Tackling the Matrix Multiplication Micro-kernel Generation with EXO

Adrián Castelló*
Universitat Politècnica de València, Spain
adcastel@disca.upv.es

Julian Bellavita§
Cornell University, USA
jb2695@cornell.edu

Grace Dinh§
UC Berkeley, USA
dinh@berkeley.edu

Yuka Ikarashi§
MIT CSAIL, USA
yuka@csail.mit.edu

Héctor Martínez§
Universidad de Córdoba, Spain
el2mapeh@uco.es

2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) | 979-8-3503-9509-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/CGO57630.2024.10444883

***Abstract*—The optimization of the matrix multiplication (or GEMM) has been a need during the last decades. This operation is considered the flagship of current linear algebra libraries such as BLIS, OpenBLAS, or Intel OneAPI because of its widespread use in a large variety of scientific applications. The GEMM is usually implemented following the GotoBLAS philosophy, which tiles the GEMM operands and uses a series of nested loops for performance improvement. These approaches extract the maximum computational power of the architectures through small pieces of hardware-oriented, high-performance code called micro-kernel. However, this approach forces developers to generate, with a non-negligible effort, a dedicated micro-kernel for each new hardware.**

In this work, we present a step-by-step procedure for generating micro-kernels with the EXO compiler that perform close to (or even better than) manually developed microkernels written with intrinsic functions or assembly language. Our solution also improves the portability of the generated code, since a hardware target is fully specified by a concise library-based description of its instructions.

***Index Terms*—code generation, high performance, Exo, linear algebra, micro-kernels**

I. INTRODUCTION

Over the past few decades, there has been a relentless effort to develop high-performance implementations of linear algebra (LA) libraries. These solutions have been designed to target a wide variety of architectures, such as vector processors, multi-core processors, data-parallel graphics processing units (GPUs), and, more recently, domain-specific architectures and accelerators. This not-insignificant effort usually comes from major hardware vendors, with some relevant products being Intel OneAPI, AMD AOCL, IBM ESSL, ARMPL, and NVIDIA cuBLAS, as well as from the academic side, with software packages such as GotoBLAS2 [1], OpenBLAS [2], BLIS [3] and ATLAS [4].

The general matrix multiplication (GEMM) is the flagship computational kernel on which these LA libraries are built upon. The overall performance of GEMM comes from the macro-kernel which comprises a series of general code optimizations such as tiling, loop reordering, and data packing, and the micro-kernel which is responsible for a significant part of the performance because is the link between the GEMM algorithm and the underlying hardware. In addition, GEMM is also a key operation for deep learning (DL) applications that use convolutional deep neural networks (DNNs) for signal processing and computer vision [5], [6]. Unfortunately, these LA libraries have some limitations:

1) The optimized routines are hardware-specific. This is the case for Intel, IBM, or ARM products. To a lesser, it also applies to GotoBLAS2, OpenBLAS, and BLIS, which use a collection of hardware-oriented micro-kernels [7].
2) Developing highly optimized, hand-written, micro-kernels for GEMM requires deep knowledge of high-performance computing and computer architecture.
3) A new architecture implies a new set of test-and-debug development of micro-kernels to extract the maximum computational power from the hardware.
4) The code of these micro-kernels usually uses productivity-enhancing macros, templates, and high-level programming techniques. Therefore, maintaining the libraries thus mostly lies in the hands of the original developers.
5) The software misses some relevant cases such as for example, support for half (i.e., 16-bit) floating point precision or integer arithmetic.
6) The implementation of a unique micro-kernel for all GEMM scenarios may incur in a performance loss for non-squarish GEMM as they appear in DL.

In this paper, we address these limitations by demonstrating that it is possible to automatically generate a set of micro-kernels for GEMM using EXO [8]. This alternative solution has the following advantages:

1) At a high level, the library micro-kernel is "replaced" by a collection of EXO generated C code. Each micro-kernel will handle a different edge case.
2) Using the appropriate backend, the generation/optimization can be easily specialized for different data types, which further enhances the portability and maintainability of the solution.

*Corresponding author.
§These authors contributed equally to this work.

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on April 02,2024 at 18:47:12 UTC from IEEE Xplore. Restrictions apply.

3) By matching the size of the micro-kernel to the problem, it is possible to outperform the high-performance realizations of GEMM of existing, widely-accepted solutions.
4) The optimization process for each problem is greatly reduced, boiling down to evaluating a number of generated micro-kernels.
5) The micro-kernel generator for ARM Neon is publicly available at https://github.com/adcastel/EXO_ukr_generator.

Moreover, this work highlights the usability of EXO by showing that the performance achieved by auto-generated C code is as good as hand-written solutions by generating different codes for matching the problem sizes.

In addition, the work in this paper has contributed to the EXO code with the support for two Neon intrinsic instructions and the support for 16-bit floating point data types for ARM[1].

The rest of the paper is organized as follows. Section II resumes the BLIS algorithm for GEMM and introduces the EXO domain-specific programming language; Section II-C visits some existing work; Section III presents a step-by-step process of how to generate optimized micro-kernel ARM codes using EXO; Section IV evaluates and compares the generated micro-kernels with other micro-kernels in three different scenarios; and Section V summarizes the work done and its contributions.

II. BACKGROUND

A. *BLIS Algorithm for* GEMM

Consider the GEMM $C = C + AB$, where the operands are matrices with the following dimensions: $m \times k$, $k \times n$, and $m \times n$ for A, B, and C, respectively. The BLIS framework (as well as other LA libraries) follows GotoBLAS [1] approach to encoding this operation as three nested loops around two *packing routines*. This structure is known as macro-kernel and its code is usually shared across different hardware architectures. In BLIS, the macro-kernel is decomposed into two additional loops around a *micro-kernel*, with the latter consisting of a single loop that performs one outer product per iteration. The Figures 1 and 2 show the *BLIS baseline algorithm* for GEMM, which includes the six loops, the two packing routines, and the micro-kernel.

The three outermost loops of the algorithm travel the n–, k–, and m–dimensions of the problem, partitioning the matrix operands to fit the processor's cache hierarchy. The cache configuration parameters n_c, k_c, and m_c, adapted to the target processor architecture [9], favor that B_c stays in the L3 cache and A_c in the L2 cache during the execution of the micro-kernel, while C is streamed from the main memory into the processor registers as shown in Figure 2

In addition, the macro-kerenl packing routines ensure that the data in A_c, B_c is accessed with unit stride from the micro-kernel.

[1]FP16 support is not in the EXO repository at the time of writing this paper but is available at https://github.com/adcastel/exo

```
1  for (jc=0; jc<n; jc+=nc)  // Loop L1
2   for (pc=0; pc<k; pc+=kc) {    // L2
3    // Pack B
4    Bc := B(pc:pc+kc-1,jc:jc+nc-1);
5    for (ic=0; ic<m; ic+=mc) {   // L3
6     // Pack A
7     Ac := A(ic:ic+mc-1,pc:pc+kc-1);
8     for (jr=0; jr<nc; jr+=nr)   // L4
9       for (ir=0; ir<mc; ir+=mr) // L5
10       // Micro-kernel          // L6
11       C(ic+ir:ic+ir+mr-1,
12         jc+jr:jc+jr+nr-1)
13         += Ac(ir:ir+mr-1,0:kc-1)
14         *  Bc(0:kc-1,jr:jr+nr-1);
15   }}
```

Fig. 1: Pseudo-code of the BLIS base GEMM algorithm.

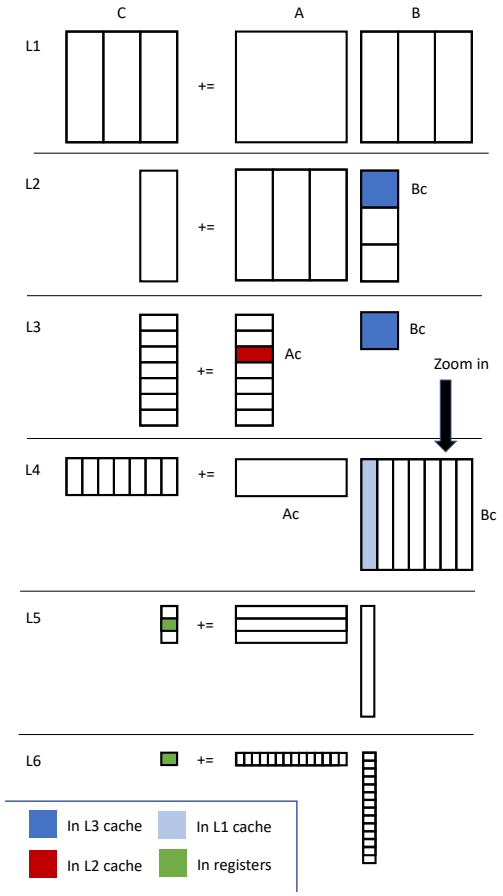

Fig. 2: Overview of the BLIS base GEMM algorithm.

The micro-kernels are pieces of hardware-oriented code that are usually encoded using assembly language. Inside the micro-kernel, a block of $m_r \times n_r$ elements of C is updated inside the k_c loop. The operands for the update are portions of A_c and B_c buffers, more specifically A_r and B_r, whose dimensions are $m_r \times k_c$ and $k_c \times n_r$, respectively. The values of m_r and n_r are commonly used to name the micro-kernel. BLIS provides specialized micro-kernels for many processors from AMD, Intel, ARM, and IBM. However, developing a new hardware-oriented micro-kernel for each new architecture is an expensive development effort, so BLIS provides one micro-

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on April 02,2024 at 18:47:12 UTC from IEEE Xplore. Restrictions apply.

kernel per architecture using a monolithic approach.

B. Compilation and Tuning Frameworks

The expense of developing optimized micro-kernels for diverse architectures largely stems from the need to perform architecture-specific optimizations to obtain maximum performance, and to generate platform-specific instructions such as hardware intrinsics. To address this, *automatic code generation* approaches have been applied to achieve performance portability across existing and new architectures, including general-purpose processors or specific-purpose accelerators [10], [11], with a minimum of programmer intervention [12].

Specifically, *user-schedulable* languages and compiler frameworks, such as Halide [13], MLIR [14], TVM [15] or EXO [8], propose a clear separation of concerns between the definition of an operation (e.g. a matrix multiplication) and the *schedule*, or set of optimizations applied to the operation, allowing common optimizations (e.g loop tiling for higher levels of the memory hierarchy) to be abstracted away from architecture-specific optimizations such as vectorization.

Compiler frameworks must also be given a *hardware specification* describing the hardware's instruction set (including vector intrinsics). Traditional compilers such as Halide, TVM, and LLVM integrate hardware specifications tightly into the compiler, requiring manual specification of code generation passes to support new hardware. Therefore, sometimes, the generation tools are bound to certain compilers so the user cannot deal with several code decisions. As an example, TVM and LLVM are bound so the user does not have to compile the code but, at the same time, there are some aspects that the user is not aware of.

EXO [8], on the other hand, externalizes the definition of hardware intrinsics by taking as input user-defined libraries. For instance, Fig. 3 provides definitions for the ARM Neon intrinsic functions `vstlq_f32` and `vfmaq_laneq_f32` in semantically equivalent Python; these definitions are used by the compiler to automatically generate calls. An entire hardware specification consists of similar definitions for each hardware intrinsic, as well as definitions for levels of the memory hierarchy (in this case, DRAM and Neon registers) and datatypes (in this case, `f32` floats). Concretely, these definitions will ensure that the user methods do not change the behavior of the original code by checking the intrinsic replacement with the expected pattern. Without this security definition, the user could change any loop by any intrinsic instruction which may evoke a different code. Another key aspect from EXO is its independence from any compiler/optimizer tool. Specifically, EXO "just" generate a C code with intrinsic instructions that need to be compiled and, therefore, the user can try different combinations of hardware/compiler to obtain the maximum performance of the generated code.

A serious drawback of some of these frameworks is the reliance on existing libraries. For example, the code generated by TVM uses a set of TVM objects that may force rewriting a large part (or all) of the software stack, in addition to incurring runtime overheads to convert datatypes to those supported by these libraries. However, tools such as EXO generate plain C code that can be used within all the existing performance-oriented libraries.

```
@instr("vst1q_f32(&{dst_data}, {src_data});")
def neon_vst_4xf32(dst: [f32][4] @ DRAM,
                   src: [f32][4] @ Neon):
  assert stride(src, 0) == 1
  assert stride(dst, 0) == 1

  for i in seq(0, 4):
    dst[i] = src[i]

@instr("{dst_data} = vfmaq_laneq_f32(
        {dst_data}, {lhs_data}, {rhs_data}, {l});")
def neon_vfmla_4xf32_4xf32(dst: [f32][4] @ Neon,
                           lhs: [f32][4] @ Neon,
                           rhs: [f32][4] @ Neon,
                           l: index):
  assert stride(dst, 0) == 1
  assert stride(lhs, 0) == 1
  assert stride(rhs, 0) == 1
  assert l >= 0
  assert l < 4

  for i in seq(0, 4):
    dst[i] += lhs[i] * rhs[l]
```

Fig. 3: Hardware specification (specified as a library) for vector intrnsic instructions in EXO

C. Autotuning and Optimization

By treating schedules as inputs user-schedulable languages facilitate the automatic exploration of scheduling spaces through auto-tuning techniques [16], [17]. For example, AutoTVM [18], as part of TVM, performs a full exploration of a search space defined by a user-specified template. While such search methods have shown strong performance in practice, they must contend with a computationally expensive search space of possible tuning parameters that grows exponentially with the dimension of the design space as well as difficulties in generalizability (both to different problem sizes and to different hardware targets) and explainability to developers.

Recently, work in [9] as part of the BLIS project, has shown that the use of analytical models for optimal configuration parameters selection is an effective way to achieve high performance without the need for auto-tuning. Replacing the auto-tuning scheme with model-based solutions has also been successfully explored in [2], [19], [20], [21].

This work, focused on micro-kernel generation, differs from [22], which uses MLIR to describe early experiences with the entire GEMM algorithm, from [23], which proposes advanced auto-tuning schemes for the primitive, and from [24] which uses a Python script and C macros to build micro-kernels. Specifically, we use EXO to extend and further analyze the manual generation of GEMM micro-kernels and integrate the resulting code into a BLIS-like GEMM algorithm.

III. CODE GENERATION

In this section, we explain in a step-by-step manner how to build an HPC micro-kernel code with EXO for ARMv8 architecture from scratch. The dimensions of the micro-kernel will be 8×12, as the one present in the BLIS library for

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on April 02,2024 at 18:47:12 UTC from IEEE Xplore. Restrictions apply.

this specific architecture. For each building phase, we first introduce the employed EXO's instructions used and explain the resulting intermediate code. We also show how to extend the code generation to handle other features of the micro-kernels. The code for the step-by-step generation is available at https://github.com/adcastel/EXO_ukr_generator.

A. Micro-kernel Generation

Figure 4 shows what the initial code looks like for a column-major micro-kernel based on the outer product (as in BLIS). To meet the BLIS GEMM algorithm features, we have modified the initial micro-kernel code as follows: 1) Since the C language is a row-major allocator, we transpose the C matrix dimensions; 2) BLIS uses data packings for the A and B operands of the micro-kernel, which guarantees unit stride access to the data. Therefore, to have this in the A_c operand, we also swap the A_c dimensions. The B_c operand is already accessed with a unit stride so no changes are needed; 3) We rearrange the internal loops in a k, j, i order to match the desired structure. Note that A_c and B_c operands are allocated in a 1-dimensional structure in the algorithm that calls the micro-kernel, so although we transpose the data buffers C and A_c there is no risk of the access pattern.

To start with the explanation of the code in Figure 4, the first line (`@proc`) tells the EXO compiler that the next function is schedulable, and will therefore produce a C-language code. The arguments of the function also show some aspects of the EXO language. First, the argument name is followed by ":" and the type, which can be `size`, `scalar`, `vector`, or `matrix`. For data that requires memory allocation, we should specify its placement. In this example, we map the A_c, B_c, and C buffers to RAM using the `@ DRAM` notation. This version covers all combinations of alpha and beta values. The buffers C_b and B_a located at lines 8 and 9 are used for the computation of $C * beta$ and $B_c * alpha$. Specifically, lines 12–14 compute the C_b results and lines 17–19 compute the B_a results.

Lines 22–25 perform the micro-kernel result by executing $C_b = C_b + A_c \times B_a$. Finally, lines 28–30 will return the result of the computation to the C matrix.

For simplicity, from this point on, we will optimize a specific version of the micro-kernel. Specifically, we will apply the step-by-step transformation to the code shown in Figure 5 that corresponds to the micro-kernel when $alpha$ and $beta$ values both equal 1.

Optimization of the initial code will involve more scheduling functions for the C_b and B_a loops (lines 11–14, 17–19, and 27–30), equivalent to those shown from this point beyond.

1) Basic Micro-kernel: First, we generate the function to schedule and we specialize the generated code by specifying that we want to use the values of 8 and 12 for the M_R and N_R arguments, respectively. This conversion is done via the `partial_eval` function that replace the values M_R and N_R. Figure 6 shows the user code and the generated representation. Lines 3 and 4 get the initial version of the micro-kernel and set the variables M_R and N_R variables, respectively. The generated code has changed the variables

```
1  @proc
2  def ukernel_ref( MR: size, NR: size, KC: size,
3    alpha: f32[1], Ac: f32[KC, MR] @ DRAM,
4    Bc: f32[KC, NR] @ DRAM, beta: f32[1],
5    C: f32[NR, MR] @ DRAM,
6    ):
7    # Tmp buffers for C * beta and B * alpha
8    Cb: f32[NR,MR] @ DRAM
9    Ba: f32[KC,NR] @ DRAM
10
11   # Cb = C * beta
12   for cj in seq(0, NR):
13     for ci in seq(0, MR):
14       Cb[cj,ci] = C[cj,ci] * beta[0]
15
16   # Ba = Bc * alpha
17   for bk in seq(0, KC):
18     for bj in seq(0, NR):
19       Ba[bk,bj] = Bc[bk,bj] * alpha[0]
20
21   # C += Ac * Bc
22   for k in seq(0, KC):
23     for j in seq(0, NR):
24       for i in seq(0, MR):
25         Cb[j, i] += Ac[k,i] * Ba[k,j]
26
27   # C = Cb
28   for cj in seq(0, NR):
29     for ci in seq(0, MR):
30       C[cj,ci] = Cb[cj,ci]
```

Fig. 4: EXO code for GEMM micro-kernel.

```
1  @proc
2  def ukernel_ref( MR: size, NR: size, KC: size,
3    alpha: f32[1], Ac: f32[KC, MR] @ DRAM,
4    Bc: f32[KC, NR] @ DRAM, beta: f32[1],
5    C: f32[NR, MR] @ DRAM,
6    ):
7
8    # C += Ac * Bc
9    for k in seq(0, KC):
10     for j in seq(0, NR):
11       for i in seq(0, MR):
12         C[j, i] += Ac[k,i] * Bc[k,j]
```

Fig. 5: Simplified EXO code for GEMM micro-kernel.

by their values (e.g. the iterations of the second loop now go from 0 to 12 instead of N_R).

```
1  # USER CODE
2  MR, NR = 8, 12
3  p = rename(ukernel_ref, "uk{}x{}".format(MR,NR))
4  p = p.partial_eval(MR,NR)
5
6  # RESULTING EXO GENERATED CODE
7  def uk_8x12( KC: size, alpha: f32[1] @ DRAM,
8    Ac: f32[KC, 8] @ DRAM, Bc: f32[KC, 12] @ DRAM,
9    beta: f32[1] @ DRAM, C: f32[12, 8] @ DRAM
10   ):
11
12   # C += Ac * Bc
13   for k in seq(0, KC):
14     for j in seq(0, 12):
15       for i in seq(0, 8):
16         C[j, i] += Ac[k, i] * Bc[k, j]
```

Fig. 6: EXO code for GEMM micro-kernel v1.

2) Loop Structure: In lines 2 and 3 of Figure 7 we split both the `i` and `j` loops to match the vector length of the architecture. As the ARM-based NVIDIA Carmel processor uses 128-bit vector registers and, with float32 data type, the

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on April 02,2024 at 18:47:12 UTC from IEEE Xplore. Restrictions apply.

vector length is 4. In addition, the BLIS-like GEMM algorithm used in this work ensures that the A_c and B_c buffer sizes are multiples of the M_R and N_R values, respectively. This action results in the nested loops `it, itt, jt` and `jtt` locate in lines 14–17. In addition, EXO has automatically tiled the access to the C, A_c, and B_c data to match the new loop structure (lines 18–20).

```
1  # USER CODE
2  p = divide_loop(p,'i', 4, ['it','itt'], perfect=True)
3  p = divide_loop(p,'j', 4, ['jt','jtt'], perfect=True)
4
5  # RESULTING EXO GENERATED CODE
6  def uk_8x12( KC: size, alpha: f32[1] @ DRAM,
7    Ac: f32[KC, 8] @ DRAM, Bc: f32[KC, 12] @ DRAM,
8    beta: f32[1] @ DRAM, C: f32[12, 8] @ DRAM
9    ):
10
11   # C += Ac * Bc
12   for k in seq(0, KC):
13     # Loop splits to match the vector length (4)
14     for jt in seq(0, 3):
15       for jtt in seq(0, 4):
16         for it in seq(0, 2):
17           for itt in seq(0, 4):
18             C[jtt + 4 * jt, itt + 4 * it] +=
19               Ac[k, itt + 4 * it] *
20               Bc[k, jtt + 4 * jt]
```

Fig. 7: EXO code for GEMM micro-kernel v2.

3) C Matrix: Figure 8 shows one of the most complex steps in the micro-kernel generation, the binding of the C matrix to vectorial registers, which includes: declaration, loading, and storing the results. Specifically, the process is as follows:

1) Map the C matrix with a vector register (lines 3 and 4), which is reflected in line 51. The `stage_mem` function binds the C operand memory to a vectorial register so EXO can then set the C operand data movement.
2) Resize the vectorial register to a 3D structure where each dimension corresponds to the iterations of each loop. This register resize is done by using the `expand_dim` method. Specifically, the first invocation (line 7) resizes the declaration to the size of the vector length, which in this case is 4; line 8 completes the size of the M_R dimension; and line 9 is bound to the N_R dimension of C. The result of these lines can be seen in line 34, where the final allocation appears.
3) The `lift_alloc` statement moves the declaration of the registers of C to the top of the generated code.
4) Lines 15–18 move the load and the store of the C matrix out of the computation loop (lines 37–43 and lines 56–62, respectively).
5) Lines 21 and 22 replace the `itt` loops of the load and store with intrinsic instructions. Lines 40 and 59 emphasize these replacements.
6) Line 25 sets the C register variable to type `Neon`.

4) A_c and B_c Operands: Figure 9 lists the actions to generate the loads from A_c and B_c to registers. Note that the code uses the name X_c for simplicity since these actions must be performed for both operands. The procedure for each operand is as follows:

```
1  # USER CODE
2  #  1) Map C buffer to vectorial register C_reg
3  Cp = 'C[4 * jt + jtt, 4 * it + itt]'
4  p = stage_mem(p, 'C[_] += _', Cp, 'C_reg')
5
6  #  2) Build a 3D structure of C_reg
7  p = expand_dim(p, 'C_reg', 4, 'itt', ...)
8  p = expand_dim(p, 'C_reg', MR//4, 'it', ...)
9  p = expand_dim(p, 'C_reg', NR, 'jt*4+jtt', ...)
10
11 #  3) Move the register declaration to the top
12 p = lift_alloc(p, 'C_reg', n_lifts=5)
13
14 #  4) Extract the C load and store from the k-loop
15 p = autofission(p, p.find('C_reg[_] = _').after(),
16                  n_lifts=5)
17 p = autofission(p, p.find('C[_] = _').before(),
18                  n_lifts=5)
19
20 #  5) Replace the indicated loops by Neon intrinsics
21 p = replace(p, 'for itt in _: _', neon_vld_4xf32)
22 p = replace(p, 'for itt in _: _', neon_vst_4xf32)
23
24 #  6) Set the C_reg memory to Neon
25 p = set_memory(p, 'C_reg', Neon)
26
27 # RESULTING EXO GENERATED CODE
28 def uk_8x12( KC: size, alpha: f32[1] @ DRAM,
29   Ac: f32[KC, 8] @ DRAM, Bc: f32[KC, 12] @ DRAM,
30   beta: f32[1] @ DRAM, C: f32[12, 8] @ DRAM
31   ):
32
33   # Registers for C
34   C_reg: f32[12, 2, 4] @ Neon
35
36   # Load C to registers
37   for jt in seq(0, 3):
38     for jtt in seq(0, 4):
39       for it in seq(0, 2):
40         neon_vld_4xf32(
41             C_reg[4 * jt + jtt, it, 0:4],
42             C[4 * jt + jtt, 4 * it:4 * it + 4]
43         )
44
45   # C += Ac * Bc
46   for k in seq(0, KC):
47     for jt in seq(0, 3):
48       for jtt in seq(0, 4):
49         for it in seq(0, 2):
50           for itt in seq(0, 4):
51             C_reg[jt * 4 + jtt, it, itt] +=
52               Ac[k, itt + 4 * it] *
53               Bc[k, jtt + 4 * jt]
54
55   # Store C from registers
56   for jt in seq(0, 3):
57     for jtt in seq(0, 4):
58       for it in seq(0, 2):
59         neon_vst_4xf32(
60             C[jtt + 4 * jt, 4 * it:4 * it + 4],
61             C_reg[jtt + 4 * jt, it, 0:4]
62         )
```

Fig. 8: EXO code for GEMM micro-kernel v3.

1) Map the X_c matrix to a vector register (line 3).
2) Resize the vector register to a 2D structure where the first dimension is the vector length and the second dimension is the outermost loop (lines 6 and 7). The result of these lines can be seen in lines 32 and 33 where the final allocation appears.
3) The `lift_alloc` statement moves the declaration of the registers outside the k-loop.
4) Lines 13 and 14 move the loading of the operands to the k-loop.

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on April 02,2024 at 18:47:12 UTC from IEEE Xplore. Restrictions apply.

5) Line 17 replaces the `xtt` loops with neon vector load instructions.
6) Line 20 sets the register variable to `Neon` type.

```
1  # USER CODE
2  #  1) Map Xc buffer to vectorial register X_reg
3  p = bind_expr(p, 'Xc[_]','X_reg')
4
5  #  2) Build a 2D structure of X_reg
6  p = expand_dim(p, 'X_reg' , 4, 'xtt', ...)
7  p = expand_dim(p, 'X_reg', XR//4, 'xt', ...)
8
9  #  3) Move the register declaration to the top
10 p = lift_alloc(p, 'X_reg', n_lifts=5)
11
12 #  4) Move the Xc load to the k-loop
13 p = autofission(p, p.find('X_reg[_] = _').after(),
14                 n_lifts=4)
15
16 #  5) Replace the xtt loop by Neon intrinsics
17 p = replace(p, 'for xtt in _: _', neon_vld_4xf32)
18
19 #  6) Set the X_reg memory to Neon
20 p = set_memory(p, 'X_reg', Neon)
21
22 # RESULTING EXO GENERATED CODE
23 def uk_8x12( KC: size, alpha: f32[1] @ DRAM,
24   Ac: f32[KC, 8] @ DRAM, Bc: f32[KC, 12] @ DRAM,
25   beta: f32[1] @ DRAM, C: f32[12, 8] @ DRAM
26   ):
27
28   # Registers for C and load C as in the
29   # previous figure.  Omitted for brevity
30
31   # Registers for Ac and Bc
32   A_reg: R[2, 4] @ Neon
33   B_reg: R[3, 4] @ Neon
34   for k in seq(0, KC):
35     # Load Ac and Bc to registers
36     for it in seq(0, 2):
37       neon_vld_4xf32(
38         A_reg[it, 0:4],
39         Ac[k, 4 * it:4 + 4 * it])
40     for jt in seq(0, 3):
41       neon_vld_4xf32(
42         B_reg[jt, 0:4],
43         Bc[k, 4 * jt:4 + 4 * jt])
44     for jt in seq(0, 3):
45       for jtt in seq(0, 4):
46         for it in seq(0, 2):
47           for itt in seq(0, 4):
48             C_reg[jtt + 4 * jt, it, itt] +=
49               A_reg[it, itt] * B_reg[jt, jtt]
50
51   # Store C from registers as in the
52   # previous figure.  Omitted for brevity
```

Fig. 9: EXO code for GEMM micro-kernel v4.

5) GEMM Operation: Figure 10 illustrates this step. We reorder the `jtt` and `it` loops of the calculation (line 2) so that the access to the B register values is sequential. Line 3 replaces the innermost loop for the `fmla` statement as shown in lines 24–26.

6) Loop Unrolling: Although this is a technique that some compilers use by default, it is also possible to do it manually in EXO. Figure 11 shows an example of unrolling for the loops that load A_c and B_c operands into registers. We show it with the `unroll_loop` statements of lines 2 and 3, resulting in lines 21–25.

7) Generated C-code: To ensure that the generated code is not only optimized in the C language but also that the compilation to assembly is done correctly, we have compiled

```
1  # USER CODE
2  p = reorder_loops(p,'jtt it')
3  p = replace(p, 'for itt in _: _',
4                 neon_vfmla_4xf32_4xf32)
5
6  # RESULTING EXO GENERATED CODE
7  def uk_8x12( KC: size, alpha: f32[1] @ DRAM,
8    Ac: f32[KC, 8] @ DRAM, Bc: f32[KC, 12] @ DRAM,
9    beta: f32[1] @ DRAM, C: f32[12, 8] @ DRAM
10   ):
11
12   # Registers for C, Ac and Bc and loads as in
13   # previous figures.  Omitted for brevity
14
15   # C += Ac * Bc
16   for k in seq(0, KC):
17     # Load Ac and Bc to registers as in
18     # previous figures. Omitted for brevity
19
20     # Computation with registers
21     for jt in seq(0, 3):
22       for it in seq(0, 2):
23         for jtt in seq(0, 4):
24           neon_vfmla_4xf32_4xf32(
25             C_reg[jtt + 4 * jt, it, 0:4],
26             A_reg[it, 0:4], B_reg[0:4, jt], jtt)
27
28   # Store C from registers as in
29   # previous figures.  Omitted for brevity
```

Fig. 10: EXO code for GEMM micro-kernel v5.

```
1  # USER CODE
2  p = unroll_loop(p,'it')
3  p = unroll_loop(p,'jt')
4
5  # RESULTING EXO GENERATED CODE
6  def uk_8x12( KC: size, alpha: f32[1] @ DRAM,
7    Ac: f32[KC, 8] @ DRAM, Bc: f32[KC, 12] @ DRAM,
8    beta: f32[1] @ DRAM, C: f32[12, 8] @ DRAM
9    ):
10
11   # Registers for C and load C as in
12   # previous figures.  Omitted for brevity
13
14   # Registers for Ac and Bc
15   A_reg: R[2, 4] @ Neon
16   B_reg: R[3, 4] @ Neon
17
18   # C += Ac * Bc
19   for k in seq(0, KC):
20     # Unrolled loads from Ac and Bc to registers
21     neon_vld_4xf32(A_reg[0, 0:4], Ac[k, 0:4 + 0])
22     neon_vld_4xf32(A_reg[1, 0:4], Ac[k, 4:4 + 4])
23     neon_vld_4xf32(B_reg[0, 0:4], Bc[k, 0:4 + 0])
24     neon_vld_4xf32(B_reg[1, 0:4], Bc[k, 4:4 + 4])
25     neon_vld_4xf32(B_reg[2, 0:4], Bc[k, 8:4 + 8])
26     # Computation with registers
27     for jt in seq(0, 3):
28       for it in seq(0, 2):
29         for jtt in seq(0, 4):
30           neon_vfmla_4xf32_4xf32(
31             C_reg[jtt + 4 * jt, it, 0:4],
32             A_reg[it, 0:4], B_reg[0:4, jt], jtt)
33
34   # Store C from registers as in
35   # previous figures.  Omitted for brevity
```

Fig. 11: EXO code for GEMM micro-kernel v6.

the c-code with the `gcc-10 -S` command, and the resulting assembly code for the k-loop is shown in Figure 12. This output is as optimized as the one implemented by hand in BLIS.

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on April 02,2024 at 18:47:12 UTC from IEEE Xplore. Restrictions apply.

```
.L3:
  ldp     q1, q0, [x3]           #load A -> q0, q1
  add     x0, x0, 1
  ldp     q4, q3, [x4]           #load B -> q3, q4
  add     x3, x3, 32
  ldr     q2, [x4, 32]           #load B -> q2
  add     x4, x4, 48
  fmla    v12.4s, v1.4s, v4.s[0] # C = C + q1 * q4[0]
  fmla    v10.4s, v1.4s, v4.s[1] # C = C + q1 * q4[1]
  fmla    v8.4s, v1.4s, v4.s[2]  # C = C + q1 * q4[2]
  fmla    v30.4s, v1.4s, v4.s[3] # ...
  fmla    v11.4s, v0.4s, v4.s[0]
  fmla    v9.4s, v0.4s, v4.s[1]
  fmla    v31.4s, v0.4s, v4.s[2]
  fmla    v29.4s, v0.4s, v4.s[3]
  fmla    v28.4s, v1.4s, v3.s[0]
  fmla    v26.4s, v1.4s, v3.s[1]
  fmla    v24.4s, v1.4s, v3.s[2]
  fmla    v22.4s, v1.4s, v3.s[3]
  fmla    v27.4s, v0.4s, v3.s[0]
  fmla    v25.4s, v0.4s, v3.s[1]
  fmla    v23.4s, v0.4s, v3.s[2]
  fmla    v21.4s, v0.4s, v3.s[3]
  fmla    v20.4s, v1.4s, v2.s[0]
  fmla    v18.4s, v1.4s, v2.s[1]
  fmla    v16.4s, v1.4s, v2.s[2]
  fmla    v6.4s, v1.4s, v2.s[3]
  fmla    v19.4s, v0.4s, v2.s[0]
  fmla    v17.4s, v0.4s, v2.s[1]
  fmla    v7.4s, v0.4s, v2.s[2]
  fmla    v5.4s, v0.4s, v2.s[3]
  cmp     x1, x0
  bne     .L3
```

Fig. 12: Assembly generated with the gcc-10 compiler of the EXO code for GEMM micro-kernel.

B. Edge Cases

One of the problems with the one micro-kernel per architecture approach adopted by the HPC libraries is the performance degradation when the dimensions of the micro-kernel do not match the optimized ones. This situation is called an edge case. Solutions such as BLIS use a non-specialized version of the micro-kernel for these situations because the edge cases have no performance impact for large problem sizes. However, newer HPC scenarios such as DL are full of these edge cases. A clear example is the sizes of the first layer of the ResNet50-v1.5 convolutional model, where after applying the IM2ROW method, the resulting GEMM dimensions are $12544, 64,$ and 147 for $M, N,$ and K, respectively.

Using EXO, and assuming that the BLIS packings are in the GEMM algorithm, all we need to do is use the code shown in Figure 6 and change the values for M_R and N_R to match the edge cases. Then, running all the steps will generate a new micro-kernel that matches those values.

However, it is possible that we do not need the packing because the data is already packed or the size of the problem is small enough that the cost of packing is not worth it. In this scenario, we should adjust the procedure of the micro-kernel generation (e.g. non-packing of A) as follows:

1) Loop `i` in Figure 7 should not be split.
2) The mapping between A_c and `A_reg` will change and the dimensions of the latter will match the value of M_R.
3) Inside the `k`-loop, `A_reg` will be broadcasted with the values A_c.
4) The calculation will use the `neon_vfmadd_4xf32_4xf32`, which computes the broadcasted `A_reg` by the entire `B_reg`.

C. Architectural Portability

EXO only needs to change the third argument in the `replace` statements in the user code to generate the desired code. If the new architecture provides an API with the same functionality, this is the required change.

However, it is possible that a statement used in this example is not present in the intrinsic Application Programming Interface (API) of other hardware (e.g., ARM Neon `vfmaq_laneq_f32`). Then, a similar approach to the one presented when the non-packing of data is available is used.

As a simple example, changing the line 21 in Figure 8 from `replace(p, 'for itt in _: _', neon_vld_4xf32)` by `replace(p, 'for itt in _: _', _mm512_loadu_ps)` will change the load intrinsic from ARM Neon to Intel AVX512.

D. Data Types

Generating micro-kernels for different data types is as easy as using the function `set_precision` function for each memory allocation and register in the code. For example, `set_precision(p, A_reg,"f16")` will use 16 bit floating point registers for the `A_reg` variable. Also, the `Neon` argument in the `set_memory()` statements must be changed to `Neon8f`.

IV. PERFORMANCE EVALUATION

In this section we evaluate the performance of the GEMM BLIS-like routines and 8×12 microkernels comparing three different microkernel implementations: `Neon`, a neon-intrinsic, hand-developed microkernel; `BLIS`, the BLIS v.0.9 microkernel; and EXO, the automatically generated code presented in Section III. The experiments in this section were performed on a single core of the NVIDIA Carmel processor (ARM v8.2) embedded on an NVIDIA Jetson AGX Xavier board, using IEEE 32-bit floating point arithmetic (FP32).

The experiments include three types of GEMM problems: microkernel performance in a solo mode (including the edge cases); large square matrices; and highly "rectangular" problems.

A. Solo Mode

In this experiment, we show that the EXO generated micro-kernel is as good as the hand-coded ones. To do this, we run the micro-kernels directly for 5 seconds and then compute the GFLOPS. The 8×12 columns in Figure 13 show the performance when the micro-kernel is invoked at its maximum performance. For this test, we have set the K_c to 512, which is the value of BLIS packing for this ARM architecture. There are minor differences between the three solutions. First of all, `NEON` is slower than `BLIS`, and the main difference is that the former is written with Neon intrinsics while the latter is in assembly. EXO is slightly better because it only supports the

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on April 02,2024 at 18:47:12 UTC from IEEE Xplore. Restrictions apply.

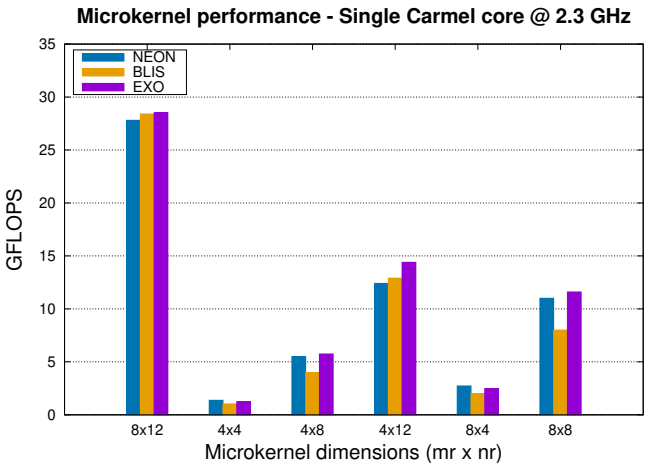

Fig. 13: Performance of different micro-kernels in solo-mode with different problem sizes. NEON, and BLIS use the same micro-kernel for all scenarios while EXO uses an ad-hoc auto-generated micro-kernel for each one.

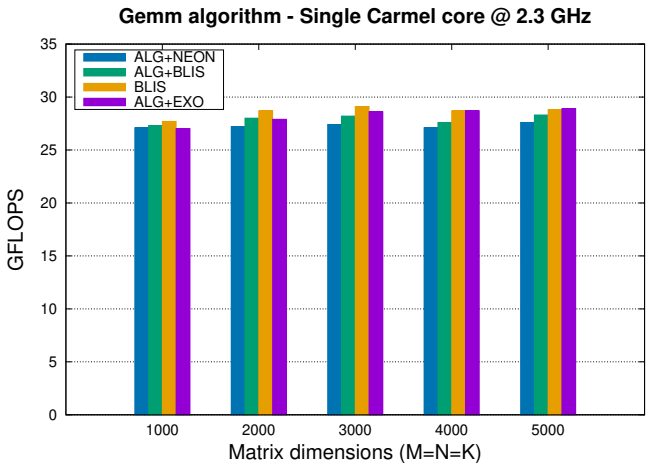

Fig. 14: Performance of different micro-kernels for squarish GEMM. NEON, and BLIS use their unique micro-kernel while EXO uses a set of auto-generated micro-kernels.

exact case of 8×12, while the other two micro-kernels also include the logic for the edge cases.

The other columns represent the performance when calling the micro-kernels with different edge cases. While NEON and BLIS run the same micro-kernel that in the case of 8×12, EXO benefits from the easy way to generate different micro-kernel sizes and therefore a specialized micro-kernel is run for each size of the problem. This approach is clearly the best solution for overcoming edge cases.

B. Squared Matrices

Figure 14 shows the performance for a complete execution of the GEMM algorithm with the micro-kernels. The columns with the prefix ALG+ indicate that we have used a BLIS-like realization of the GEMM algorithm, which also includes the theoretical model presented in [9] for optimizing the packings. The suffix of these columns indicates the employed micro-kernel (or micro-kernels for EXO) used. In addition, the column labeled as BLIS is the performance when calling the GEMM function of the BLIS library.

BLIS performs better in this case because the GEMM algorithm used in the BLIS library implements prefetching inside the micro-kernel that is not used in the ALG+BLIS approach. ALG+EXO outperforms other ALG+ and considering that the packings of the micro-kernel operands are equal due to the model, the only difference is the use of different micro-kernels (EXO) or one micro-kernel for all cases. Specifically, for the EXO approach we have used the 8×4 micro-kernel for the 1,000 and 4,000 problem sizes and the 8×8 micro-kernel for the 2,000 and 5,000 problem sizes.

C. Rectangular Matrices

Given the current interest in DL inference, the dimensions of this experiment are chosen to be those obtained by applying the IM2ROW transform [25] to the convolution layers in the ResNet50 v1.5 and VGG16 deep neural network (DNN) models in the form of a GEMM. The "batch" size for the inference scenario is set to 1 sample. Since some layers share the same parameters, resulting in GEMM problems of the exact dimensions, we report the results for these only once; see Table I and Table II for reference.

TABLE I: DIMENSIONS OF THE GEMM RESULTING FROM APPLYING THE IM2ROW TRANSFORM TO THE LAYERS OF THE RESNET50 V1.5 MODEL WITH A BATCH SIZE OF 1.

Layer id.	Layer numbers ResNet50 v1.5	m	n	k
1	001	12,544	64	147
2	006	3,136	64	64
3	009/021/031	3,136	64	576
4	012/014/024/034	3,136	256	64
5	018/028	3,136	64	256
6	038	3,136	128	256
7	041/053/063/073	784	128	1,152
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	2,304
13	086/098/108/118/128/138	196	1,024	256
14	088	196	1,024	512
15	092/102/112/122/132	196	256	1,024
16	142	196	512	1,024
17	145/157/167	49	512	4,608
18	148/160/170	49	2,048	512
19	150	49	2,048	1,024
20	154/164	49	512	2,048

Figure 15 reflects the advantages of ad-hoc micro-kernels for edge cases. The ALG+EXO implementation is the best option for 9 out of 20 layers for ResNet50 v1.5, while BLIS with prefetching is the best for 6 of them. For this execution, ALG+EXO uses the micro-kernels 8×12, 8×4, 4×4, 4×8, 4×12, 1×8, and 1×12.

To put these results in terms of absolute performance, Figure 16 shows the aggregated time for the entire model execution. Although the difference is small the best performance is achieved by ALG+EXO, followed by BLIS, ALG+BLIS,

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on April 02,2024 at 18:47:12 UTC from IEEE Xplore. Restrictions apply.

TABLE II: DIMENSIONS OF THE GEMM RESULTING FROM APPLYING THE IM2ROW TRANSFORM TO THE LAYERS OF THE VGG16 MODEL WITH A BATCH SIZE OF 1.

Layer id.	Layer numbers VGG16	m	n	k
1	01	50,176	64	27
2	03	50,176	64	576
3	06	12,544	128	576
4	08	12,544	128	1,152
5	11	3,136	256	1,152
6	13/15	3,136	256	2,304
7	18	784	256	2,304
8	20/22	784	512	4,608
9	25/27/29	196	512	4,608

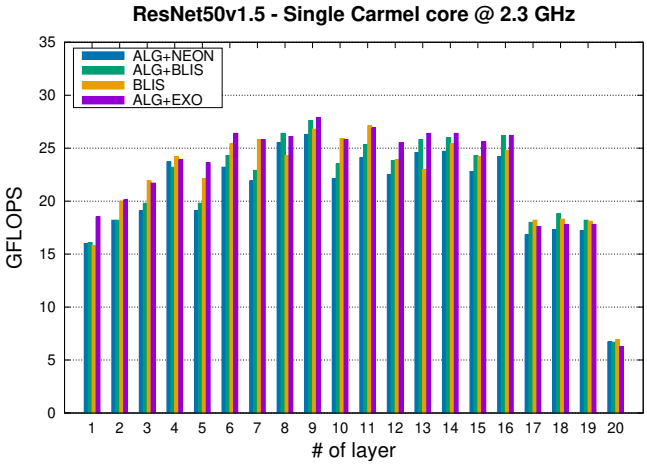

Fig. 15: Performance per layer of ResNet50 v1.5 model.

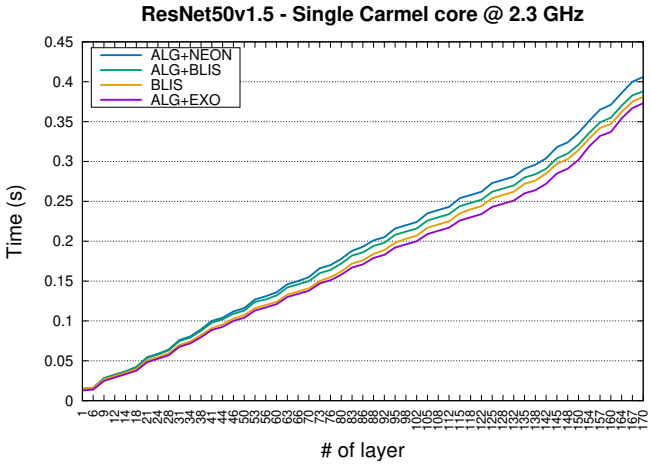

Fig. 16: Aggregated inference time for ResNet50 v1.5 model.

and ALG+Neon. Figure 17 also improves the use of EXO, which is the best for 3 layers, BLIS with prefetching in the other four of them, while the ALG+BLIS is the best on two of them. In terms of aggregated time, Figure 18 shows that the performance of ALG+EXO and BLIS solutions are close.

V. CONCLUSIONS

In this paper, we have addressed the problem of the monolithic approach taken by LA libraries by generating specific,

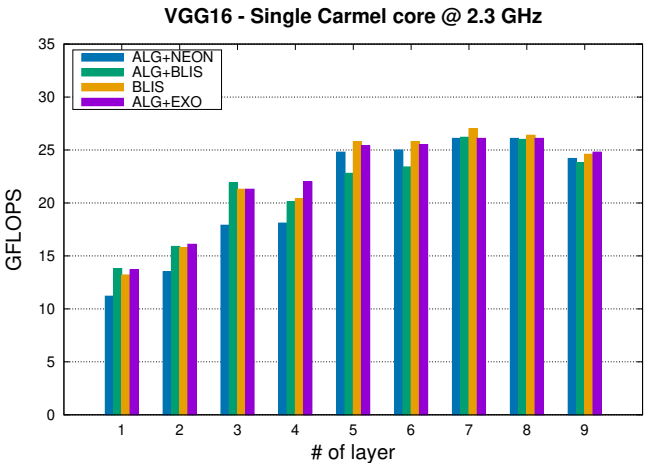

Fig. 17: Performance per layer of VGG16 model.

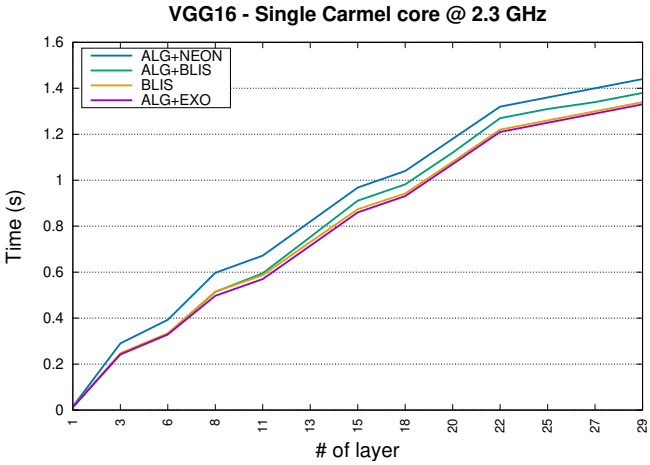

Fig. 18: Aggregated inference time for VGG16 model.

hardware-oriented micro-kernel C-code using the EXO tool. We have described in a step-by-step process how to build a micro-kernel generator from scratch that produces a C-code that performs close to (or even better than) the well-known code in the BLIS library. We have also provided hints for adapting the micro-kernel generator to meet different software requirements such as different data types. We have analyzed this performance comparison in three different scenarios: micro-kernel execution, large, squarish matrix multiplications, and rectangular GEMM generated by the DL convolutional models against Neon intrinsics-based and assembly micro-kernels. The micro-kernel generator is available at https://github.com/adcastel/CGO_GEMM_ukernels_exo_artifact. In addition, this work has contributed to the EXO tool code with the support for some ARM features.

As future work, we will adapt and analyze the micro-kernel generator tool with other architectures such as Intel, RISC-V, or the matrix engine. In addition, we plan to tackle the generation of other pieces of code for LA libraries or specific domain codes such as convolutional codes.

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on April 02,2024 at 18:47:12 UTC from IEEE Xplore. Restrictions apply.

VI. Data-Availability Statement

The artifact code is available in [26].

Acknowledgments

We thank Prof. Gilbert Bernstein from the University of Washington and Prof. Jonathan Ragan-Kelley from Massachusetts Institute of Technology for their active collaboration. A. Castelló is a FJC2019-039222-I fellow supported by MCIN/AEI/10.13039/501100011033. Y. Ikarashi is supported by the Funai Overseas Scholarship, Masason Foundation, and Great Educators fellowships. H. Martínez is a postdoctoral fellow supported by the *Consejería de Transformación Económica, Industria, Conocimiento y Universidades de la Junta de Andalucía*.

Appendix

A. Abstract

This appendix documents the EXO_ukr_generator software artifact and the procedure of how to install and execute it for the reproduction of the results shown in the paper. This software package is aimed at reproducing the overall sections of the paper including both the step-by-step micro-kernel design and the experimental results. This software is designed and configured for ARMv8 processors because it is the hardware used in the paper. However, it can be executed over ARM processors with the support of Neon intrinsics instructions The artifact is publicly available at https://github.com/adcastel/CGO_GEMM_ukernels_exo_artifact and includes a series of scripts for building the environment and executing the experimentation.

B. Artifact Check-list (meta-information)

- **Algorithm:** High-performance matrix multiplication
- **Compilation:** Exo compiler
- **Hardware:** ARM Neon (v8)
- **Metrics:** GFLOPS
- **Experiments:** Stand-alone micro-kernels and deep neural networks GEMM
- **How much time is needed to prepare workflow (approximately)?:** 20 minutes
- **How much time is needed to complete experiments (approximately)?:** 10 minutes
- **Publicly available?: Available at** https://github.com/adcastel/CGO_GEMM_ukernels_exo_artifact
- **Code licenses (if publicly available)?:** None

C. Description

1) How delivered: This artifact is available via the https://github.com/adcastel/CGO_GEMM_ukernels_exo_artifact Github repository without any license. This feature is indicated in the REQUIREMENTS.txt file.

2) Hardware Dependencies: The artifact is available for ARM v8 (Neon instructions support) and it has been tested in NVIDIA Xavier, Orin, and Nano platforms. This feature is indicated in the REQUIREMENTS.txt file.

3) Software Dependencies: This artifact comprises Python and C code, and therefore, versions of `Python3.9` and `gcc-10` (or higher) are mandatory. It is possible that there are some Python packages that are not currently installed. This possible scenario has been treated in the configuration scripts, however, package failure may occur. The software for the Exo and the Blis libraries is also included in the package. This feature is indicated in the REQUIREMENTS.txt file. For the plotting procedure, this artifact uses `gnuplot` tool.

4) Data Sets: The data sets for the experimentation are included in the repository code so there is not any extra requirement for them.

D. Installation

1) For the installation, we first need to clone the repository with the command:

   ```
   git clone https://github.com/adcastel/CGO_GEMM_ukernels_exo_artifact
   ```

2) Then we need to enter the directory with:

   ```
   cd CGO_GEMM_ukernels_exo_artifact
   ```

3) and execute the `build.sh` script as:

   ```
   source build.sh
   ```

   This script will check the existent compilers and will build and install the Blis and the Exo software as well as set the environment variables.

E. Experiment Workflow

For the micro-kernel generation, we only need to execute the script as follows:

```
./microkernel_generator.sh
```

This script will generate the micro-kernel explained in Section III of the paper, showing the generated code in each step of the process as shown in the section mentioned above.

F. Evaluation and Expected Result

For the evaluation reproduction, we need to use two different scripts corresponding to the different evaluations in the paper.

1) For the evaluation of the micro-kernel in solo-mode we need to execute the corresponding script as follows:

   ```
   ./execute_ukernel_solo.sh
   ```

   This script will execute the experiments shown in Figure 13.

2) For reproducing the experiments of figures 14-18 we should execute the following script as:

   ```
   ./execute_algorithm.sh
   ```

   This evaluation is the one that consumes more time due to the different combinations of algorithm, micro-kernel, and GEMM sizes (including both square and DL models).

3) For generating the plots, we should execute the plotting script as follows:

   ```
   ./do_plots.sh
   ```


Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on April 02,2024 at 18:47:12 UTC from IEEE Xplore. Restrictions apply.

The plots of the paper will be located in the `plots` folder.

Please, notice that the user can build and execute the artifact using just one script:

```
./build_and_execute_all.sh
```

This script will build the environment and execute all the experiments on their own.

G. Experiment Customization

This artifact can be customized in several ways. The user is able to modify the generated micro-kernel or the evaluation of the GEMM.

In the case of the micro-kernel generation, the user should modify the `generator.py` file inside the `EXO_ukr_generator` directory. There, the user can set the values of M_R and N_R (that are the sizes of the micro-kernel) and additionally the datatype. Please, notice that in that file, the first call is the one used in Section III of the paper while there are other commented lines that can be used for generating different micro-kernel sizes.

If the user wants to change the experimental setup for the algorithm+micro-kernel evaluation, one needs to change the files in `CGO_GEMM_ukernels_exo_artifact/gemm_blis_family/cnn_models/` or to generate a new input file and then add the corresponding line to the `execute_algorithm.sh` file.

Please, notice that the overall evaluation of a different micro-kernel (of those that are used in the paper) will involve different changes in every test/configuration. Therefore, the "random" attempts with non-usual micro-kernel sizes are not recommended without the guidance of the artifact developers.

REFERENCES

[1] 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, May 2008.

[2] Z. Xianyi, W. Qian, and Z. Yunquan, "Model-driven level 3 BLAS performance optimization on Loongson 3A processor," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*, 2012.

[3] 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.

[4] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proc. ACM/IEEE Conference on Supercomputing*, ser. SC '98. USA: IEEE Computer Society, 1998, p. 1–27.

[5] 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.

[6] 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.

[7] F. G. V. Zee, T. M. Smith, B. Marker, T. M. Low, R. A. V. D. Geijn, F. D. Igual, M. Smelyanskiy, X. Zhang, M. Kistler, V. Austel, J. A. Gunnels, and L. Killough, "The BLIS framework: Experiments in portability," *ACM Trans. Math. Softw.*, vol. 42, no. 2, June 2016. [Online]. Available: https://doi.org/10.1145/2755561

[8] Y. Ikarashi, G. L. Bernstein, A. Reinking, H. Genc, and J. Ragan-Kelley, "Exocompilation for productive programming of hardware accelerators," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 703–718. [Online]. Available: https://doi.org/10.1145/3519939.3523446

[9] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Ortí, "Analytical modeling is enough for high-performance BLIS," *ACM Trans. Math. Softw.*, vol. 43, no. 2, pp. 12:1–12:18, Aug. 2016.

[10] T. Moreau, T. Chen, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "VTA: an open hardware-software stack for deep learning," *CoRR*, vol. abs/1807.04188, 2018. [Online]. Available: http://arxiv.org/abs/1807.04188

[11] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.

[12] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, and D. Qian, "The deep learning compiler: A comprehensive survey," *CoRR*, vol. abs/2002.03794, 2020. [Online]. Available: https://arxiv.org/abs/2002.03794

[13] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "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, 2013, p. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[14] C. Lattner, J. A. Pienaar, M. Amini, U. Bondhugula, R. Riddle, A. Cohen, T. Shpeisman, A. Davis, N. Vasilache, and O. Zinenko, "MLIR: A compiler infrastructure for the end of moore's law," *CoRR*, vol. abs/2002.11054, 2020. [Online]. Available: https://arxiv.org/abs/2002.11054

[15] 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

[16] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Trans. Graph.*, vol. 35, no. 4, jul 2016. [Online]. Available: https://doi.org/10.1145/2897824.2925952

[17] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, "Ansor: Generating High-Performance Tensor Programs for Deep Learning," arXiv, Tech. Rep., arXiv:2006.06762 [cs, stat] type: article. [Online]. Available: http://arxiv.org/abs/2006.06762

[18] 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

[19] K. Yotov, X. Li, M. J. Garzarán, D. Padua, K. Pingali, and P. Stodghill, "Is search really necessary to generate high-performance BLAS?" *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, 2005.

[20] A. Olivry, G. Iooss, N. Tollenaere, A. Rountev, P. Sadayappan, and F. Rastello, "IOOpt: automatic derivation of i/o complexity bounds for affine programs," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, jun 2021. [Online]. Available: https://doi.org/10.1145/3453483.3454103

[21] Q. Huang, M. Kang, G. Dinh, T. Norell, A. Kalaiah, J. Demmel, J. Wawrzynek, and Y. S. Shao, "Cosa: Scheduling by constrained optimization for spatial accelerators," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 554–566.

[22] U. Bondhugula, "High performance code generation in MLIR: an early case study with GEMM," *CoRR*, vol. abs/2003.00532, 2020. [Online]. Available: https://arxiv.org/abs/2003.00532

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

[24] G. Alaejos, A. Castelló, H. Martínez, P. Alonso-Jordá, F. D. Igual, and E. S. Quintana-Ortí, "Micro-kernels for portable and efficient matrix multiplication in deep learning," *The Journal of Supercomputing*, pp. 1–24, 2022.

[25] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *International Workshop on Frontiers in Handwriting Recognition*, 2006.

[26] A. Castelló, J. Bellavita, G. Dinh, Y. Ikarashi, and H. Martínez, "Exo microkernel generator artifact," ACM, 2024. [Online]. Available: https://doi.org/10.1145/3580428

Authorized licensed use limited to: UNIVERSITY OF ALBERTA. Downloaded on April 02,2024 at 18:47:12 UTC from IEEE Xplore. Restrictions apply.