutional layers in ResNet-50 v1.5+ImageNet for batch size $b = 1$

Layer	n	m	k	Best option
1	64	12,544	147	`Autogen/ARMPL`
2	64	3136	64	Autogen
3	64	3136	576	Autogen
4	256	3136	64	Autogen
5	64	3136	256	Autogen
6	128	3136	256	Autogen/ARMPL
7	128	784	1152	Autogen
8	512	784	128	Autogen/ARMPL
9	512	784	256	Autogen/ARMPL
10	128	784	512	Autogen
11	256	784	512	Autogen
12	256	196	2304	Autogen
13	1024	196	256	ARMPL
14	1024	196	512	Autogen/ARMPL
15	256	196	1024	Autogen
16	512	196	1024	Autogen
17	512	49	4608	`ARMPL`
18	2048	49	512	`ARMPL`
19	2048	49	1024	`Autogen/BLIS`
20	512	49	2048	`Autogen/ARMPL`

The horizontal lines divide the layers into four groups depending on the dimension *m*: large, medium, small, and tiny. The rightmost column reports the best GEMM implementation(s), for that particular layer of the CNN model, on the NVIDIA Jetson AGX Xavier groups in the middle. Concretely, for medium m and $k \geq 512$, `Autogen-ASM/Templates` offers the best performance while, for the same group and smaller k, the performance of that option is similar to that of `ARMPL`. Similarly, for small m and large k(≥1,024) `Autogen-ASM/Templates` is again the best but its performance tends to decay with respect to `ARMPL` as k decreases. In general, for large m the best automatically generated micro-kernel is $m_r \times n_r = 20 \times 4$, moving towards other variants with smaller m_r ($16 \times 4, 12 \times 8, 8 \times 12$) as m becomes also smaller.

Explaining the different behaviour of the GEMM instances requires a careful case-by-case analysis as it is the consequence of a combination of factors related to the micro-kernel, that we revise in the following list:

- **Micro-kernel.** The dimensions of the micro-kernel and the ratio m_r/n_r determine its arithmetic intensity [12] and, therefore, its performance under ideal conditions. For example, in a separate experiment, we could determine that, on the NVIDIA Jetson AGX Xavier, the larger micro-kernels automatically generated with our tool ($m_r \times n_r$= 8×12, 12×8, 4×16, 4×20, 16×4, and 20×4, are compute-bound while the smaller ones are bounded by the `L2` cache bandwidth.
- **Dimensions** n_r **and** k. According to the GotoBLAS scheme, an $k_c \times n_r$ micro-panel of B should populate a significant fraction of the `L1` cache, proportional to the ratio $n_r/(m_r + n_r)$ [11]. Now, looking to the dimension k of the problems in Table 1, we observe that $k_c \leq k$ and, therefore, a large value n_r, which depends on that dimension of the micro-kernel, can yield a more efficient use of `L1` cache even when $k_c = k$ is small.
- **Packing routines**. Fourth, the packing routines help to reduce cache eviction but, at the same time, introduce a certain overhead, which can be significant in case there is not enough reuse of the buffers A_c, B_c. As the matrices A/B can be stored in either row- or column-major order, and they have to be packed into narrow micro-panels, respectively, of m_r rows/n_r columns, the dimension of the micro-kernel interacts with the storage format to impact the cost of the packing procedures.
- *Edge cases*. The cache configuration parameters m_c, n_c, k_c, and the micro-kernel dimensions m_r, n_r decompose the GEMM problem into a collection of packing operations and micro-kernels of different sizes. When m is not an integer multiple of m_c and/or the latter is not an integer multiple of m_r, these edge cases could benefit from specialised routines, which are not always efficiently implemented. The same applies to the trio n, n_c, n_r; and the pair k, k_c.

Note that in this list we have omitted the cache configuration parameters. These are set differently for each library/platform and obviously play a role on the performance as they have a direct impact on the utilisation of the cache hierarchy. In summary, explaining how all these factors interact to determine the overall performance of a specific GEMM implementation, for a particular problem dimension, is very interesting, but also quite complex, especially for sophisticated libraries implemented by others.

NVIDIA Jetson Nano. The performance of the five routines on this system show a wider variability of the best option. Our GEMM routine does not stand out as optimal yet, in general, it does not lose track with respect to the best option. Concretely, we find layers for which some routines, like ARMPL, perform well for many layers (#7, #20 of ResNet-50 v1.5) but quite poorly for others (layers #12, #16). OpenBLAS is better tuned for the core in Nano than for that in Xavier. The regularity in performance of Autogen-ASM is one of the most remarkable aspects of our proposal compared with the use of optimised libraries.

The superior performance of BLIS and OpenBLAS for the NVIDIA Jetson Nano is mainly due to a couple of factors: 1) BLIS and OpenBLAS contain micro-kernels with extensive use of hardware prefetching; and 2) BLIS and OpenBLAS provide vectorised version of the packing routines. From our observations, while these two factors have no major effect on the two other platforms, on a resource-constrained system such as the NVIDIA Jetson Nano, they explain the superior performance of the instances of GEMM in these libraries.

5.5 Aggregated time

The GFLOPS rate provides a normalised metric to evaluate the performance of the different implementations of GEMM, but it does not reveal the contribution of the layers to the total execution time and therefore the relevance of the differences. The final experiment, with results shown in Figs. 5, 6, and 7, shows the aggregated time on the two CNN models and the three platforms. To reflect a realistic execution, we report the execution time for all the convolution layers in the CNN models, not only those with different dimensions, as was the case of the previous experiments in this section.

The results show that, when comparing the option with automatically generated micro-kernels to the best library-based solution, the overall gain for the NVIDIA Jetson AGX Orin is small, between 2% and 3% depending on the CNN model; it is larger for the NVIDIA Jetson AGX Xavier, between 8% and 12%. Finally, in the NVIDIA Jetson Nano, the loss is between 6% and 10%.

6 Conclusion

We have proposed two approaches to automatically generate GEMM micro-kernels that mimic the encoding effort done by an expert, relieving this programmer from a significant part of the effort required in (the initial steps of) this error-prone task. Concretely, our generators produce either a C code with vector intrinsics or directly an assembly routine, for any data type and micro-kernel dimensions. Furthermore, they integrate high performance techniques such as loop unrolling and software pipelining.

Our experimental study shows the benefits of the automatic solution in comparison with optimised implementations of GEMM in state-of-the-art libraries for three ARM-based processors and a representative collection of problem instances.

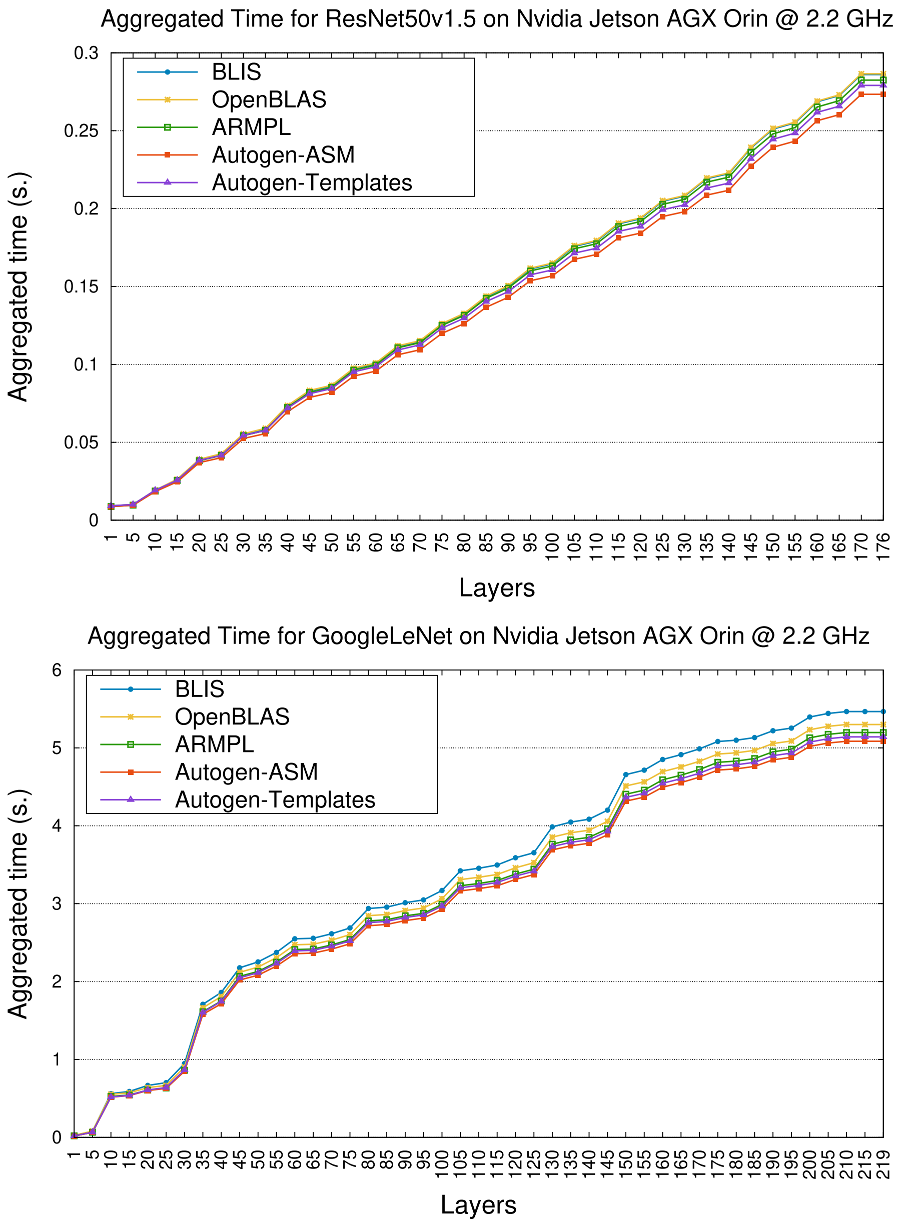
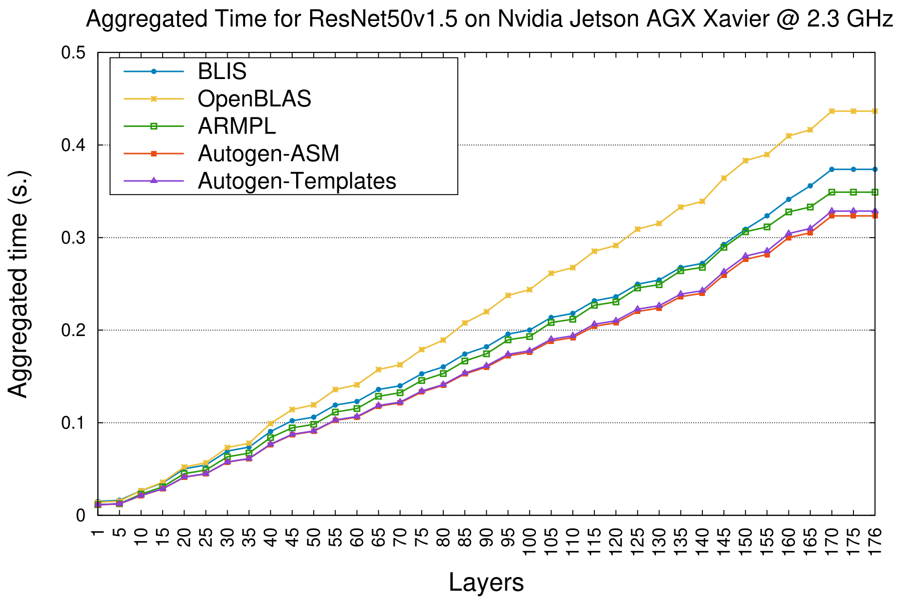


Fig. 5 Aggregated time of the convolutional layers of ResNet-50 v1.5 (top) and GoogleLeNet (bottom) on the NVIDIA Jetson AGX Orin



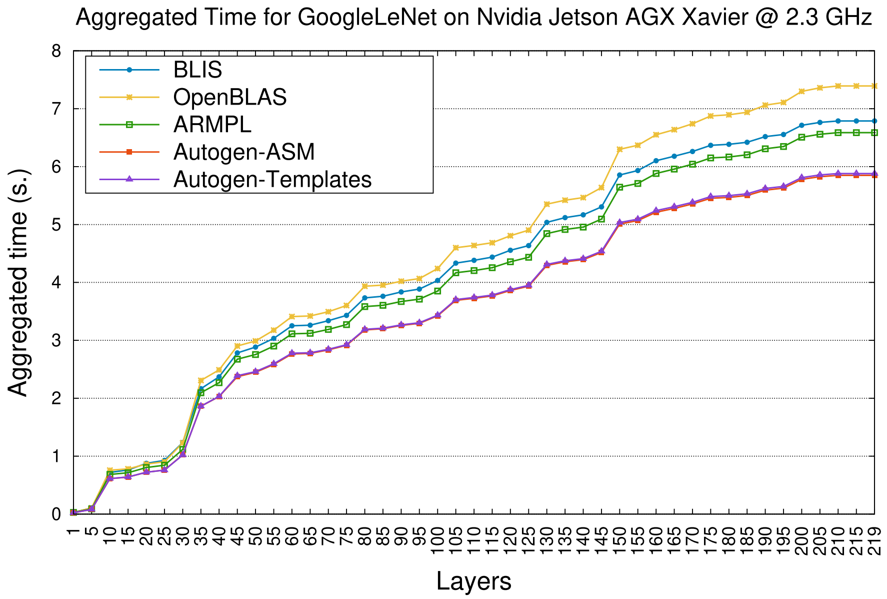
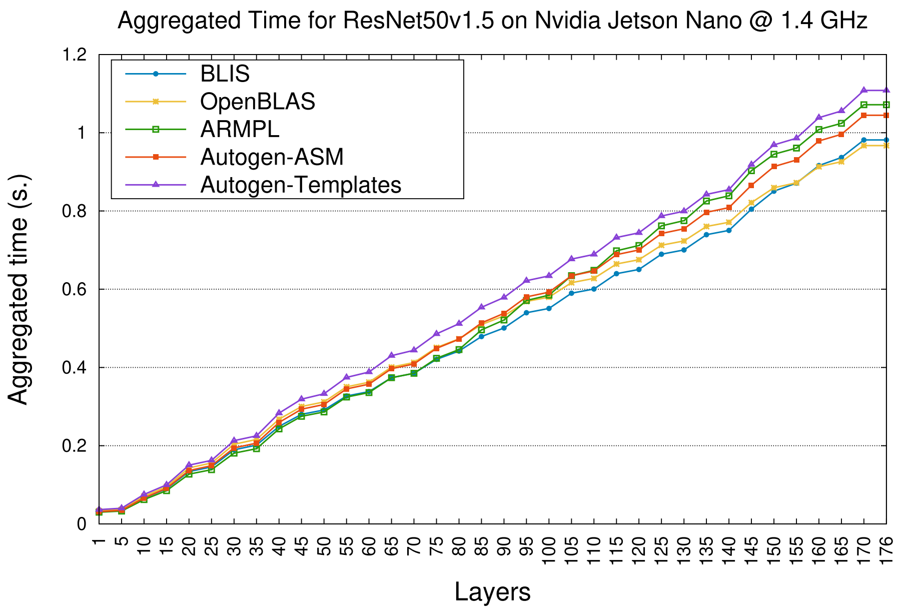


Fig. 6 Aggregated time of the convolutional layers of ResNet-50 v1.5 (top) and GoogleLeNet (bottom) on the NVIDIA Jetson AGX Xavier



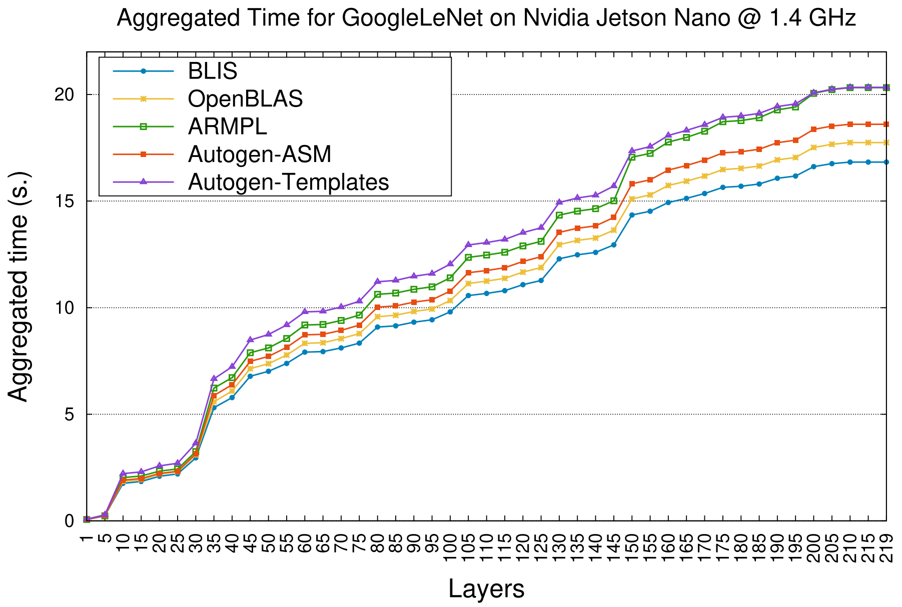


Fig. 7 Aggregated time of the convolutional layers of ResNet-50 v1.5 (top) and GoogleLeNet (bottom) on the NVIDIA Jetson Nano

The possibility of dynamically generating a family of micro-kernels, choosing the most efficient one as a function of the problem dimension, is demonstrated to be key to outperforming the static implementation of GEMM in this libraries, which only include a single micro-kernel per architecture.

A Additional code snippets

```
// vloadC templates to unroll mr/nr loops to load of micro-tile of C
template<int i, int j, int k, int l, typename T>
inline void vloadC_nr(vregister *Cr, T *C) {
  if constexpr (j == 0) return;
  else { vloadC_nr<i, j-1, k, l>(Cr, C);
         vload(*(Cr + (j - 1) * k + i), C + (j - 1) * vl_fp32); }}

template<int i, int j, int k, int l, typename T>
inline void vloadC_mv_nr(vregister *Cr, T *C, int ldC) {
  if constexpr (i == 0) return;
  else { vloadC_mv_nr<i-1, j, k, l>(Cr, C, ldC);
         vloadC_nr<i-1, j, k, l>(Cr, &C[(i - 1) * ldC]); }}

// vload template to unroll mv/nv loops to load of Ar/Br into vregs
template<int i, int j, typename T>
inline void vload_(vregister *ar, T *Ar) {
  if constexpr (i == 0) return;
  else { vload_<i-1, j>(ar, Ar);
         vload(*(ar + i - 1), Ar + (i - 1) * j); }}

// vupdate templates to unroll mv/nv/vl loops to load Ar/Br into vregs
template<int k>
inline void vupdate_vl(vregister *Cr, vregister ar, vregister br) {
  if constexpr (k == 0) return;
  else { vupdate_vl<k-1>(Cr, ar, br);
         vupdate(*(Cr + k - 1), ar, br, k - 1); }}

template<int j, int k>
inline void vupdate_vl_nv(vregister *Cr, vregister ar, vregister *br) {
  if constexpr (j == 0) return;
  else { vupdate_vl_nv<j-1, k>(Cr, ar, br);
         vupdate_vl<k>((Cr + (j - 1) * k), ar, *(br + j - 1)); }}

template<int i, int j, int k, int nr>
inline void vupdate_vl_nv_mv(vregister*Cr, vregister*ar, vregister*br){
  if constexpr (i == 0) return;
  else { vupdate_vl_nv_mv<i-1, j, k, nr>(Cr, ar, br);
         vupdate_vl_nv<j, k>((Cr + (i - 1) * nr), *(ar + i - 1), br);}}

// vstoreC templates to unroll mr/nr loops to store of micro-tile of C
template<int i, int j, int k, int l, typename T>
inline void vstoreC_mv(T *C, vregister *Cr) {
  if constexpr (j == 0) return;
  else { vstoreC_mv<i, j-1, k, l>(C, Cr);
         vstore(C + (j - 1) * vl_fp32, *(Cr + (j - 1) * k + i)); }}

template<int i, int j, int k, int l, typename T>
inline void vstoreC_mv_nr(T *C, vregister *Cr, int ldC) {
  if constexpr (i == 0) return;
  else { vstoreC_mv_nr<i-1, j, k, l>(C, Cr, ldC);
         vstoreC_mv<i-1, j, k, l>(&C[(i - 1) * ldC], Cr); }}
```

Listing 10: Auxiliary generic micro-kernel function templates to operate with any $m_r \times n_r$ micro-tile using vector intrinsics.

```
.LOOP_12x8:                  // Main loop
ldr arq0,   [Ar]             // Load the pr-th
add Ar, Ar, #shift           // column/row of
ldr arq1,   [Ar]             // Ar, Br
add Ar, Ar, #shift
ldr arq2,   [Ar]
add Ar, Ar, #shift
ldr brq0,   [Br]
add Br, Br, #shift
ldr brq1,   [Br]
add Br, Br, #shift

fmla Crv00, arv0, brv0[0] // Update the
fmla Crv01, arv0, brv0[1] // micro-tile
fmla Crv02, arv0, brv0[2] // with AXPYs
fmla Crv03, arv0, brv0[3]

fmla Crv04, arv0, brv1[0]
fmla Crv05, arv0, brv1[1]
fmla Crv06, arv0, brv1[2]
fmla Crv07, arv0, brv1[3]

fmla Crv10, arv1, brv0[0]
fmla Crv11, arv1, brv0[1]
fmla Crv12, arv1, brv0[2]
fmla Crv13, arv1, brv0[3]

fmla Crv14, arv1, brv1[0]
fmla Crv15, arv1, brv1[1]
fmla Crv16, arv1, brv1[2]
fmla Crv17, arv1, brv1[3]

fmla Crv20, arv2, brv0[0]
fmla Crv21, arv2, brv0[1]
fmla Crv22, arv2, brv0[2]
fmla Crv23, arv2, brv0[3]

fmla Crv24, arv2, brv1[0]
fmla Crv25, arv2, brv1[1]
fmla Crv26, arv2, brv1[2]
fmla Crv27, arv2, brv1[3]

sub kc, kc, 1                // Prepare for
cmp kc, 0                    // the next
b.ne .LOOP_12x8              // iteration
```

Listing 11: Main loop of the micro-kernel that operates with a 12×8 micro-tile using ARM NEON assembly. The macros introduced to ease the presentation are omitted for brevity.

Acknowledgements This work received funding from projects PID2020-113656RB and PID2021-12657NB-I00 of MCIN/AEI/ 10.13039/501100011033; PROMETEO 2023-CIPROM/2022/20 and 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, Norway. A. Castelló is a FJC2019-039222-I fellow supported by MCIN/AEI/10.13039/501100011033. M. F. Dolz is supported by the Plan Gen–T grant CIDEXG/2022/013 of the *Generalitat Valenciana*. H. Martínez is a POSTDOC_21_00025 postdoctoral fellow supported by *Junta de Andalucía*.

Author contributions G.A executed the experiments, H.M. implemented the assembly generation, and A.C implemented the generic generator script and wrote different sections of the paper. H.M, P.A. and A.C reviewed G.A work and methodology. M.D. implemented the C++ template approach. F.I. and E.Q conducted the research and wrote several paper sections. All authors reviewed the manuscript.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. European Commission, European Union, 95555. Junta de Andalucía, POSTDOC_21_00025 Agencia Estatal de Investigación, FJC2019-039222, PID2020-113656R, PID2021-12657NB-I00. Generalitat Valenciana, CIDEXG/2022/013, PROMETEO 2023-CIPROM/2022/20

Data availability Not applicable.

Declarations

Conflict of interest The authors declare no competing interests.

Ethical approval Not applicable

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. Dongarra JJ, Du Croz J, Hammarling S, Duff I (1990) A set of level 3 basic linear algebra subprograms. ACM Trans Math Softw 16(1):1–17
2. Kågström B, Ling P, van Loan C (1998) GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. ACM Trans Math Softw 24(3):268–302
3. Goto K, van de Geijn R (2008) High-performance implementation of the level-3 BLAS. ACM Trans Math Soft 35(1):1–14
4. 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
5. 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
6. Goto K, van de Geijn RA (2008) Anatomy of a high-performance matrix multiplication. ACM Trans Math Softw 34(3):12:1-12:25
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. OpenBLAS, http://xianyi.github.com/OpenBLAS/ (2012)
9. Alaejos G, Castelló A, Martínez H, Alonso-Jordá P, Igual FD, Quintana-Ortí ES (2023) Micro-kernels for portable and efficient matrix multiplication in deep learning. J Supercomput 79:8124–8147
10. Martínez H, Catalán S, Igual FD, Herrero JR, Rodríguez-Sánchez R, Quintana-Ortí ES (2023) Co-design of the dense linear algebra software stack for multicore processors, arXiv:2304.14480
11. 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
12. Williams S, Waterman A, Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. Commun ACM 52(4):65–76. https://doi.org/10.1145/1498765.1498785
13. Zee FGV, Smith TM, Marker B, Low TM, Geijn RAVD, Igual FD, Smelyanskiy M, Zhang X, Kistler M, Austel V, Gunnels JA, Killough L (2016) The BLIS framework: Experiments in portability. ACM Trans Math Softw 42(2). https://doi.org/10.1145/2755561

14. Catalán S, Igual FD, Mayo R, Rodríguez-Sánchez R, Quintana-Ortí ES (2016) Architecture-aware configuration and scheduling of matrix multiplication on asymmetric multicore processors. Clust Comput 19(3):1037–1051
15. Dowd K, Severance CR (1998) High performance computing, 2nd edn. O'Reilly
16. 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
17. 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, pp 770–778
18. Szegedy C, et al (2014) Going deeper with convolutions. CoRR [Online]. Available: arXiv:1409.4842
19. Chellapilla K, Puri S, Simard P (2006) High performance convolutional neural networks for document processing. In: International Workshop on Frontiers in Handwriting Recognition
20. ArmPL: Arm Performance Libraries, https://developer.arm.com/downloads/-/arm-performance-libraries. Accessed July 2023

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

Authors and Affiliations

Guillermo Alaejos[1] · Héctor Martínez[2] · Adrián Castelló[1] · Manuel F. Dolz[3] · Francisco D. Igual[4] · Pedro Alonso-Jordá[1] · Enrique S. Quintana-Ortí[1]

✉ Adrián Castelló
adcastel@disca.upv.es

Guillermo Alaejos
galalop@upv.es

Héctor Martínez
el2mapeh@uco.es

Manuel F. Dolz
dolzm@uji.es

Francisco D. Igual
figual@ucm.es

Pedro Alonso-Jordá
palonso@upv.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