A predictable SIMD library for GEMM routines

Iryna De Albuquerque Silva[1], Thomas Carle[2], Adrien Gauffriau[3], Victor Jegu[3], Claire Pagetti[1]
[1]*ONERA, France*
[2]*IRIT - Univ. Toulouse 3, France*
[3]*Airbus, France*

***Abstract*—The resource-constrained environment and the certification requirements underlying embedded safety-critical real-time systems impose an adapted development process for software applications. In this work, we propose an efficient and traceable implementation of an existing blocked general matrix multiplication (GEMM) algorithm. We target time-predictability in a COTS processor with single-instruction multiple-data (SIMD) extensions. We provide a set of rules for tuning the algorithm parameters and predict with precision its number of memory accesses and cache misses, which paves the way for a static WCET analysis. Our experiments show that time-predictability comes at the cost of a performance degradation of only 2.54% on average. Moreover, tuning the parameters allows for reducing cache misses by up to 60% in certain parts of the algorithm.**

***Index Terms*—Real-Time, GEMM, SIMD, cache analysis.**

I. Introduction

Basic Linear Algebra Subprograms (BLAS) libraries [1] consist of highly optimized algorithms to perform common linear algebra operations, including the general matrix multiplication (GEMM) routine. This routine is at the core of traditional scientific simulations as well as the most recent machine learning tasks, in particular to implement convolutions.

Context: While GEMM optimization benefits from years of research in the high-performance computing (HPC) domain, a different approach is required for its use in the embedded safety-critical real-time sphere. Indeed, the development process of software applications in safety-critical real-time systems must follow safety standards throughout the construction, validation, and verification (V&V) development cycle. In the avionics domain, the DO-178C [2] involves conducting reviews and analyses of high-level and low-level requirements, software architecture, and source code; but also explicitly imposes deriving tight worst-case execution time (WCET) bounds [3]. This raises the challenge of developing predictable software that achieves high performance levels, by efficiently exploiting the available hardware resources.

Motivation: The authors of [4] proposed a first GEMM implementation using vector extensions in real-time safety-critical systems. However, they do not implement the full GEMM algorithm with matrix blocking and do not detail how predictability can be ensured. In this work, we have three objectives, the first being to implement the blocked GEMM algorithm. Second, we want strong guarantees on the WCET estimation, and static WCET analysis [5] is a trustful way to compute such upper bounds. However, these methods require among other things to determine if instructions lead to miss or hit in the cache hierarchy. If approaches exist for the instruction cache, the analysis is harder for data caches as it is necessary for each memory instruction to determine which range of addresses it can access, then to determine the possible access patterns, and finally to combine these information to determine a conservative bound on the number of misses caused by the memory instruction. Each of these steps is complex and usually degrades the precision of the analysis [6]. Finally, our last objective is to allow the GEMM algorithm parameters to be tuned to improve the predictability.

This work benefited from the AI Interdisciplinary Institute ANITI, which is funded by the French "Investing for the Future-PIA3" program under the Grant agreement ANR-19-P3IA-0004. It was also supported by a grant overseen by the French National Research Agency (ANR) as part of the MeSCAliNe (ANR-21-CE25-0012) project and by the DGAC PHYLOG 2 project.

Contributions: We target single-core processors equipped with vector (SIMD) extensions. The implementation is single-threaded and bare-metal. We study, adapt and implement an existing architecture-aware GEMM algorithm focusing notably on 1) removing compiler optimizations, while still trying to achieve an efficient implementation that leverages vector extensions and, 2) exploiting the memory hierarchy effectively and predictably. In addition, our objective is to provide analytical formulae of a tight upper bound on the number of cache misses and of the number of memory accesses of the GEMM algorithm, in order to later use these numbers in e.g. the Implicit Path Enumeration Technique (IPET) to tighten the WCET estimation. These formulae are obtained using our detailed knowledge and understanding of the memory access patterns implemented in these routines. We only targeted an L1 data cache since the L2 cache on our target platform is not predictable, but these formulae can be extended to support any number of layers in a cache hierarchy, as long as each cache has a predictable replacement policy. Our experiments show that time-predictability in our code comes at the cost of a performance degradation of only 2.54% on average.

The outline of the paper is as follows. Section II introduces the GEMM algorithm on which our implementation is based. Section III discusses related work. Section IV presents the target processor and accompanying concepts for the experimental part. Section V describes and evaluates the optimizations performed to obtain a predictable yet efficient implementation of the algorithm. Section VI details our formulae, together with the experimental results. Finally, Section IX provides concluding remarks.

Algorithm 1: Blocked GEMM as introduced in GotoBLAS [7]

```
     Parameters: $n_c$, $m_c$, $k_c$, $m_r$, $n_r$
     Input: Matrix A of shape $(m, k)$, Matrix B of shape $(k, n)$
     Input/Output: Matrix C of shape $(m, n)$
(L1) for $j_c = 0$ to $n - 1$ step $n_c$ do
(L2)   for $p_c = 0$ to $k - 1$ step $k_c$ do
E1       $\tilde{B}c = B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1)$;   // Packing Bc
(L3)     for $i_c = 0$ to $m - 1$ step $m_c$ do
E2         $\tilde{A}c = A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1)$; // Packing Ac
(L4)       for $j_r = 0$ to $n_c - 1$ step $n_r$ do          // Macro-kernel
(L5)         for $i_r = 0$ to $m_c - 1$ step $m_r$ do
(L6)           for $p_r = 0$ to $k_c - 1$ step 1 do         // Micro-kernel
E3               $Cc(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$ +=
                   $\tilde{A}c(i_r : i_r + m_r - 1, p_r)\tilde{B}c(p_r, j_r : j_r + n_r - 1)$;
               end
             end
           end
         end
       end
     end
```

II. ARCHITECTURE-AWARE GEMM ROUTINE

The GEMM routine computes $C \leftarrow \alpha A \cdot B + \beta C$ where C, A and B are matrices and α and β are scaling factors. For the applications targeted in this study, we consider $\alpha = 1$ and $\beta = 1$. The core optimization principle of GEMM consists in: 1) performing a *blocked matrix multiplication* by dividing A, B, and C into submatrices, or *blocks*, such that 2) those blocks fit in the different levels of cache in order to improve spatial and temporal locality. To further favor locality, the blocks are copied in *copy blocks* so that elements are contiguous in memory and can be efficiently accessed. This copy is called *packing routine* in the literature.

An optimized GEMM algorithm is described in [7] and is the base of the implementation of many popular BLAS libraries, including OpenBLAS [8] and BLIS [9]. The algorithm is parameterized by five parameters to be tuned by the designer depending on the hardware. More precisely, three blocking parameters n_c, k_c, and m_c are chosen according to cache memory features, and the m_r and n_r parameters are chosen in view of the processor registers. Given these parameters, the algorithm (see pseudo-code in Algorithm 1) essentially consists of six nested loops around an outer product that is highly optimized to update each time a small *tile* of C residing in registers.

Blocking for caches. The partitioning of A, B and C into submatrices targeting cache layers is performed by the three outermost loops. Loop $L1$ traverses matrices C and B along the n dimension (i.e. the columns) and partitions them in column panels of shapes $m \times n_c$ and $k \times n_c$ respectively. The second loop $L2$ operates on the k dimension and divides matrix A into column panels of shape $m{\times}k_c$, as well as divides a column panel of shape $k \times n_c$ of B into blocks B_c of shape $k_c \times n_c$. Then, loop $L3$ further partitions a column panel of shape $m \times k_c$ of A into blocks A_c of shape $m_c \times k_c$. Finally, the current column panel of C is also divided into blocks C_c of shape $m_c{\times}n_c$. The blocks are shown in Figure 1. Each B_c and A_c block is copied into $\tilde{B}_c$ and $\tilde{A}_c$ with a *packing routine*.

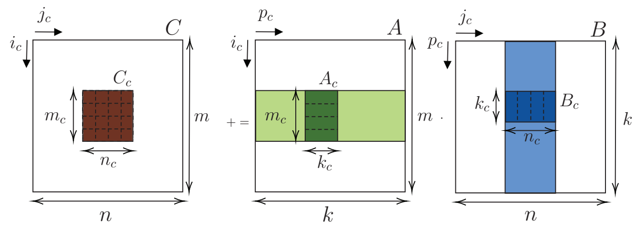


Fig. 1. Partitioning of matrices A, B and C, all stored in row-major order. A block C_c is computed using the whole row panel with blocks A_c and the column panel with blocks B_c. The dashed lines represent the micro-tiles C_r in C_c as well as the micro-panels A_r and B_r in A_c and B_c, respectively.

This step is detailed later.

Macro-kernel. The macro-kernel (Loops $L4$ to $L6$) computes the block C_c of shape $m_c \times n_c$ as the result of the accumulation of the previous value of C_c and the multiplication of $\tilde{A}_c$ and $\tilde{B}_c$. Note that the complete computation of one block C_c requires calling the macro-kernel several times as it results from the multiplication of all blocks A_c of the green row of Figure 1 by the blocks B_c composing the blue column. Within a macro-kernel, loop $L4$ partitions a copy block $\tilde{B}_c$ into *micro-panels* named B_r of shape $k_c \times n_r$ and loop $L5$ partitions a copy block $\tilde{A}_c$ into *micro-panels* named A_r of shape $m_r \times k_c$. A block C_c is divided into *micro-tiles* C_r of shape $m_r \times n_r$. Figure 4, which appears further in the text as it helps understand cache-related formulae, illustrates the order in which the micro-tiles C_r are computed and which micro-panels are involved.

Micro-kernel. The micro-kernel (loop $L6$) is the last level of decomposition as it updates a micro-tile C_r, via a sequence of k_c outer products (a vector-scalar multiplication). At each iteration, it computes the product of one column of a micro-panel A_r, i.e., a vector of shape $m_r \times 1$, and one row of a micro-panel B_r, i.e., a vector of shape $1 \times n_r$. The sizes of these column and row vectors (m_r and n_r) are chosen to take advantage of the vector registers of the architecture. Figure 2 illustrates the operation that is performed inside the micro-

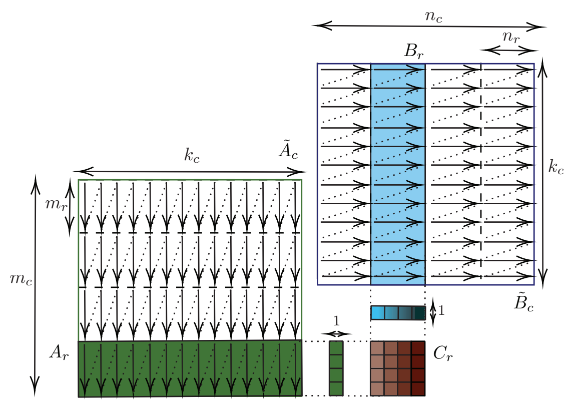


Fig. 2. Result of the packing routines to create copy blocks $\tilde{A}_c$ and $\tilde{B}_c$. The arrows represent the order in which the elements are laid out in memory and accessed in the micro-kernel routine. A micro-panel A_r is stored in column-major order, with a leading dimension m_r, while a micro-panel B_r is stored in row-major order, with a leading dimension n_r.

kernel, with an example where $m_r = n_r = 4$.

Packing routines of A and B. The motivation for packing (copying and rearranging) A_c and B_c to create copy blocks $\tilde{A}_c$ and $\tilde{B}_c$ is to favor cache locality and thus increase performance. Concretely, micro-panels A_r of A_c are copied and stored in $\tilde{A}_c$ in column-major order, with a *leading dimension* m_r. Column-major [10] means: elements within a column are stored sequentially in memory, each column is read one after the other and the leading dimension is the stride between consecutive columns. Similarly, each micro-panel B_r of $\tilde{B}_c$ is arranged in row-major order, with a *leading dimension* n_r. Likewise, row-major means: elements within a row of B_r are stored sequentially and the stride between consecutive rows is n_r. Figure 2 shows the memory layout of copy blocks and the benefit for the micro-kernel.

For more details about the blocked GEMM algorithm the reader is referred to [9] and [11][1].

III. Related Work

A. Optimized implementation of GEMM-based algorithms

Since our focus is on embedded targets with vector extensions we highlight works that studied optimized implementations of GEMM-based algorithms on ARM cores.

Vanilla GEMM. The authors of [4] propose an implementation of a non-blocked GEMM algorithm in ARM processors with vector extensions. They explicitly employ vector instructions (via NEON intrinsics), in opposition to compiler-generated vectorization, and show great gains in performance compared to the non-vectorized versions of the algorithm. However, they also observe that when the matrices are too big to fit in the L1 data cache, the potential of vector registers is not fully exploited. Our work can be considered an extension of theirs, as we follow their guidelines on SIMD usage in real-time safety-critical systems and go further on exploiting the hardware architecture for performance and predictability.

Blocked GEMM. The authors of LIBXSMM [12] propose optimized implementations of the blocked GEMM algorithm for small matrices based on *just-in-time* compilation to only build the required micro-kernel variant at run time. This idea is very powerful but not suitable for our application domain as we must privilege a static approach. In contrast, BLASFEO [13] consists of a static library designed to optimize small and irregularly-shaped matrix multiplications. The authors adopt the OpenBLAS GEMM algorithm principles but propose modifications such as converting the layout of matrices to improve cache locality and conditional use of packing routines, as they may be too expensive in small workloads. LibShalom [14] extends the library approach of [13] to perform packing in parallel to computations and also proposes an instruction scheduling optimization targeting ARM v8 multi-core processors. However, these works only target performance improvement and rely notably on intensive compiler optimizations. In addition, LibShalom uses OpenMP for parallel execution. We thus consider that their ideas about conditional packing and customized matrices layouts can be adapted to our work scope as future work.

[1]The figures in this section were also inspired by the ones found in [11].

Direct convolution using blocked GEMM ideas. Recent works have proposed a loop-blocking strategy to optimize the original direct convolution algorithm. A 2D-convolution direct algorithm is a 7-dimensional nested loop, thus an architecture-aware loop-tiling strategy such as the one used for GEMM can be derived. The work of [15] introduces a strategy for mapping loops to the architecture and proposes new data layouts for the input and convolution weights tensors so that data is accessed in a cache-friendly way. In [16] the authors adapted the blocking strategy presented in [15] to allow for preserving the conventional *NHWC* data layout used in convolution layers and also presented a variant capable of exploiting GEMM micro-kernels. Both works target high performance on ARM v8 cores. In future work, we aim to extend our work to this application domain and add a predictability perspective to their approach.

B. Choice of blocking parameters for GEMM algorithm

One can perform empirical searches to find the best matrix block shapes for a given hardware and matrices configurations, as is done in [17]. Although pertinent, this method is very time-consuming. The authors of [11] develop an analytical approach based on the modeling of the target architecture to tune the parameters of the GEMM kernels with a focus on single-threaded implementation. It is the methodology that we adopted and will further discuss in Section V. Finally, some works investigate the use of an auto-tuning framework to search for the optimal parameters, such as [18], [19] and [20]. In the work of [21] the authors enhance the auto-tuning approach by relying on a structured search space that embeds expert knowledge about the target and imposes divisibility constraints on tile sizes and unroll factors to exclude non-feasible cases. This work can be complemented by ours with the addition of predictability constraints in their selection of blocking parameters.

IV. Experimental setup

A. Presentation of the target processor

For our study, we conducted experiments on the Texas Instruments KEYSTONE II SoC [22], which integrates a Cortex-A15 processor quad-core cluster, implementing the ARM v7-A architecture. The studied ARM Cortex-A15 has a Level 1 data cache (L1D), and a Level 1 instruction cache (L1I) of size 32KB ($size_{L1}$) each. Both L1 caches have an associativity degree (w_{L1}) of 2, a cache line size (c_{L1}) of 64B, a number of sets (s_{L1}) of 256, and implement a Least Recently Used (LRU) cache replacement policy. The Level 2 cache (L2) of 4MB ($size_{L2}$) is shared among the four cores. It has a 16-way set-associative cache structure (w_{L2}), with a fixed cache line size of 64B (c_{L2}), 4,096 cache sets (s_{L2}), and a random replacement policy.

B. SIMD instructions

Vector extensions are functional units that implement SIMD (single-instruction multiple-data) instructions, which process in parallel and independently distinct data elements arranged into vectors. Vector instructions have the same structure as a scalar instruction, i.e., a mnemonic, source register(s), and a destination register. The particularity is that vector registers can accommodate multiple values of different sizes and types. Thus unlike scalar instructions that operate on one element at a time, vector instructions exploit data-level parallelism.

In the remainder of this work, we consider the NEON instructions [23], which is the SIMD extension to the ARM v7-A instruction set, implemented in our target. It has 32 64-bit registers (D0-D31), which can also be seen as 16 128-bit registers (Q0-Q15). Each of the Q0-Q15 registers maps to a pair of D registers. Using the notation of [11], we note N_{VEC} as the number of elements of size s_{data} that a vector register can hold, which is a dimensionless quantity. Additionally, each vector instruction has a throughput of N_{INST}, expressed in $cycles^{-1}$, and a latency of L_{INST}, expressed in $cycles$.

C. Performance monitor unit

In order to obtain quantitative results for the performance of the GEMM implementation discussed in Section V as well as the cache usage analysis introduced in Section VI we used the *performance monitor unit* (PMU) [24] present in the target. The PMU architecture uses specific codes to identify events and then associates an event to a counter so that only one event is monitored at a time. The advantage of using PMU counters directly is that they are natively implemented and thus do not require any external monitoring feature, which guarantees accuracy and minimizes the impact of measures on the analyzed values. The studied core has six configurable event counters plus a specific cycle counter. Among the events we tracked are the number of cycles taken to execute a function, the L1D cache accesses and the L1D cache refills.

Our measurement methodology consisted of (1) selecting the event counter to use and assign an event to track; (2) resetting existing counter value; (3) reading the selected counter register; (4) executing the code portion under analysis and (5) reading the final value in the selected counter register.

V. Efficient implementation of GEMM routines

In this section, we discuss our methodology to attain an efficient yet predictable and traceable implementation of the GEMM algorithm presented in Section II. We note that the different parts of the GEMM algorithm, namely packing A_c, packing B_c, macro-kernel, and micro-kernel, are implemented as separate functions. It allows us to provide a library with modular code, enables reusability, and facilitates the study of the software behavior.

A. Determination of blocking parameters

In practice, the blocking parameters n_c, k_c, m_c, m_r and n_r are adjusted taking into account the latency of the vector units, the number and size of vector registers, and the size/associativity degree of the cache levels. The objective is that a micro-panel B_r (of shape $k_c \times n_r$) resides in the L1 cache, so it can be rapidly accessed by $\left\lceil \frac{m_c}{m_r} \right\rceil$ successive calls to the micro-kernel, as it is reused. Similarly, it is convenient that the copy block $\tilde{A_c}$ (of shape $m_c \times k_c$) stays in the L2 cache as its micro-panels are needed for the entirety of the macro-kernel function. Finally, the copy block $\tilde{B_c}$ (of shape $k_c \times n_c$) is suitable to reside in the L3 cache if one is present in the architecture.

For the following experiments, we applied the formulae of [11] to analytically determine the convenient GEMM parameters. For our target, we have $N_{\text{VEC}} = 4$ for a single-precision floating-point implementation, $N_{\text{VMLA}} = 0.5$ for the throughput of the VMLA instruction and $L_{\text{VMLA}} = 8$ for its latency. Thus, the formulae provide the following parameter values:

$$m_r = \left\lceil \frac{\sqrt{N_{\text{VEC}} \cdot L_{\text{VMLA}} \cdot N_{\text{VMLA}}}}{N_{\text{VEC}}} \right\rceil \cdot N_{\text{VEC}} = 4$$

$$n_r = \left\lceil \frac{N_{\text{VEC}} \cdot L_{\text{VMLA}} \cdot N_{\text{VMLA}}}{m_r} \right\rceil = 4$$

$$k_c = \frac{s_{L1} \cdot c_{L1}}{2 \cdot m_r \cdot s_{data}} = 512$$

$$m_c = \frac{(w_{L2} - 2) \cdot s_{L2} \cdot c_{L2}}{k_c \cdot s_{data}} = 1,792$$

Since the ARM Cortex-A15 does not have an L3 cache, this methodology claims that the value of n_c is not crucial. The only advice is to choose a number big enough to avoid redundant partitioning of matrices B and C, and that is a power of two, for cache alignment reasons. We thus adopted $n_c = 4,096$. More details on how the formulae are obtained can be found in [11].

B. Optimizations in source code

As explained in Section I, the certification process requires the guarantee that the designed source code and the final binary have the same behavior, thus a certain level of traceability of the transformations performed between the two representations is required. To achieve this task one can rely on formally verified compilers [25] or depend on generic widely-used compilers, but without using any of their optimizations.

We decided to use the GCC compiler, because of its popularity and support for various architectures, including our target. However, when no optimizations are allowed, the compiler produces binaries with very inefficient register allocation, resulting in many redundant *store* and *load* operations to and from the stack. Such behavior is detrimental to traceability, as an instruction in source code is mapped to various instructions in binary, and efficiency, as the extra memory accesses significantly increase execution time. To remedy this, we proceeded to carefully place all the intermediate variables directly in registers and, whenever possible, inline functions to limit stack usage. This approach improved the code while using at most 10 general-purpose registers simultaneously, which is less than the number of general-purpose registers available in the ARM v7 architecture.

TABLE I
MEASURED EXECUTION TIMES, IN CYCLES, ON AN ARM CORTEX-A15 WITH -O0 FLAG. DIFFERENCE W.R.T. *Base* IS GIVEN IN PARENTHESES.

Matrix configuration				**GEMM implementation**						
m	n	k		**Base**	**First optimized**		**Optimized with intrinsics**		**Optimized in assembly**	
272	272	272	mean	2 543 708 367	988 682 239	(-61.13%)	2 220 393 123	(-12.71%)	235 168 329	(-90.75%)
			min	2 543 704 067	988 676 103		2 220 386 698		235 153 669	
			max	2 543 711 485	988 692 654		2 220 398 214		235 181 181	
			min-max var.	2.92e−4%	1.67e−3%		5.19e−4%		1.17e−2%	
528	528	528	mean	6 920 673 125	2 883 613 750	(-58.33%)	3 312 934 607	(-52.13%)	1 566 589 440	(-77.36%)
			min	6 920 653 603	2 883 100 642		3 312 863 994		1 566 565 684	
			max	6 920 698 753	2 888 548 217		3 312 956 562		1 566 603 968	
			min-max var.	6.52e−4%	1.89e−1%		2.79e−3%		2.44e−3%	
192	736	528	mean	5 789 246 930	3 622 514 818	(-37.43%)	3 875 069 543	(-33.06%)	778 924 693	(-85.55%)
			min	5 789 237 191	3 622 514 793		3 875 069 520		778 923 061	
			max	5 789 266 393	3 622 514 892		3 875 069 565		778 926 162	
			min-max var.	5.02e−4%	2.73e−6%		1.16e−6%		3.98e−4%	
256	784	2,016	mean	2 120 705 709	579 964 569	(-72.65%)	857 163 412	(-59.58%)	134 562 545	(-93.65%)
			min	2 120 700 674	579 962 833		857 161 851		134 561 357	
			max	2 120 709 745	579 966 116		857 165 391		134 563 809	
			min-max var.	4.28e−4%	5.66e−4%		4.13e−4%		1.82e−3%	

C. *Micro-kernel implementation in assembly*

The NEON instructions can be accessed through special functions called NEON intrinsics. Intrinsic functions and data types provide access to low-level NEON functionality at C (or C++) source code level. These allow the creation of C variables that directly map to NEON registers, allowing NEON vectors to be passed as function arguments or return values. Then, in the compilation toolchain, the function calls are replaced by the appropriate NEON instruction(s). Unfortunately, in our setting where the use of compiler optimizations is limited, the generated binary includes many redundant instructions and a great amount of register spills. For the totality of a micro-kernel code, 397 extra instructions are identified in the binary when using NEON intrinsics.

To circumvent the aforementioned issues, we proceeded to manually implement the GEMM micro-kernel using inline assembly in order to completely control the selection of vector instructions and the register allocation. The micro-kernel function can have its instructions separated into three main subroutines in order to facilitate verification of correctness, which are: load the micro-tile C_r into registers, update the value of C_r in the registers and finally store it. In Algorithm 2 we expose the pseudo-code of the micro-kernel for $m_r = n_r = 4$, in single-precision floating-point (s_{data} = 4B). It means that every NEON Q register (128-bit) contains 4 floating-point elements.

D. *Evaluation of optimizations in* GEMM *implementation*

We evaluate the GEMM algorithm on square matrices and on irregular-shaped configurations that are common in deep learning workloads [26], [27]. Table I presents the measured execution times of the different GEMM implementations on an ARM Cortex-A15 with no compiler optimizations. For each experiment, the reported measures contain the mean, the maximum and the minimum observed times of 100 executions. The *Base* implementation is the one without any optimizations, and the *First optimized* contains register allocation and function inlining techniques. The last two implementations extend the *First optimized* version by using the NEON: *Optimized with intrinsics* uses the intrinsics functions whereas *Optimized in assembly* is the full optimized version. The percentages represent the reduction in the mean measured execution times with respect to the *Base* implementation.

We observe a great reduction in overheads (compared to the *Base* implementation) after the first optimization of the source code, with execution times reduced by 92.00% in the best case. The execution times are increased compared to the first optimization when using NEON intrinsics, contrary to what one could initially expect, due to the very inefficient register allocation performed by the compiler. Finally, when the micro-kernel is also manually optimized with the introduction of assembly blocks, we obtain performances that are comparable to a first level of compiler optimization (-O1 flag). We remark that the amount of improvement brought by the different optimizations is not constant but highly dependent on the configuration of the matrices. In addition, we observe a minimal timing variability in our bare-metal implementation, as evidenced by the reported minimum and maximum observed execution times.

VI. TIME-PREDICTABILITY OF EFFICIENT GEMM ROUTINES

The GEMM algorithm presented in Section II is intrinsically predictable: there is no conditional branching and all the

Algorithm 2: Micro-kernel implementation

```
$Q0, Q1, Q2, Q3 = VLOAD(Cr)$ // Load Cr into quad-registers
// Update Cr
for $p_r = 0$ to $k_c - 1$ step 1 do
    // Load $m_r$ (resp. $n_r$) elements of micro-panel $A_r$ (resp. $B_r$)
    $Q4 = VLOAD(A_r(0, p_r))$
    $Q5 = VLOAD(B_r(p_r, 0))$
    // Vector-scalar multiplications
    $Q0 = VMLA(Q4, Q5[0], Q0)$
    $Q1 = VMLA(Q4, Q5[1], Q1)$
    $Q2 = VMLA(Q4, Q5[2], Q2)$
    $Q3 = VMLA(Q4, Q5[3], Q3)$
end
// Store quad-registers values to Cr
$Cr = VSTORE(Q0, Q1, Q2, Q3)$
```

blocking parameters are known at compile time. However, to the best of our knowledge, no existing static analysis method is capable of detecting and leveraging precisely the memory access patterns that GEMM routines implement.

A. Philosophy of the approach

We perform an analytical study by deriving the formulae for the total number of memory accesses and defining an upper bound on the number of data cache misses in each part of the algorithm. We focus on these two numbers as they can be exploited in static WCET estimation techniques such as IPET in order to tighten the WCET bound. More precisely, looking at Algorithm 1, we assert that memory accesses are caused only by the packing and the macro-kernel routines. Indeed, the *for-loop* associated variables are stored in registers and thus do not generate memory accesses. We thus focus on the core of the routines, which are represented in Algorithm 1 by expression *E1*, which manages the packing of B_c, expression *E2*, which manages the packing of A_c, and expression *E3*, which manages the tile C_c. Stack accesses are performed at the beginning and at the end of the calls to the functions that implement these routines. We identified their exact number in the assembly code and count them separately. Our objective is to reason about the data cache behavior and to be able to obtain formulae that 1) for the packing routines, guarantee that the number of cache misses is the theoretical minimum and 2) for the macro-kernel routine, only slightly overestimate the actual number of cache misses.

We summarize the notations in Table II. Subsequently, we start by presenting general results on matrix placement in caches and then focus on each of the aforementioned expressions.

TABLE II
NOTATION USED ON THE STUDY OF GEMM IMPLEMENTATION.

Notation	Description
m	Height of matrices A and C
n	Width of matrices B and C
k	Width of matrix A and height of matrix B
m_c	Height of a block A_c ($\tilde{A_c}$) and of a block C_c
n_c	Width of a block B_c ($\tilde{B_c}$) and of a block C_c
k_c	Width of a block A_c ($\tilde{A_c}$) and height of a block B_c ($\tilde{B_c}$)
m_r	Height of a micro-panel A_r
n_r	Width of a micro-panel B_r
$size_{Li}$	Size of L_i cache
w_{Li}	Associativity degree of L_i cache
s_{Li}	Number of sets of L_i cache
c_{Li}	Cache line length of L_i cache
s_{data}	Size of data
$X_{Li} = \frac{c_{Li}}{s_{data}}$	Number of elements per cache line of L_i cache

B. Mapping a matrix to a data cache

We consider a data cache with s_{Li} sets. We number the cache blocks in memory starting with index zero.

Theorem 1. *Let $b, n \in \mathbb{N}$. If n is odd, then the cache blocks of indices $b, b+n, b+2\cdot n, \dots, b+(s_{Li}-1)\cdot n$ are all mapped to different cache sets.*

Proof. Let us consider the following function that associates cache block indices to their corresponding set in the cache:

$$\begin{array}{rccc} f: & \{k\cdot n \mid k \in [\![0, s_{Li}-1]\!]\} & \to & [\![0, s_{Li}-1]\!] \\ & x & \mapsto & x \bmod s_{Li} \end{array}$$

For each cache block to be mapped to a different cache set, function f must be bijective. If we represent $[\![0, s_{Li}-1]\!]$ as the cyclic group $\mathbb{Z}/s_{Li}\mathbb{Z}$ equipped with the classical $+$ operator, then f bijective is equivalent to saying that n is a generator of $(\mathbb{Z}/s_{Li}\mathbb{Z}, +)$. Now, the generators of $(\mathbb{Z}/s_{Li}\mathbb{Z}, +)$ are the integers that are coprime with s_{Li}. As a consequence, f is bijective *iff* n is coprime with s_{Li}. Since the number of cache sets is usually implemented as a power of two, it is sufficient that n be odd to ensure the property. Finally, the property holds for any set that is isomorphic to the set $\{k\cdot n \mid k \in [\![0, s_{Li}-1]\!]\}$, in particular if we apply a translation by a constant integer b. □

Let X_{Li} be the number of elements a cache line can store. A data cache has s_{Li} sets, each with w_{Li} ways. Thus, using only one way per cache set it is possible to store at most $s_{Li} \cdot X_{Li}$ elements.

We consider a matrix M of shape $u \times v$, stored in row-major order, aligned in memory to the start of a cache block, such that v is a multiple of X_{Li}. Then the number of cache blocks occupied by a row of matrix M is $N_b = \frac{v}{X_{Li}}$. We now consider a slice of M composed of X_{Li} adjacent columns on s_{Li} consecutive rows, also aligned in memory to a cache block boundary. We study how to store such a slice using exactly one way of each data cache set.

Corollary 1. *If N_b is an odd integer, any slice of shape $s_{Li} \times X_{Li}$ of matrix M, whose first row is aligned to the start of a cache block, has each of its s_{Li} cache blocks mapped to a different cache set.*

Proof. First, we suppose that the first row of the first slice of shape $s_{Li} \times X_{Li}$ of matrix M is mapped to the cache block of index $c \in \mathbb{N}$. Since v is a multiple of X_{Li}, each row of M (and thus of the slice of shape $s_{Li} \times X_{Li}$) starts at the beginning of a cache block. In particular, the second row of the first slice of shape $s_{Li} \times X_{Li}$ of M is mapped to the cache block of index $c + N_b$. The third row is mapped to the cache block of index $c + 2 \cdot N_b$. Finally, the k-th row is mapped to block $c + (k-1) \cdot N_b$.

From Theorem 1 we have that if N_b is an odd integer, the s_{Li} consecutive rows of the first slice of shape $s_{Li} \times X_{Li}$ of matrix M are mapped to different cache sets. In fact, the property holds for any slice of shape $s_{Li} \times X_{Li}$ aligned in memory to the start of a cache block, and thus separated from the first slice of shape $s_{Li} \times X_{Li}$ by an integer offset multiple of X_{Li}. □

We introduce a running numerical example to facilitate the discussion of the following sections on the time-predictability of the GEMM algorithm with respect to the data cache.

Example 1. *We consider square matrices A, B and C wherein $m = n = k = 528$. We analyze an L1D cache like the one of our target, with the following characteristics: an LRU*

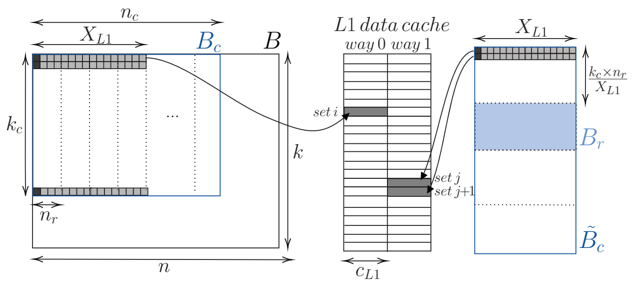


Fig. 3. Mapping rows of matrix B (B_c) and $\tilde{B}_c$ to cache blocks. The rows of a block B_c read during packing are not mapped into contiguous cache sets. In contrast, the different micro-panels B_r of $\tilde{B}_c$ are stored in sequence.

replacement policy, associativity degree of 2, size of 32KB, a cache line size (c_{L1}) of 64B, and with a number of sets (s_{L1}) of 256. Moreover, we consider single-precision floating-point data, where s_{data} is 4B. Thus, the number of elements per cache line in this example is $X_{L1} = \frac{c_{L1}}{s_{data}} = 16$.

C. L1D cache behavior in packing B_c

As explained in Section II, the packing algorithm reads elements from B_c and writes them in a copy block $\tilde{B}_c$. Both operations are done alternately, i.e. a reading instruction is followed by a writing instruction. Thus, to optimally use the data cache, the objective is to share it between the two operations. We do not wish that writing operations evict cache blocks loaded for the reading and that still may be needed. The contrary is also true: we do not want cache blocks loaded for the writing to be replaced before all their elements are stored. Practically, to prevent this kind of conflict, we dedicate half of the cache for lines used to read from B_c and half of the cache for lines to write to $\tilde{B}_c$. As our processor comes with a 2-way set associative L1D cache, for each cache set, one cache way is dedicated to B_c and the other to $\tilde{B}_c$. Note that the idea can be generalized to any even (> 1) or odd associativity degree cache. Figure 3 illustrates this philosophy.

The process of writing $\tilde{B}_c$ is done linearly to contiguous addresses. The corresponding cache blocks are thus allocated to cache sets with contiguous indices. Hence, to adhere to efficiently use only half of the cache for the writing, we must have $k_c = s_{L1}$. In contrast, the process of reading from matrix B_c is done by groups of n_r contiguous elements before treating the next matrix row, which is n elements away. In total, k_c matrix rows are accessed before returning to the first row of B_c, which has already been charged in the cache. Thus, for the reading, we find $k_c \leqslant s_{L1}$. In order to avoid conflicts and under-utilization of the cache sets, we wish to guarantee that each cache block with elements of B_c needed at a time is loaded to a different cache set. We thus choose to set $k_c = s_{L1}$.

However, since the elements read from B_c to form one micro-panel are not contiguous in memory, the question of the mapping between matrix rows and cache sets arises. To guarantee that these blocks are mapped to distinct cache sets, it is sufficient (Corollary 1) that the number of cache blocks composing a row of B be an odd integer. This can be ensured by padding each row of B by 1 cache block if necessary.

For Example 1, $\frac{n}{X_{L1}} = 33$, thus no padding is required, and k_c is chosen so that $k_c = s_{L1} = 256$.

C.1. Number of memory accesses

A block B_c of shape $k_c \times n_c$ is partitioned in $\left\lceil \frac{n_c}{n_r} \right\rceil$ micro-panels. We define:

- $n_p = \left\lfloor \frac{n_c}{n_r} \right\rfloor$: number of complete micro-panels in B_c (N.1)
- $n_f = n_c \bmod n_r$: width of the potential incomplete last micro-panel (N.2)

The n_p full micro-panels of shape $k_c \times n_r$ are read and then stored in the order they are accessed in the micro-kernel function, i.e. in the row-major order. For the incomplete micro-panel, n_f elements are read and then padded with zeros during writing, to attain a width of n_r and completely use the vector register's size.

To pack full micro-panels into $\tilde{B}_c$, there are $2 \cdot n_p \cdot k_c \cdot n_r$ memory accesses (same number of accesses for read and for write). To pack the last incomplete micro-panel, there are $2 \cdot n_f \cdot k_c + (n_r - n_f) \cdot k_c$ memory accesses. The total number of memory accesses involved in packing B_c is $2 \cdot n_p \cdot k_c \cdot n_r + 2 \cdot n_f \cdot k_c + (n_r - n_f) \cdot k_c$.

For Example 1, $n_r = 4$ and $n_c = 528$, i.e. the minimum between n and the n_c tailored for the target (see Section V-A). It follows that $n_p = 132$ and $n_f = 0$. Matrix B is then partitioned into 3 blocks B_c of shape $k_c \times n_c$: the first two are of shape 256×528 and the third has shape 16×528. Thus, the total number of memory accesses for packing the B_c blocks into $\tilde{B}_c$ blocks is $557,568$.

C.2. Upper bound on the number of cache misses

In order to argue on the number of cache misses in the L1D, we make the following assumptions:

- B and $\tilde{B}_c$ are aligned to a cache block boundary, (A.1)
- $\frac{n}{X_{L1}}$ is an odd integer, (A.2)
- n_r is a divisor of X_{L1}, (A.3)
- $k_c = s_{L1}$ (A.4)

In addition, we present the following notation relevant to the discussions of this section:

- $\left\lceil \frac{n_c}{X_{L1}} \right\rceil$: number of slices of shape $k_c \times X_{L1}$ in B_c (N.3)

To create a copy block $\tilde{B}_c$, a B_c block is read by groups of $k_c \times n_r$ elements, i.e. the micro-panels. To read the first micro-panel (leftmost), there are:

$$k_c \text{ misses} + k_c \cdot (n_r - 1) \text{ hits.}$$

Remember we chose k_c so that performing these first reads loads exactly k_c cache blocks in the L1D cache, each in a different set (A.4). Since n_r is a divisor of X_{L1} (A.3), the reads for the next $\frac{X_{L1}}{n_r} - 1$ micro-panels target the same cache blocks and thus do not generate additional misses.

Figure 3 illustrates this reasoning for a slice of shape $k_c \times X_{L1}$ of matrix B_c, where the darker gray color represents the cache misses at the beginning of each matrix row and the clearer gray represents the subsequent cache hits. We observe that there exist $\left\lceil \frac{n_c}{X_{L1}} \right\rceil$ slices of shape $k_c \times X_{L1}$ in matrix B_c (Notation N.3).

Thus, for reading the whole B_c block of width n_c, the algorithm performs:

$$k_c \cdot \left\lceil \frac{n_c}{X_{L1}} \right\rceil \text{ misses}$$

Similarly, for the writing of matrix $\tilde{B_c}$ we reason in terms of slices of shape $k_c \times X_{L1}$, aligned to the start of a cache block. We find that, at the beginning of each row of such a slice there is one cache miss, followed by cache hits to write the remaining $(X_{L1} - 1)$ elements present in the same cache block. Since $\tilde{B_c}$ is also composed of $\left\lceil \frac{n_c}{X_{L1}} \right\rceil$ slices of shape $k_c \times X_{L1}$ (Notation N.3), we find in total:

$$k_c \cdot \left\lceil \frac{n_c}{X_{L1}} \right\rceil \text{ misses as well.}$$

For Example 1, the first slice of shape $k_c \times X_{L1}$ of B_c contains the elements:
$B[0]...B[15]; B[528]...B[543]; ...; B[134640]...B[134655]$.
We then look at the first micro-panel, which is composed of:
$B[0]...B[3]; B[528]...B[531]; ...; B[134640]...B[134643]$.
When $B[0]$ is read, the cache loads the block composed of $B[0]...B[15]$. Without loss of generality, let us assume this block is mapped to set $\#10$. When $B[0]$ is written in $\tilde{B_c}$, the cache loads the corresponding block. Again, without loss of generality[2], let us assume this block is mapped to set $\#3$. The reading and writing of the next three elements of the first micro-panel ($B[1]...B[3]$) cause no extra cache misses, as the corresponding block is already in the cache. The next element to be read is $B[528]$. It belongs to another cache block that is loaded in set $\#(10 + 33) \bmod 256 = 43$, because $\left\lceil \frac{n}{X_{L1}} \right\rceil = 33$. Its writing however is done in the cache block present in set $\#3$. When element $B[1584]$, i.e. the first of the fourth row of the first micro-panel, is written in $\tilde{B_c}$, a new cache block is loaded in set $\#4$. The same reasoning applies until the 256th matrix row. When the copies of the second micro-panel start, with element $B[4]$, its reading causes no extra misses, while its writing loads a new cache block in set $\#(3 + 256/4) \bmod 256 = 67$.

In the end, since $\left\lceil \frac{n_c}{X_{L1}} \right\rceil = 33$, packing the three B_c blocks results in $34,848$ L1D cache misses.

D. L1D cache behavior in packing A_c

The process of packing A_c to create the copy block $\tilde{A_c}$ is similar to the one of packing B_c: reading and writing happen alternately as well. We can thus also partition the cache between the two operations.

The difference is that now, to construct the micro-panels of shape $m_r \times k_c$, we read one element from m_r consecutive rows of A_c before returning to the first of these m_r rows to read another element. This process is repeated k_c times (see Figure 2). It follows that, for reading from A_c, only m_r cache blocks are needed at a time. As explained, m_r represents a quantity to be stored in vector registers, which is significantly smaller than s_{L1}. To ensure that the corresponding m_r cache blocks are mapped to distinct cache sets, once again following Corollary 1, it is sufficient that each row of A_c be composed of an odd number of cache blocks. As with B, this can be enforced by padding each row of A with one cache block if necessary.

Finally, the $m_r \cdot k_c$ elements of a micro-panel are written to contiguous memory addresses in $\tilde{A_c}$. Thus, once a cache block gets loaded, it is filled by stores before the next one is loaded, at which point the block is no longer needed.

D.1. Number of memory accesses

We define m_p (resp. m_f) in the same way as n_p (resp. n_f) for block A_c. In order to pack full micro-panels present in A_c, the algorithm does $2 \cdot m_p \cdot k_c \cdot m_r$ memory accesses. To pack a potential incomplete micro-panel of A_c, $2 \cdot m_f \cdot k_c + (m_r - m_f) \cdot k_c$ memory accesses are performed. In total, to create a copy block $\tilde{A_c}$, there are $2 \cdot m_p \cdot k_c \cdot m_r + 2 \cdot m_f \cdot k_c + (m_r - m_f) \cdot k_c$ memory accesses.

For Example 1, we have $m_c = 528$, $m_r = 4$, $m_p = 132$ and $m_f = 0$. A is then also partitioned into 3 A_c blocks. Thus, the total number of memory accesses during the packing of blocks A_c is $557,568$.

D.2. Upper bound on the number of cache misses

To reason about the packing of A_c, we add the following assumptions:

- A and $\tilde{A_c}$ are aligned to a cache block boundary, (A.5)
- $\dfrac{k}{X_{L1}}$ is an odd integer, (A.6)
- k_c is a multiple of X_{L1}, (A.7)

We also introduce the following relevant notations:

- $\left\lceil \dfrac{k_c}{X_{L1}} \right\rceil$: number of cache blocks used by k_c elements (N.4)
- $\left\lceil \dfrac{m_r \cdot k_c}{X_{L1}} \right\rceil$: number of cache blocks occupied by A_r (N.5)
- $\left\lceil \dfrac{m_c}{m_r} \right\rceil$: number of micro-panels A_r in A_c (and $\tilde{A_c}$) (N.6)

To read one micro-panel of shape $m_r \times k_c$ of A_c, knowing Notation N.4, we find:

$$m_r \cdot \left\lceil \frac{k_c}{X_{L1}} \right\rceil \text{ misses}$$

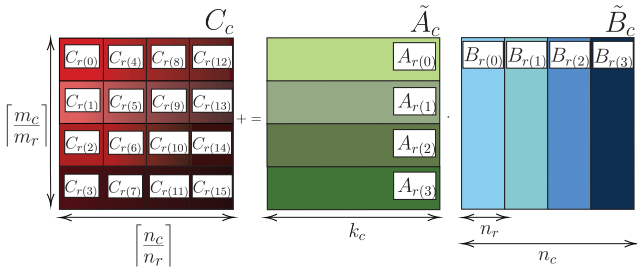
Thus, to read the whole A_c block, there are:

$$\left\lceil \frac{m_c}{m_r} \right\rceil \cdot m_r \cdot \left\lceil \frac{k_c}{X_{L1}} \right\rceil \text{ misses.}$$

To write one micro-panel in $\tilde{A_c}$, $\left\lceil \frac{m_r \cdot k_c}{X_{L1}} \right\rceil$ cache blocks are needed (Notation N.5). We thus have, for one micro-panel, a bound of:

$$\left\lceil \frac{m_r \cdot k_c}{X_{L1}} \right\rceil \text{ misses.}$$

[2]In particular, the block could also be mapped to set $\#10$.



C_c
$\tilde{A}_c$
$\tilde{B}_c$
$C_{r(0)}$ $C_{r(4)}$ $C_{r(8)}$ $C_{r(12)}$
$C_{r(1)}$ $C_{r(5)}$ $C_{r(9)}$ $C_{r(13)}$
$C_{r(2)}$ $C_{r(6)}$ $C_{r(10)}$ $C_{r(14)}$
$C_{r(3)}$ $C_{r(7)}$ $C_{r(11)}$ $C_{r(15)}$
$+=$
$A_{r(0)}$
$A_{r(1)}$
$A_{r(2)}$
$A_{r(3)}$
$B_{r(0)}$ $B_{r(1)}$ $B_{r(2)}$ $B_{r(3)}$
$\left\lceil\frac{m_c}{m_r}\right\rceil$
$\left\lceil\frac{n_c}{n_r}\right\rceil$
k_c
n_r
n_c

Fig. 4. Macro-kernel algorithm for $\frac{m_c}{m_r} = \frac{n_c}{n_r} = 4$. To update the micro-tile $C_{r(0)}$, micro-panels $A_{r(0)}$ and $B_{r(0)}$ are used. The next treated micro-tile is $C_{r(1)}$, using micro-panels $A_{r(1)}$ and again $B_{r(0)}$. Micro-tile $C_{r(9)}$, for example, is updated by micro-panels $A_{r(1)}$ and $B_{r(2)}$.

Consequently, to write the whole $\tilde{A}_c$ block (Notation N.6), we find:

$$\left\lceil\frac{m_c}{m_r}\right\rceil \cdot \left\lceil\frac{m_r \cdot k_c}{X_{L1}}\right\rceil \text{ misses.}$$

In Ex. 1, the first micro-panel of A_c is composed of:
$A[0], A[528], A[1056], A[1584]$;
$A[1], A[529], A[1057], A[1585]; ...;$
$A[255], A[783], A[1311], A[1839]$

While reading element $A[0]$ from A_c, a new cache block is loaded to, let us assume, set $\#5$. Writing $A[0]$ to $\tilde{A}_c$ loads a cache block to, again hypothetically, set $\#1$. Since $\frac{k}{X_{L1}} = 33$, for the reading of element $A[528]$, a cache block is loaded to set $\#(5 + 33) \bmod 256 = 38$, and this element is written to the cache block present in set $\#1$. Reading element $A[4]$ causes no extra cache misses as this block was already loaded, but its writing requires a new cache block, which is mapped to set $\#2$. Copying element $A[16]$, for example, requires a new cache block both for reading and for writing, mapped respectively to sets $\#6$ and $\#(1 + 16/4) \bmod 256 = 5$. Hence, packing the different A_c blocks into $\tilde{A}_c$ blocks results in 34,848 L1D cache misses.

E. L1D cache behavior in macro-kernel

In the macro-kernel function, the copy blocks $\tilde{A}_c$ and $\tilde{B}_c$ are partitioned in micro-panels used to update a different micro-tile C_r in each call to the micro-kernel function (cf. Algorithm 1). Between successive calls to the micro-kernel, a new micro-tile C_r, as well as a new micro-panel A_r, are brought into registers. The algorithm was designed to favor each micro-panel B_r to stay in the L1D cache for $\left\lceil\frac{m_c}{m_r}\right\rceil$ executions of the micro-kernel function before being replaced. However, not knowing exactly where the elements of these matrices are mapped in memory, it is not possible to guarantee that the expected behavior occurs. Figure 4 illustrates the macro-kernel algorithm.

Example 2. *The sequence of operations performed in the macro-kernel applied to the example of Figure 4 is:*

$$R\text{-}C_{r(0)}; (R\text{-}A_{r(0)} \| R\text{-}B_{r(0)}); W\text{-}C_{r(0)};$$
$$R\text{-}C_{r(1)}; (R\text{-}A_{r(1)} \| R\text{-}B_{r(0)}); W\text{-}C_{r(1)};$$
$$R\text{-}C_{r(2)}; (R\text{-}A_{r(2)} \| R\text{-}B_{r(0)}); W\text{-}C_{r(2)}; ...$$

Where R stands for reading and W for writing. The $\|$ symbol highlights that A_r and B_r are read alternately, by groups of m_r (respectively n_r) elements, as shown in Algorithm 2.

E.1. Number of memory accesses

Hereby we outline the following clarifying notation:

- $\left\lceil\frac{m_c}{m_r}\right\rceil \cdot \left\lceil\frac{n_c}{n_r}\right\rceil$: number of micro-tiles C_r in a block C_c (N.7)

The micro-kernel routine is then executed $\left\lceil\frac{m_c}{m_r}\right\rceil \cdot \left\lceil\frac{n_c}{n_r}\right\rceil$ times. In the micro-kernel, a micro-panel A_r is accessed k_c times to load m_r consecutive elements into registers each time. Since in our case m_r has the same size as one vector register (4 FP32 elements), with one memory access it is possible to fill the register. Thus, there are k_c memory accesses to read A_r in the micro-kernel. The same is also true for a micro-panel B_r, with k_c memory accesses to read it in the micro-kernel. The micro-tile C_r, in contrast, is accessed $m_r \cdot n_r$ times for reading because the elements required in the registers for C_r are not contiguous in memory (as they come from multiple rows, see Figure 4) so reads cannot be grouped. The same applies for the $m_r \cdot n_r$ writes. Thus in total, given N.7, there are:

$$\left\lceil\frac{m_c}{m_r}\right\rceil \cdot \left\lceil\frac{n_c}{n_r}\right\rceil \cdot (2 \cdot k_c + 2 \cdot m_r \cdot n_r)$$

memory accesses in the macro-kernel function.

For Example 1, $\left\lceil\frac{m_c}{m_r}\right\rceil \cdot \left\lceil\frac{n_c}{n_r}\right\rceil = 17,424$. The macro-kernel is also called 3 times, twice with $k_c = 256$ and once with $k_c = 16$. Hence, the total of memory accesses in the macro-kernel function is 20,072,448.

To reason about the L1D misses in the macro-kernel, we add the following assumption:

- C is aligned to a cache block boundary. (A.8)

Let us recall some properties of the submatrices involved in the macro-kernel:

Property 1. *$\tilde{A}_c$ and $\tilde{B}_c$ are linear arrays in memory. Thus, for all i, micro-panels $A_{r(i)}$ and $A_{r(i+1)}$ occupy contiguous memory addresses and are mapped to distinct - and adjacent - cache sets. The same holds for $B_{r(j)}$ and $B_{r(j+1)}$ for all j.*

Property 2. *Within a micro-panel A_r (or B_r), the elements are placed at contiguous memory addresses. It follows that the cache blocks corresponding to a micro-panel are mapped to distinct cache sets.*

Property 3. *The m_r rows of each of $\left\lceil\frac{m_c}{m_r}\right\rceil$ successive micro-tiles C_r are mapped to distinct cache sets. This property derives from Corollary 1 and potential padding of C to achieve Assumption A.2.*

Property 4. *Because in our study $m_r = n_r$, micro-panels A_r and B_r occupy the same number of cache blocks (see Notation N.5). Thus, a micro-panel A_r cannot share cache* *sets with more than two micro-panels B_r, and conversely. See Figure 5 for illustration.*

E.2. Upper bound on the number of cache misses - general case

In the following steps, we study the cache behavior in the macro-kernel algorithm for the expected lifetime of a micro-panel B_r, i.e. for $\left\lceil \frac{m_c}{m_r} \right\rceil$ successive calls to the micro-kernel.

a) Upper bound on cache misses due to reading Cr:

A micro-tile C_r occupies m_r cache blocks. For any given micro-panel B_r, $\left\lceil \frac{m_c}{m_r} \right\rceil$ micro-tiles are updated, leading to:

$$\left\lceil \frac{m_c}{m_r} \right\rceil \cdot m_r \text{ misses.} \qquad (1)$$

Let us consider the first micro-tile, $C_{r(0)}$. For Example 1, it is composed of the elements: $C[0]...C[3]; C[528]...C[531]; ...; C[1584]...C[1587]$. Assuming that element $C[0]$ is mapped to cache set #0, element $C[528]$ (resp. $C[1056]$ and $C[1584]$) is mapped to cache set $\#(0+33) \bmod 256 = 33$ (resp. #66 and #99). Reading $C_{r(0)}$, incurs 4 cache misses. Hence, to read the $\left\lceil \frac{m_c}{m_r} \right\rceil$ micro-tiles, there are $132 \times 4 = 528$ cache misses.

b) Upper bound on cache misses due to reading Ar:

A micro-panel A_r occupies $\left\lceil \frac{m_r \cdot k_c}{X_{L1}} \right\rceil$ cache blocks. For any given micro-panel B_r, $\left\lceil \frac{m_c}{m_r} \right\rceil$ micro-panels A_r are read, leading to:

$$\left\lceil \frac{m_c}{m_r} \right\rceil \cdot \left\lceil \frac{m_r \cdot k_c}{X_{L1}} \right\rceil \text{ misses.} \qquad (2)$$

For Example 1, let us consider the first micro-panel, $A_{r(0)}$. It contains 1,024 elements. Its first element corresponds to $\tilde{A}_c[0]$, which we assume is mapped to cache set #1. Since the elements within a micro-panel are placed in contiguous addresses in memory (Property 2), $\tilde{A}_c[16]$ is mapped to cache set #2, while $\tilde{A}_c[1023]$ is loaded to cache set #64. Reading micro-panel $A_{r(0)}$ causes 64 cache misses, and reading the $\left\lceil \frac{m_c}{m_r} \right\rceil$ needed micro-panels generates 8,448 misses.

c) Upper bound on cache misses due to reading Br:

i) Initial reading of B_r

A micro-panel B_r occupies $\left\lceil \frac{k_c \cdot n_r}{X_{L1}} \right\rceil$ cache blocks. As we focus on the lifetime of a micro-panel B_r, for its initial loading in the cache we find:

$$\left\lceil \frac{k_c \cdot n_r}{X_{L1}} \right\rceil \text{ misses.} \qquad (3)$$

For Example 1, let us consider the first micro-panel, $B_{r(0)}$. Assuming its first element $\tilde{B}_c[0]$ is mapped to cache set #3, element $\tilde{B}_c[1023]$ is then mapped to cache set #66. Thus, for reading micro-panel $B_{r(0)}$ there are 64 cache misses.

ii) B_r becomes the least recently used because of A_r and is evicted by C_r

Let us consider the sequence of operations presented in Example 2 from $R\text{-}C_{r(0)}$ to $R\text{-}C_{r(1)}$.

Following Properties 1 and 4, the worst case of conflicts between cache blocks of micro-panels A_r and B_r happens when, for example, the mapping of micro-panel $A_{r(0)}$ starts exactly one cache block above the micro-panel $B_{r(0)}$, as depicted in Figure 5. In this case, after the sequence $(R\text{-}A_{r(0)} \| R\text{-}B_{r(0)})$, all the cache blocks of $B_{r(0)}$ become the least recently used of their respective sets, except for the last one. Then, if $C_{r(0)}$ and $C_{r(1)}$ are both mapped to cache sets where $B_{r(0)}$ is old, the actions of $W\text{-}C_{r(0)}$ and $R\text{-}C_{r(1)}$ cause the eviction of blocks of $B_{r(0)}$. From Property 3 we know that $C_{r(0)}$ and $C_{r(1)}$ are not mapped to the same cache sets, thus each of the micro-tiles can contribute to a maximum of m_r misses in $B_{r(0)}$. Following Property 4 only $A_{r(0)}$ and $A_{r(1)}$ conflict with blocks of $B_{r(0)}$. Thus only $C_{r(0)}$ and $C_{r(1)}$ can evict blocks of $B_{r(0)}$, as they are loaded and stored in the same time frame as $A_{r(0)}$ and $A_{r(1)}$.

According to Property 4, this scenario can happen at most once each time the $\left\lceil \frac{m_c}{m_r} \right\rceil$ micro-panels of $\tilde{A}_c$ fill one cache way of all cache sets. In the worst case we thus have:

$$\left\lceil \frac{\left\lceil \frac{m_c}{m_r} \right\rceil \cdot \left\lceil \frac{m_r \cdot k_c}{X_{L1}} \right\rceil}{s_{L1}} \right\rceil \cdot 2 \cdot m_r \text{ misses.} \qquad (4)$$

For Example 1, after the $(R\text{-}A_{r(0)} \| R\text{-}B_{r(0)})$ sequence , elements $\tilde{B}_c[492]...\tilde{B}_c[495]$ and $\tilde{A}_c[528]...\tilde{A}_c[531]$ are present in set #33, with the elements of $B_{r(0)}$ being the least recently used (due to the fact that $\tilde{B}_c$ starts in set #3 and $\tilde{A}_c$ in set #0). Thus, when $C[528]$, which belongs to a block that is also mapped to set #33, is written, the block containing $\tilde{B}_c[492]...\tilde{B}_c[495]$ is evicted.

iii) B_r becomes the least recently used because of C_r and is evicted by A_r

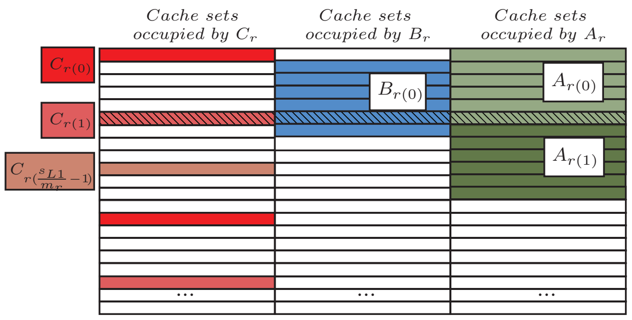
We now consider a sequence of operations comprising three micro-kernel executions, such as the one starting with $R\text{-}C_{r(0)}$ and ending with $W\text{-}C_{r(2)}$ in Example 2.

We suppose that cache blocks of $B_{r(0)}$ are mapped to the same cache sets as cache blocks of $C_{r(1)}$ and of $A_{r(1)}$. In this case, when the cache blocks of $B_{r(0)}$ become the least recently used after $R\text{-}C_{r(1)}$, they are evicted by the operations of $R\text{-}A_{r(1)}$. Following the algorithm, when they become old again after $W\text{-}C_{r(1)}$, they cannot be evicted by $R\text{-}A_{r(2)}$, because by construction the cache sets that are occupied by the blocks of $A_{r(1)}$ and $A_{r(2)}$ are distinct (Property 1).

Thus, reading and writing to a given micro-tile C_r cannot contribute to evicting B_r twice. Again, we must multiply the cache blocks occupied by B_r by the number of times one cache way of each cache set is filled with the $\left\lceil \frac{m_c}{m_r} \right\rceil$ micro-tiles C_r. We thus have:

$$\left\lceil \frac{\left\lceil \frac{m_c}{m_r} \right\rceil \cdot m_r}{s_{L1}} \right\rceil \cdot \left\lceil \frac{k_c \cdot n_r}{X_{L1}} \right\rceil \text{ misses.} \qquad (5)$$

For Example 1, after the $(R\text{-}A_{r(0)} \| R\text{-}B_{r(0)})$ sequence, $B_{r(0)}$ has the most recent blocks in sets #65 and #66. Then writing $C[1056]$, of $C_{r(0)}$, in set #66 makes $B_{r(0)}$ old in this set. In the next sequence, $A_{r(1)}$ is read. When element $\tilde{A}_c[1040]$ is read and its block loaded in set #66, the reads of $B_{r(0)}$ have



Cache sets occupied by C_r
Cache sets occupied by B_r
Cache sets occupied by A_r
$C_{r(0)}$
$C_{r(1)}$
$C_{r(\frac{s_{L1}}{m_r}-1)}$
$B_{r(0)}$
$A_{r(0)}$
$A_{r(1)}$
...

Fig. 5. Potential mapping of micro-tiles C_r, micro-panels B_r and A_r in cache sets. The hatched lines represent a cache set where a conflict exists.

not yet reached the block mapped to set #66, so the $B_{r(0)}$ block that was loaded in this set gets evicted.

d) Upper bound on cache misses due to writing Cr:

We now look at the first four operations of Example 2. As the cache blocks within any given micro-panel do not conflict (Property 2), the problematic case is when cache blocks of $A_{r(0)}$ and of $B_{r(0)}$ are mapped to the same cache sets as cache blocks previously loaded for $C_{r(0)}$. In this case, R-$A_{r(0)}$ makes elements of $C_{r(0)}$ to be the least recently used, and R-$B_{r(0)}$ can evict them.

Now considering the $\left\lceil \frac{m_c}{m_r} \right\rceil$ consecutive calls to the micro-kernel, Figure 5 illustrates a potential mapping of successive micro-tiles C_r to cache sets. Each micro-tile occupies m_r (potentially scattered) cache blocks (Property 3) and $\frac{s_{L1}}{m_r}$ micro-tiles are needed to fill one way of each cache set (for space reasons, not all sets are present in the figure). The figure also depicts a possible mapping of a micro-panel B_r and of micro-panels A_r. Since our studied L1D cache only has two physical ways, conflicts can arise. However, we also observe that only the cache blocks of micro-tiles C_r that are mapped to the same sets as B_r can be evicted, i.e. a maximum of $\left\lceil \frac{k_c \cdot n_r}{X_{L1}} \right\rceil$ cache blocks.

For Example 1, it corresponds to writing element $C[528]$ of $C_{r(0)}$ to cache set #33, since after the $(R\text{-}A_{r(0)} \| R\text{-}B_{r(0)})$ sequence both $A_{r(0)}$ and $B_{r(0)}$ have a block in this set.

We must multiply it by the number of times that the $\left\lceil \frac{m_c}{m_r} \right\rceil$ micro-tiles C_r load a block in one cache way of each set once, which leads to an upper bound of:

$$\left\lceil \frac{\left\lceil \frac{m_c}{m_r} \right\rceil \cdot m_r}{s_{L1}} \right\rceil \cdot \left\lceil \frac{k_c \cdot n_r}{X_{L1}} \right\rceil \text{ misses.} \qquad (6)$$

Finally, to obtain the upper bound on the total number of cache misses in the macro-kernel one must sum up all the previous formulae and multiply the sum by $\left\lceil \frac{n_c}{n_r} \right\rceil$, i.e. the number of micro-panels B_r. We thus have:

$$\left\lceil \frac{n_c}{n_r} \right\rceil \cdot ((1) + (2) + (3) + (4) + (5) + (6)) \text{ misses.}$$

For Example 1, we find an upper bound of $2,703,897$ L1D cache misses in the macro-kernel executions.

VII. Evaluation of the analysis

The results of Section V-A were obtained using the parameters from *Low et al.* [11]. Using the formulae of this section, we select new blocking parameters (called *Ours* subsequently):

$$k_c = 256 \quad m_c = 1,792 \quad n_c = 4,096 \quad n_r = 4 \quad m_r = 4$$

In our experimental setup, we enforce Assumptions A.1, A.5 and A.8 using the *aligned* attribute of the C language, Assumptions A.2 and A.6 by padding the matrices rows by at most 1 cache block if necessary, and Assumptions A.4 and A.7 by choosing proper values for the parameters. Finally, our target hardware platform validates Assumption A.3.

To the values obtained using the formulae detailed in the previous sections, we add the number of accesses and cache misses caused by the stack management when calling the packing A_c, packing B_c, macro-kernel and performance monitoring functions. These extra accesses are invariant for each call of the functions, and are obtained by carefully reading the assembly code of the program.

Table III summarizes the measures obtained for the memory accesses and L1D refill for each part of the GEMM algorithm. We note that the measured memory accesses in *Ours* are exactly the same as expected, thus the values are only displayed once in the table. For the L1D refill evaluation, together with the measures for *Ours* and *Low et al.*, we present the theoretical values derived from the formulae of Section VI. For memory accesses and L1D refills, the percentages represent the difference w.r.t. measurements using our blocking parameters.

The measures of the number of memory accesses of *Low et al.* are slightly inferior to *Ours*. It is explained by the choice of parameter k_c: since it is smaller in our case (256 vs 512), the matrices are partitioned into more blocks. As a consequence, the functions for packing A_c, packing B_c and the macro-kernel are called more times, which adds a small overhead.

We also notice that, particularly for the packing B_c routine, carefully choosing the parameter k_c allows us to control and reduce the number of L1D refills by more than 60%, in the best case, compared to *Low et al.* Note that in matrix configuration *(iv)*, $\frac{n}{X_{L1}}$ is not an odd integer. The asterisk indicates that each row of matrix B was padded with zeros so it occupies an odd number of cache blocks. This modification only affects the stride between successive rows.

In matrix configuration *(iii)*, we padded matrix A because $\frac{k}{X_{L1}}$ is not an odd integer. However, we remark from the L1D refills when packing A_c in the column of *Low et al.* that for this particular matrix there are no extra misses, even though the rows of A occupy an even number of cache blocks. As explained, for packing A_c this condition is not as crucial as for the other functions as only m_r cache blocks are used at a time to read A_c, which reduces the occurrence of two of these blocks being mapped to the same set. Depending on the application, the potential padding of matrices A and B can be performed beforehand or using DMA copies. Otherwise, the padding routine must be included in the execution time. For the evaluation of cache misses in the macro-kernel, the first three

TABLE III
MEASURES FOR THE NUMBER OF MEMORY ACCESS AND L1D REFILL OF THE GEMM ROUTINE.

	Matrix configuration				Memory accesses			L1D refill				
	m	n	k		***Ours***	***Low et al.***		***Ours***	***Low et al.***		**Theoretical**	
(i)	272	272	272	Packing B	148 032	148 000	(-0.02%)	9 253	10 612	(12.81%)	9 254	(0.01%)
				Packing A	148 032	148 000	(-0.02%)	9 252	9 251	(-0.01%)	9 254	(0.02%)
				Macro-kernel	2 811 414	2 663 435	(-5.26%)	333 950	347 307	(3.85%)	384 886	(13.23%)
				Total	3 107 478	2 959 435	(-4.76%)	352 455	367 170	(4.01%)	403 394	(12.63%)
(ii)	528	528	528	Packing B	557 664	557 632	(-0.01%)	34 856	78 145	(55.40%)	34 857	(0.00%)
				Packing A	557 664	557 632	(-0.01%)	34 854	34 852	(-0.01%)	34 857	(0.01%)
				Macro-kernel	20 072 481	19 514 902	(-2.78%)	2 552 600	2 509 309	(-1.73%)	2 703 897	(5.60%)
				Total	21 187 809	20 630 166	(-2.63%)	2 622 310	2 622 306	(0.00%)	2 773 611	(5.46%)
(iii)	256	784	2,016*	Packing B	3 161 344	3 161 216	(0.00%)	197 591	341 213	(42.09%)	197 592	(0.00%)
				Packing A	1 032 448	1 032 320	(-0.01%)	64 528	64 520	(-0.01%)	64 536	(0.01%)
				Macro-kernel	53 788 760	52 183 084	(-2.99%)	6 894 927	6 766 186	(-1.90%)	7 217 528	(4.47%)
				Total	57 982 552	56 376 620	(-2.77%)	7 157 046	7 171 919	(0.21%)	7 479 656	(4.31%)
(iv)	192	736*	528	Packing B	777 312	777 248	(-0.01%)	48 584	121 948	(60.16%)	48 585	(0.00%)
				Packing A	202 848	202 784	(-0.03%)	12 677	12 675	(-0.02%)	12 681	(0.03%)
				Macro-kernel	10 174 497	9 891 862	(-2.78%)	1 248 905	1 258 791	(0.79%)	1 322 057	(5.53%)
* Padding applied.				Total	11 154 657	10 871 894	(-2.53%)	1 310 166	1 393 414	(5.97%)	1 383 323	(5.29%)

configurations use the general formulae of Section VI-E.2., and for the fourth configuration, the last macro-kernel corresponds to a particular case, presented in [28]. We observe that in most matrix configurations our formulae overestimate the number of misses in the macro-kernel by approximately 5%, which is reasonable given the complexity of the analysis. For matrix configuration *(i)*, the overestimation is more significant. In this case, the dimensions of the submatrices of the last macro-kernel are slightly bigger than in [28] but the remainder is not large enough to fill the cache. This leads to a larger overhead, as our formulae assume full usage of the cache.

Finally, Table IV shows the mean (and variability between minimum and maximum) execution times of the GEMM algorithm when using our parameters, as well as the difference with the execution times obtained with the parameters of Low et al. (from Table I). We observe that the cost of predictability does not exceed 4.22% in our experiments.

The L2 cache of our target has a random replacement policy, so our formulae cannot be extended for it. However, in our experiments, it was large enough not to interfere.

TABLE IV
MEASURED EXECUTION CYCLES, ON AN ARM CORTEX-A15 WITH -O0 FLAG. DIFFERENCE W.R.T. *Low et al.* IS GIVEN IN PARENTHESES.

Matrix config.				Execution time		
m	n	k		***Low et al.***	**Ours**	
272	272	272	mean	235 168 329	241 592 816	(2.66%)
			min	235 153 669	241 590 696	
			max	235 181 181	241 594 779	
			var.	1.16e−4%	1.69e−3%	
528	528	528	mean	1 566 589 440	1 588 903 889	(1.40%)
			min	1 566 565 684	1 588 886 193	
			max	1 566 603 968	1 588 925 463	
			var.	2.44e−3%	2.47e−3%	
256	784	2,016	mean	134 562 545	140 491 277	(4.22%)
			min	134 561 357	140 490 036	
			max	134 563 809	140 492 596	
			var.	1.82e−3%	1.82e−3%	
192	736	528	mean	778 924 693	793 768 157	(1.87%)
			min	778 923 061	793 765 272	
			max	778 926 162	793 769 629	
			var.	3.98e−4%	5.49e−4%	

VIII. GENERALIZABILITY OF THE ANALYSIS

The formulae presented in Section VI are tailored for the state-of-art GEMM algorithm of [7] and for an ARM v7-A core of the KEYSTONE II SoC. Our discussions do not focus on whether this algorithm is the most efficient, but reason on how to make it time-predictable. To analyze a different GEMM algorithm (e.g. different matrix partitioning so that A_c, instead of B_c, stays in the L1D cache [14]), the formulae must be adapted, following the same reasoning that is detailed in this paper. Theorem 1, regarding the maximization of data sets usage when performing strided matrix accesses, is general.

Regarding the hardware model, our main assumptions concern the associativity degree (>1) and the replacement policy (LRU) of the L1D cache, as well as the absence of prefetching mechanisms. Thus, if the assumptions hold, the formulae remain valid and can simply be tuned to a different target, as they are parameterized by key hardware characteristics (e.g. vector units latency, cache sizes). When considering a distinct instruction set architecture (ISA), such as a RISC-V, only the micro-kernel function (Algorithm 2) must be adapted, requiring minimal changes to the library.

IX. CONCLUSION

We propose a predictable and traceable yet efficient implementation of a blocked GEMM algorithm. Compared to a non-optimized implementation, we reduce the execution time by up to 99%, without compiler optimizations. Moreover, within each part of the algorithm, we find the correct number of memory accesses. The number of L1D cache misses is only overestimated by around 5% in all but one of the tested matrix configurations. Our results show that the cost of predictability is less than 5%. In future work, we will extend our formulae to predictable L2 (and potentially L3) caches. We will also refine the analysis for cache misses in the macro-kernel function when the submatrices do not consistently fill the cache. Lastly, we will automatize the generation of the proposed GEMM algorithm code for a given architecture and matrix configuration, and integrate it in larger applications, such as machine learning frameworks.

Acknowledgements

The authors would like to thank Simon Marmorat and Emmanuel Rio for their insights on the theorem proofs.

References

[1] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, p. 1–17, Mar. 1990. [Online]. Available: https://doi.org/10.1145/77626.79170

[2] *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Inc. Std., 2011.

[3] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, 2008.

[4] R. Pujol, J. Jorba, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella, and F. Cazorla, "Vector extensions in COTS processors to increase guaranteed performance in real-time systems," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 2, pp. 1–26, Jan. 2023. [Online]. Available: https://doi.org/10.1145/3561054

[5] J. Abella, C. Hernandez, E. Quiñones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega, "WCET analysis methods: Pitfalls and challenges on their trustworthiness," in *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2015, pp. 1–10.

[6] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi, "A survey on static cache analysis for real-time systems," *Leibniz Trans. Embed. Syst.*, vol. 3, no. 1, pp. 05:1–05:48, 2016. [Online]. Available: https://doi.org/10.4230/LITES-v003-i001-a005

[7] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software*, vol. 34, no. 3, pp. 1–25, May 2008. [Online]. Available: https://doi.org/10.1145/1356052.1356053

[8] Z. Xianyi, W. Qian, and W. Saar, "OpenBLAS: An optimized BLAS library," Mar 2011. [Online]. Available: https://www.openblas.net/

[9] F. G. Van Zee and R. A. van de Geijn, "BLIS: A Framework for Rapidly Instantiating BLAS Functionality," *ACM Transactions on Mathematical Software*, vol. 41, no. 3, pp. 14:1–14:33, 2015. [Online]. Available: https://doi.org/10.1145/2764454

[10] A. Butterfield and G. E. Ngondi, Eds., *A Dictionary of Computer Science*. Oxford University Press, Jan. 2016. [Online]. Available: https://doi.org/10.1093/acref/9780199688975.001.0001

[11] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, "Analytical modeling is enough for high-performance BLIS," *ACM Transactions on Mathematical Software*, vol. 43, no. 2, pp. 1–18, Aug. 2016. [Online]. Available: https://doi.org/10.1145/2925987

[12] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "LIBXSMM: Accelerating small matrix multiplications by runtime code generation," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov. 2016. [Online]. Available: https://doi.org/10.1109/sc.2016.83

[13] G. Frison, D. Kouzoupis, T. Sartor, A. Zanelli, and M. Diehl, "BLASFEO: Basic Linear Algebra Subroutines for Embedded Optimization," *ACM Trans. Math. Softw.*, vol. 44, no. 4, jul 2018. [Online]. Available: https://doi.org/10.1145/3210754

[14] W. Yang, J. Fang, D. Dong, X. Su, and Z. Wang, "LIBSHALOM: optimizing small and irregular-shaped matrix multiplications on ARMv8 multi-cores," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Nov. 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476217

[15] J. Zhang, F. Franchetti, and T. M. Low, "High performance zero-memory overhead direct convolutions," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 5776–5785. [Online]. Available: https://proceedings.mlr.press/v80/zhang18d.html

[16] S. Barrachina, A. Castelló, M. F. Dolz, T. M. Low, H. Martínez, E. S. Quintana-Ortí, U. Sridhar, and A. E. Tomás, "Reformulating the direct convolution for high-performance deep learning inference on ARM processors," *Journal of Systems Architecture*, vol. 135, p. 102806, Feb. 2023. [Online]. Available: https://doi.org/10.1016/j.sysarc.2022.102806

[17] S. Catalán, F. D. Igual, R. Mayo, R. Rodríguez-Sánchez, and E. S. Quintana-Ortí, "Architecture-aware configuration and scheduling of matrix multiplication on asymmetric multicore processors," *Cluster Computing*, vol. 19, no. 3, pp. 1037–1051, Aug. 2016. [Online]. Available: https://doi.org/10.1007/s10586-016-0611-8

[18] R. Whaley and J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the IEEE/ACM SC98 Conference*. IEEE, 1998. [Online]. Available: https://doi.org/10.1109/sc.1998.10004

[19] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 3393–3404.

[20] 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," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'20. USA: USENIX Association, 2020.

[21] N. Tollenaere, G. Iooss, S. Pouget, H. Brunie, C. Guillon, A. Cohen, P. Sadayappan, and F. Rastello, "Autotuning convolutions is easier than you think," *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 2, pp. 1–24, Mar. 2023. [Online]. Available: https://doi.org/10.1145/3570641

[22] *66AK2Hxx Multicore DSP+ARM KeyStone II System-on-Chip (SoC)*, Texas Instruments, 2017.

[23] Arm, "Neon architecture," 2013. [Online]. Available: https://developer.arm.com/Architectures/Neon

[24] ——, *Cortex-A15 MPCore Technical Reference Manual*, revision: r3p0 ed., 2012.

[25] D. Kästner, U. Wünsche, J. Barrho, M. Schlickling, B. Schommer, M. Schmidt, C. Ferdinand, X. Leroy, and S. Blazy, "CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler," in *ERTS 2018: Embedded Real Time Software and Systems*. SEE, Jan. 2018.

[26] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Comput.*, vol. 1, no. 4, pp. 541–551, 1989.

[27] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations (ICLR)*, Y. Bengio and Y. LeCun, Eds., 2015.

[28] I. De Albuquerque Silva, T. Carle, A. Gauffriau, V. Jegu, and C. Pagetti, "Appendix for paper: A predictable SIMD library for GEMM routines," 2024. [Online]. Available: https://www.irit.fr/~Thomas.Carle/appendix-for-paper-a-predictable-simd-library-for-gemm-routines/