autoGEMM: Pushing the Limits of Irregular Matrix Multiplication on Arm Architectures

Du Wu[2,3,*], Jintao Meng[1,*], Wenxi Zhu[4], Minwen Deng[4], Xiao Wang[5], Tao Luo[6],
Mohamed Wahib[3,†], Yanjie Wei[1,†]
[1] *Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, China*
[2] *Tokyo Institute of Technology, Japan*
[3] *RIKEN Center for Computational Science, Japan*
[4] *Tencent AI Lab, China*
[5] *Oak Ridge National Laboratory, USA*
[6] *Institute of High Performance Computing, A*STAR, Singapore*
Email: du.wu@riken.jp, jt.meng@siat.ac.cn, {wenxizhu, danierdeng}@tencent.com, wangx2@ornl.gov
luo_tao@ihpc.a-star.edu.sg, mohamed.attia@riken.jp, yj.wei@siat.ac.cn

***Abstract*—This paper presents an open-source library that pushes the limits of performance portability for irregular General Matrix Multiplication (GEMM) on the widely-used Arm architectures. Our library, autoGEMM, is designed to support a wide range of Arm processors: from edge devices to HPC-grade CPUs. autoGEMM generates optimized kernels for various hardware configurations by auto-combining fragments of auto-generated micro-kernels that employ hand-written optimizations to maximize computational efficiency. We optimize the kernel pipeline by tuning the register reuse and the data load/store overlapping. In addition, we use a dynamic tiling scheme to generate balanced tile shapes. Finally, we position autoGEMM on top of the TVM framework where our dynamic tiling scheme prunes the search space for TVM to identify the optimal combination of parameters for code optimization. Evaluations on five different classes of Arm chips demonstrate the advantages of autoGEMM. For small matrices, autoGEMM achieves 98% of peak and up to 2.0x speedup over state-of-the-art libraries such as LIBXSMM and LibShalom. For irregular matrices (i.e. tall skinny and long rectangles), autoGEMM is 1.3-2.0x faster than widely-used libraries such as OpenBLAS and Eigen. autoGEMM is publicly available at: https://github.com/wudu98/autoGEMM.**

I. Introduction

Improving the performance of small and non-squared shaped GEMM (General Matrix Multiplication) kernels can have a significant impact on speeding up DL (Deep Learning) [1], large language models [2], [3] and some scientific applications (e.g. CFD [4], N-body [5], [6], Imaging [7], [8]). Dense and large-squared GEMM is well-studied and has been highly optimized by open linear algebra libraries such as OpenBLAS [9], Eigen [10], and vendor libraries (e.g. Intel MKL [11]). These implementations achieve near-optimal computing performance using Goto's methodology [12] for large and approximately square matrices. However, optimizing irregular GEMM is still underway and has attracted heavy attention from the HPC community in recent years [13], [14], [15], [16].

The optimization of irregular GEMM through code automation on Arm architectures is motivated by the fact that Arm architectures proliferate in the landscape of computing and are increasingly gaining ground in HPC and datacenter-grade CPUs, e.g. AWS Graviton series and Nvidia Grace. Our goal is to address the challenges posed by using Arm CPUs for workloads with irregular matrix shapes and sizes, which is a common issue in deep learning (inference and training), and some scientific applications [17]. These matrices can vary greatly in size and shape, making manual optimizations for each case time-consuming and inefficient. Code generation streamlines the optimization process, making it possible to optimize for a wider range of matrix shapes and sizes with minimal manual effort. Optimized irregular and small GEMM on Arm architectures can lead to improved performance, reduced latency, and higher energy efficiency. This can have a significant impact on the overall efficiency of various DL applications [18], [19], in both edge devices and data centers.

To achieve close-to-peak performance in irregular GEMM computations, we face three major challenges. (1) *Irregular Matrix Shapes*: The shapes of matrices in various applications, such as DL and some scientific simulations [20], [21], [22], [23], [24], [25] are often irregular. Irregular matrices can arise from transformed fully connected layers [24] or convolution layers [22], [23], [25], small matrices in simulation algorithms [20], [21] , and mismatched matrix dimensions in some layers of deep neural networks [24]. These irregular shapes pose a challenge for existing BLAS libraries such as OpenBLAS [9], Eigen [10], and LibShalom [14], [26], as they are optimized for regular matrix shapes and may not perform optimally for irregular shapes. (2) *Hardware Diversity*: There are many different types of Arm SoCs and processors with varying cache sizes, memory sub-systems, pipelines, instruction sets, etc. This creates a challenge for traditional manual tuning by experts. For example, the developers of OpenBLAS stopped porting the library to newer Arm architectures after porting it to 14 different Arm architectures [9]. This diversity in hardware architectures makes it difficult to achieve optimal performance across the Arm family of architectures. (3) *Coding Difficulties*: To accommodate a large number of input

*Co-first authors; †Corresponding authors

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/31.00©2024IEEE

shapes and hardware configurations, an enormous amount of highly efficient code is required. This is a daunting task since the parameter space for hand-written codes is large, and the hardware configuration space is also significantly large.

We develop a framework that builds on hand-optimized assembly-coded wrapped in auto-generated micro-kernels that are adaptable to different input shapes, parameterizations, and hardware targets. First, we automatically generate assembly-coded micro-kernels for *irregular matrix shapes*. These micro-kernels can be generated for different tile sizes/shapes, such as 8×8, 6×12, 5×16 and 4×20, and act as building blocks for generating optimized small-sized and irregularly shaped GEMM kernels. Second, we optimize the instruction pipeline of the micro-kernels to the underlying hardware target based on a performance model of the relevant hardware features that we collect for target Arm SoCs and processors. **Adding support for a new Arm processor in our library requires only adding the relevant features of the new processor to our performance model, without the need for any changes to the library itself**. Using this performance model, the micro-kernels are then optimized at a very fine granularity using native Arm assembly for improving register reuse, and to overlap data load/store with computation by dividing the kernel to three stages: prologue, main, and epilogue. Third, we use the TVM framework [27] to auto-tune autoGEMM. We apply a dynamic micro-tiling scheme on the micro-kernels to generate code optimized for micro-tiles of balanced sizes. TVM allows us to handle matrices of any shape and tune all available parameters by calling the micro-kernels repeatedly. Unlike previous approaches, the dynamic micro-tiling algorithm aims to minimize the number of micro tiles, while maximizing the arithmetic intensity [12], [28]. By combining multiple micro tiles into sub-matrices that fit into the cache, our approach achieves significant improvements in performance over existing methods. Fourth, we drive our framework by a performance model for the dynamic micro-tiling, which allows us to achieve an end-to-end performance that is close to peak. The performance model enables TVM to prune the search space for input shapes and algorithm parameters, thereby identifying the optimal parameters for code optimization.

While some of the individual optimization techniques we use have been proposed before in other contexts, no prior-art combines these techniques to create a systematic way to auto-generate irregular matrix multiplication kernels from expert-optimized assembly building blocks that are adaptable to hardware of different characteristics. The contributions of this paper are listed below:

- We create a collection of micro-kernels that can accommodate varying tile sizes and serve as building blocks for optimizing small-sized and irregularly shaped GEMMs for Arm architectures. Additionally, we optimize the micro-kernels instruction pipeline by leveraging register reuse and latency hiding with native Arm assembly. The effectiveness of these optimizations are driven by a performance model that is proven effective by empirical results.

TABLE I: Summary of comparison with GEMM libraries w.r.t. irregular-shaped and small matrices.

	OpenBLAS	Eigen	LibShalom	FastConv	LIBXSMM	TVM	**Ours**
Hand-written Micro-kernels	✓	✓	✓	✓	✓		✓
Code Generation					✓	✓	✓
Auto-tuning				✓		✓	✓
Loop Scheduling						✓	✓
Small GEMM Efficiency (M=N=K=64)	*35%*	*50%*	*95%*	*58%*	*68%*	*78%*	***98%***
Irregular GEMM Efficiency (M=256, N=3136, K=64)	*47%*	*49%*	*86%*	*79%*	*N/A*	*72%*	***91%***

- We propose a dynamic micro-tiling algorithm to implement autoGEMM with TVM. Our algorithm combines multiple balanced micro-tiles into submatrices to minimize the number of tiles, while maximizing the arithmetic intensity. We use different micro-kernels that are auto-generated to cater to the specific sizes of the micro-tiles.
- We show end-to-end and comprehensive performance comparisons. For small matrices, autoGEMM achieves 98% of peak and a 1.5-2.0x speedup, on average, in comparison to LIBXSMM [13] and LibShalom [14], [26]. For irregular matrices, autoGEMM is 1.3-2.0x faster than widely-used libraries such as OpenBLAS [9] and Eigen [10].

The rest of this paper is organized as follows: In Section II, we provide a discussion of the background of matrix multiplication and the related works. Section III outlines the optimization of micro-kernels. Section IV presents our framework for auto-tuning kernels. Section V displays the evaluated result. Finally, Section VI concludes.

II. Background

A. Matrix Multiplication

Irregular GEMM refers to matrix multiplication on irregular-shaped or small matrices. This means that one dimension is significantly smaller than the other, or the matrix is small enough to fit into the last-level cache of the processor. Examples of irregular-shaped matrices include long-rectangular, skinny-tall, and small matrices with dimensions up to 80 [13]. To efficiently compute GEMM, it is important to maximize the data locality [12]. This involves dividing the matrices into smaller submatrices, known as *cache blocking* [29], [30], [12]. Next, those submatrices are divided into micro-tiles that fit optimally into existing registers without causing register pressure, which is known as *register tiling* [9], [12], [14], [31]. By repeatedly loading these micro-tiles into registers, we can calculate the output matrix with minimal data movement along the memory and cache subsystem. Overall, an efficient GEMM implementation optimizes data reuse and minimizes data movement to improve efficiency.

B. Related Works

Irregular and small GEMM: There are two main ways that libraries follow to optimize GEMM computations: Hand-optimized libraries and automatic code generated libraries.

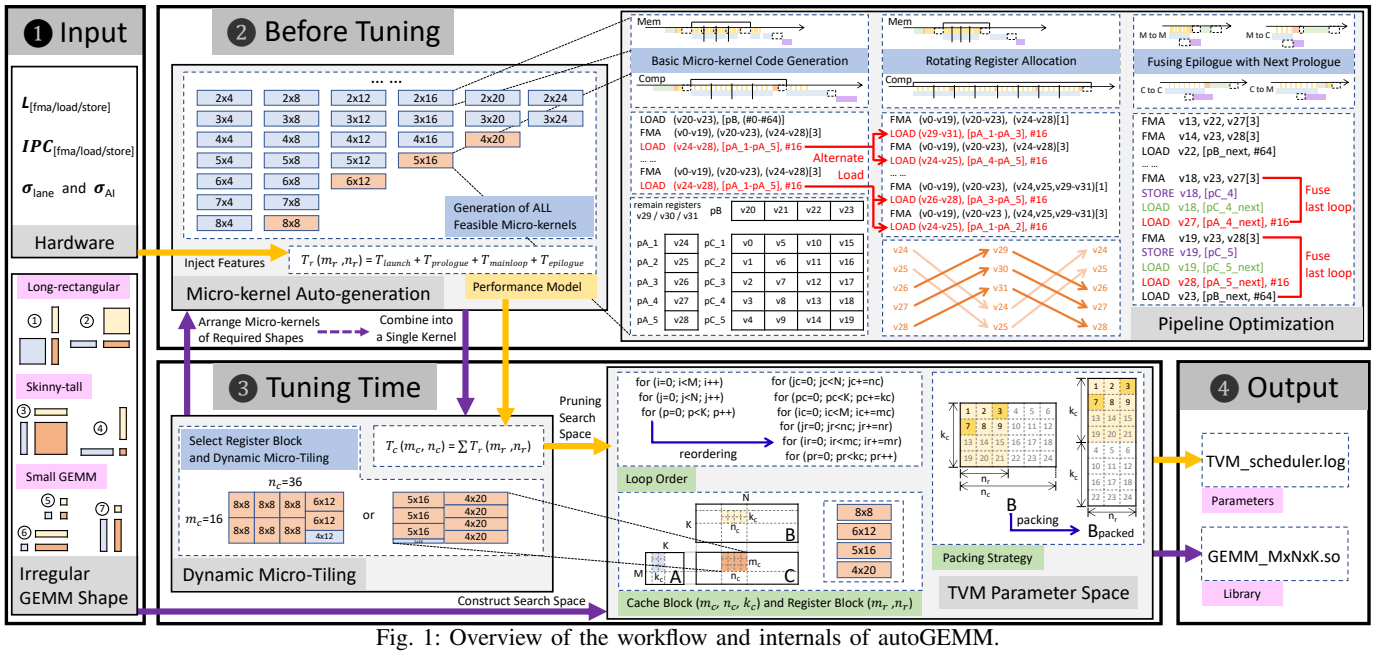


Fig. 1: Overview of the workflow and internals of autoGEMM.

Older and mature libraries such as OpenBLAS [9], Eigen [10], and LibShalom [14], [26] follow the first approach. However, due to the large space of configurations that exist for Arm processors [32], [33], it is not feasible to hand-optimize these libraries for each new Arm hardware architecture. More modern libraries, such as LIBXSMM [13] and TVM [27], focus on the second approach: generating micro-kernels with just-in-time (JIT) [34] or ahead-of-time (AOT) [35] compilation. However, this approach lacks the flexibility of hand-arranging the instruction pipelines for different hardware targets.

Code Generation and Auto-tuning: In addition to the traditional open source and vendor BLAS libraries (i.e. [9], [10], [11]), state-of-the-art libraries that can generate code and tune dense matrix/tensor operators include Google JAX [36], Halide [37], and the TVM series [27], [38], [39]. JAX [36] uses XLA [40], a specialized compiler for linear algebra, to transform numerical functions. TVM [27], [38] builds on Halide's ideas and aims to be performance portable for deep learning applications across various hardware configurations. AutoTVM [39] generates a performance model using the machine learning method XGBoost [41], and applies an auto-tuning process using statistical modeling (via a simulated annealing algorithm) to effectively remove local optimal saddle points.

We compare various GEMM libraries in Table I. OpenBLAS [9], Eigen [10], LibShalom [14], [26], and FastConv [33] have highly optimized hand-tuned kernels. Among those state-of-the-art libraries, LibShalom delivers the best performance, on average. In addition, LIBXSMM generates its own micro-kernels using JIT-based code generation techniques, while TVM adopts AOT-based code generation, auto-tuning, and loop scheduling techniques. Our work proposes a method to generate an in-library packing kernels that combine and adapt hand-optimized code blocks, automatically. We also employ TVM's auto-tuning capabilities [42], [43] to push the limits of irregular GEMM computations.

III. Generation and Performance Modeling of Micro-kernels

We proposed the **autoGEMM** framework and its workflow overview are presented in Figure 1. The workflow comprises three main steps. ❶ **autoGEMM** generates and optimizes high-performance micro-kernels, which are then automatically adapted to different Arm chip configurations. ❷ An auto-tuning mechanism searches for optimal performance parameters for irregular-shaped matrices, using the generated micro-kernels and fine-grained scheduling methods. ❸ **autoGEMM** generates high-performance code using the optimal parameters and packages it in the library. In this section, we first demonstrate the generation and optimization of micro-kernels.

A. Micro-kernel Auto-generation

This section elaborates on the auto-generation of efficient micro-kernels. Regarding a micro-kernel of size (m_r, n_r), the operations in the GEMM computation [12], [14], [44] can be expressed as:

$$C(m_r, n_r) = C(m_r, n_r) + A(m_r, k_c) \cdot B(k_c, n_r) \quad (1)$$

Here, sub-matrices of $A(m_r, k_c)$, $B(k_c, n_r)$, and $C(m_r, n_r)$ are stored in L1 Cache. In particular, n_r and k_c in Eqn(1) should be a multiple of σ_{lane}, which is 4 for Armv8 architecture and 16 for SVE-supporting [45] architectures like A64FX [46], [47] and Graviton3. We use $\vec{n_r} = n_r/\sigma_{lane}$, and $\vec{k_c} = k_c/\sigma_{lane}$ to denote the vectorized versions of n_r, and k_c, respectively.

1) Selecting the Micro-kernel Sizes: Our main goal is to generate micro-kernels with high Arithmetic Intensity (AI) [12], [33], which would allow us to fully utilize the vector registers available on the specified chip. For a single micro-tile expressed in Eqn (1), the maximum AI becomes:

$$AI_{max} = 2 \cdot m_r \cdot n_r / (m_r + n_r) \quad (2)$$

TABLE II: Auto-generated micro-kernel of various tile sizes are listed. When possible, we select the tile size with higher arithmetic intensities (colored in blue). Other tiles sizes are used to handle the corner cases.

m_r \ n_r	4	8	12	16	20	24	28
2	2.67	3.20	3.43	3.56	3.64	3.69	3.73
3	3.43	4.36	4.80	5.05	5.22	5.33	5.42
4	4.00	5.33	6.00	6.40	6.67	-	-
5	4.44	6.15	7.06	7.62	-	-	-
6	4.80	6.86	8.00	-	-	-	-
7	5.09	7.47	-	-	-	-	-
8	5.33	8.00	-	-	-	-	-

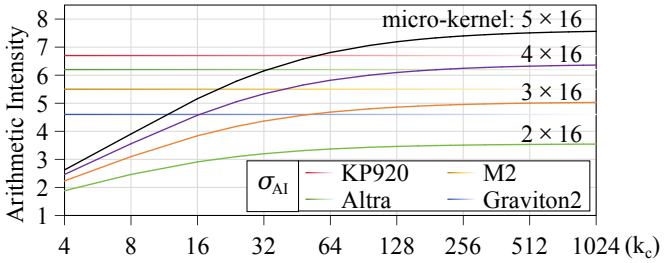


Fig. 2: Arithmetic Intensity (AI) trend for micro-kernel of tile size $m_r \times 16$ with k_c increased, as well as four different hardware σ_{AI} (the lower the easier to achieve). When k_c is small, micro-kernels with low AI are memory-bound, rather than compute-bound; the bottleneck is at the prologue and epilogue.

With 32 vector registers being the common upper limit in ARM chips, there are only 58 feasible tile sizes, part of which are enumerated in Table II. We calculate the AI value for all tile sizes and select 4 tile sizes colored in blue with higher AI as our first-choice micro-kernel shapes. The remaining tile sizes are used for filling corner cases of GEMM computation.

For multiplication on large and square matrices, a default assumption is made that $k_c \gg m_r(n_r)$ and micro-tile can always reach AI_{max}. However, in irregular matrices (especially small or skinny-tall), this assumption can no longer be guaranteed. Incorporating k_c into the Equ 2, the actual AI should be written as follows:

$$AI = 2 \cdot m_r \cdot \vec{n_r} \cdot k_c / (2 \cdot m_r \cdot \vec{n_r} + m_r \cdot \vec{k_c} + k_c \cdot \vec{n_r}) \quad (3)$$

Figure 2 illustrates the trend of AI for $m_r \times 16$ micro-kernel with a increasing k_c. Note that σ_{AI} is a threshold value obtained by micro-benchmarking a target hardware. If $AI \geq \sigma_{AI}$, the micro-kernel can potentially achieve close-to-peak performance on the specified chip. Lower σ_{AI} means it is easier to optimize on hardware, and thus the micro-kernel for corner case can also achieve close-to-peak performance.

2) Micro-kernel Code Generation: Implementation of micro-kernels for GEMM has strong regularity, and the automatic code generation method can greatly reduce the amount of hand-written codes. The generation of micro-kernels with various tile sizes listed in Table II is described by Listing 1. We automatically generate assembly codes for micro-kernels using a Python script. Here we assume that n_r and k_c is divisible with σ_{lane} to simplify the code example. The input has three parameters m_r, n_r and k_c. The output is C++ code that uses the native Arm assembly [48], [49]. Line 4 and lines 47-51 are the interface that calls the ASM code in a C++

Listing 1: Python script that generates **micro-kernel** of shape $m_r \times n_r$.

```
1 def MicroKernel_Generator(mr, nr, kc):
2     code = f"
3 void MicroKernel_{mr}x{nr}x{kc}(float A[mr][kc], float B[kc
  ][nr], float C[mr][nr], int lda, int ldb, int ldc) {
4 asm volatile(
5     prfm    PLDL1KEEP, [%[A], #64]    # Prefetch A[0][0:15]
6     prfm    PLDL1KEEP, [%[B], #64]    # Prefetch B[0][0:15]
7     prfm    PLDL1KEEP, [%[C], #64]    # Prefetch C[0][0:15]
8     lsl     %[lda], %[lda], #2        # lda = lda * 4(byte)
9     lsl     %[ldb], %[ldb], #2        # ldb = ldb * 4(byte)
10    lsl     %[ldc], %[ldc], #2        # ldc = ldc * 4(byte)
11    mov     x{6}, %[A]                # Line 11-16 for getting position
12    mov     x{6+mr}, %[C]"            # of the matrix A and C
13     for row in range(1, mr) :
14         code += f"
15    add     x{6+row}, x{6+row-1}, %[lda]
16    add     x{6+mr+row}, x{6+mr+row-1}, %[ldc]"
17     for row in range(mr) :           # Line 17-19
18         code += f"                   # Load A[mr][0] before unroll loop
19    ldr     q{mr*nr+row}, [x{6+row}], #{σ_lane*4}"
20     for col in range(nr) :           # Line 20-22
21         code += f"                   # Load B[0][nr] before unroll loop
22    ldr     q{mr*nr+mr+col}, [%[B], #{col*σ_lane*4}]"
23     code += f"
24    add     %[B], %[B], %[ldb]
25    mov     x29, #{kc}                    # The loop counter
26 1:"
27     for i in range(σ_lane) :             # Unroll Loop σ_lane
28         for col in range(nr) :           # Binding one load B
29             for row in range(mr) :       # with mr FMA
30                 code_str += f" # C[row][col]+=A[row][p_i]*B[p_i][col]
31    fma v{row*nr+col}.4s,v{mr*nr+mr+col}.4s,v{mr*nr+row}.s{{i}}"
32                 code += f"               # Load B[p_i][col]
33    ldr     q{mr*nr+mr+col}, [%[B], #{col*σ_lane*4}]"
34         code += f"
35    add     %[B], %[B], %[ldb]"
36     for row in range(mr) :               # Load A[row][p:p+σ_lane]
37         code += f"
38    ldr     q{mr*nr+row}, [x{6+row}], #{σ_lane*4}"
39     code += f"
40    subs    x29, x29, #1
41    bne     1f"                          # Loop jump to Line 26
42     for row in range(mr) :              # Line 42-45 for storing C[mr][nr]
43         for col in range(nr) :
44             code_str += f"
45    str     q{row*nr+col}, [x{6+mr+row}, #{col*σ_lane*4}]"
46     code += f"
47 : [A]"=r"(A),[B]"=r"(B),[C]"=r"(C),
48   [lda]"=r"(lda),[ldb]"=r"(ldb),[ldc]"=r"(ldc)
49 : ...
50 : "cc", "memory" ...
51 );"
52     return code
```

code. The generator generates three code blocks: The first code block (**prologue**) defines all of the needed scalar/vector registers that represent matrix A, B and C using lines 5-24. The second code block (**mainloop**), generated by Lines 25-41, will calculate $m_r \cdot \vec{n_r}$. Fused Multiply-Add (FMA) operations iteratively and accumulate the results. Finally, the third code block (**epilogue**), in Lines 42-45, terminates the process by storing the accumulated results in the main memory. We generate micro-kernels with about 200 Python code lines. This not only greatly reduces the amount of expert hand-written codes, but also aids in achieving higher compute efficiency. A quantitative model in the next subsection will discuss the performance benefits of this light-weight approach.

B. Micro-kernel Performance Modeling

After generating the micro-kernels with all feasible sizes, we use a performance model to analyze the micro-kernels' performance, be that compute or memory bound. Table III summarizes the algorithm and hardware-related parameters used in our model. We optimize the GEMM computation following a two-layered approach: cache blocking (m_c, n_c, k_c) and register tiling (m_r, and n_r). We also use parameterized data packing [50] ($\sigma_{packing}$) and loop ordering [51] (σ_{order}) to

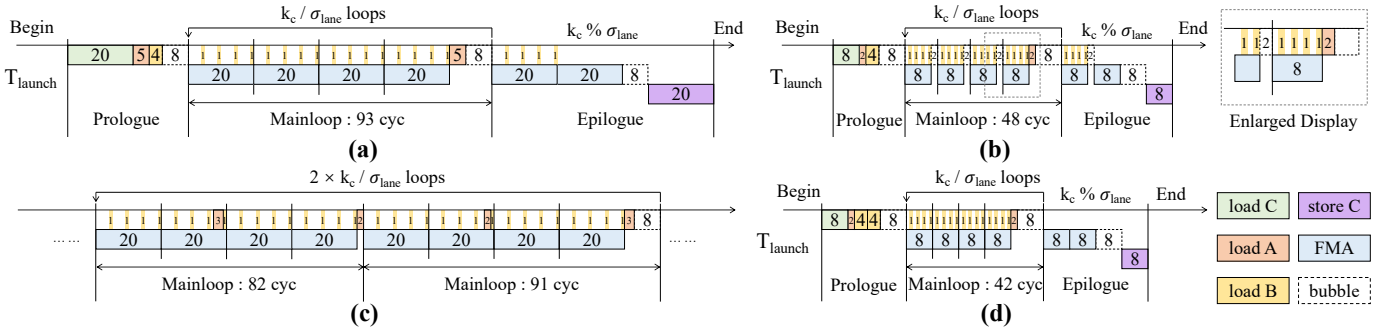


Fig. 3: Runtime measured in cycles for Listings 1. (a) is based on a generated compute-bound micro-kernel of tile size 5×16, and (b) is based on a memory-bound micro-kernel of tile size 2×16. (c) and (d) show the details of rotating register allocation optimization. (c) emphasize how the FMA instructions can overlap with the load instructions of sub-matrices $A(m_r, k_c)$, and (d) emphasizes how to overlap FMA instructions with the $B(k_c, n_r)$ sub-matrices.

TABLE III: Performance model parameters for matrix multiplication.

	Definition	Parameters
Algorithm	Input matrix shape	M, N, K
	Leading dimension	lda, ldb, ldc
	Cache block shape	m_c, n_c, k_c
	Register tile size	m_r, n_r
	Loop order mode	σ_{order}
	Data packing mode	$\sigma_{packing}$
Hardware	FMA/Load/Store instruction cycles	$L_{[fma/load/store]}$
	FMA/Load/Store instruction per cycle	$IPC_{[fma/load/store]}$
	SIMD instruction set data lane	σ_{lane}
	Threshold AI to reach peak performance	σ_{AI}

control data layout and access pattern [52]. Hardware features like FMA/load/store instruction latency ($L_{[fma/load/store]}$), instruction per cycle for FMA/load/store ($IPC_{[fma/load/store]}$), threshold AI to reach peak performance (σ_{AI}), and SIMD data lane (σ_{lane}) have an impact on register tiling, vectorization, and pipelining, which also affects the performance of the micro-kernel. Above all, our performance model leverages the concepts of instructions per cycle (IPC) and instruction cycles (latency). When there are no data dependencies, $IPC_{\#}$ instructions can be issued each clock cycle. However, when data dependencies exist, the instruction must wait $L_{\#}$ latency cycles before executing subsequent dependent instructions.

1) Performance Model of Compute-Bound Tile Sizes: The total time comprises launch time, prologue time, mainloop time, and epilogue time:

$$T_r(m_r, n_r) = T_{launch} + T_{prologue} + T_{mainloop} + T_{epilogue} \quad (4)$$

Here, T_{launch} are the cycles used to launch the micro-kernel. $T_{prologue}$ involves necessary preparation before the multiplication, such as the initial pre-fetching of matrices $A(m_r, k_c)$ and $B(k_c, n_r)$ and data loading on matrices $C(m_r, n_r)$. It can be calculated as:

$$T_{prologue} = (m_r \cdot \vec{n_r} + m_r + \vec{n_r}) \cdot IPC_{load} + L_{load} \quad (5)$$

For the actual compute portion, the projected runtime $T_{mainloop}$ in Listing 1, with FMA instructions covers all matrices $B(k_c, n_r)$ load instructions:

$$\begin{aligned} T^{comp}_{mainloop} =& m_r \cdot \vec{n_r} \cdot IPC_{fma} \cdot (\lfloor \vec{k_c} \rfloor \cdot \sigma_{lane}) \\ &+ \lfloor \vec{k_c} \rfloor \cdot (m_r \cdot IPC_{load} + L_{load}) \end{aligned} \quad (6)$$

Finally, the epilogue time includes post-remainder data calculation, and storing the result matrix $C(m_r, n_r)$ as:

$$\begin{aligned} T_{epilogue} =& m_r \cdot \vec{n_r} \cdot IPC_{fma} \cdot (k_c - \lfloor \vec{k_c} \rfloor \cdot \sigma_{lane}) + L_{fma} \\ &+ m_r \cdot \vec{n_r} \cdot IPC_{store} \end{aligned} \quad (7)$$

We combine the above Eqn(5), (6), (7) to project the total runtime in Eqn(4) for the generated code with a given tile size (m_r, n_r) from Listing 1. For example, the runtime of tile size 5×16 is illustrated as an example in Figure 3-(a). In this example, we assume load, store, and FMA to take 8 cycles ($L_{[load/store/fma]} = 8$): the IPC is equal to 1 ($IPC_{[load/store/fma]} = 1$). First, after the launch time of T_{launch}, we load $C(m_r, n_r)$ with 20 cycles, and the first load of $A(m_r, k_c)$ and $B(k_c, n_r)$ takes 5 and 4 cycles. Another 8 cycles are required for the last loading instruction to finish. The combined FMA instructions take 80 cycles to cover all matrices $B(k_c, n_r)$ load instructions. For every unroll σ_{lane} loops, there are 5 matrices $A(m_r, k_c)$ load instructions that could not be overlapped (line 38 in Listing 1), and require an additional 8 clock cycles before the next iteration starts. Finally, we wait 8 cycles for the FMA instruction calculation to finalize, and then store $C(m_r, n_r)$ in the last 20 cycles. All in all, in addition to the launch time, the micro-kernel generated from tile size 5×16 will use $20 \cdot k_c + 13 \cdot \lfloor \vec{k_c} \rfloor + 65$ cycles.

2) Performance Model of Memory-Bound Tile Sizes: As we listed in Table II, not all tile sizes of micro-kernels have high enough AI to be compute-bound. For these micro-kernels, FMA instructions in the main loop can no longer overlap with all the load instructions of the matrices $B(k_c, n_r)$. Based on the micro-kernel generated by Listing 1, the runtime $T_{mainloop}$ in the memory-bound case is:

$$T^{mem}_{mainloop} = m_r \cdot IPC_{load} \cdot \lfloor \vec{k_c} \rfloor \cdot \sigma_{lane} + L_{load} \cdot \lfloor \vec{k_c} \rfloor \cdot (\sigma_{lane} + 1) \quad (8)$$

Figure 3-(b) illustrates a memory-bound micro-kernel of tile size 2×16 on the same parameters settings as Figure 3-(a). In the main loop part, the FMA instructions cannot completely overlap with the load instructions of submatrices $B(k_c, n_r)$, leading to a bubble of 2 cycles for every iteration. The 2 cycles here are the minimum cycles required to ensure that a register satisfies the $FMA \rightarrow LOAD \rightarrow FMA$ dependency. Thus, the projected main loop runtime for the generated micro-kernel of tile size 2×16 is $48 \cdot \lfloor \vec{k_c} \rfloor$ cycles.

C. Micro-kernel Performance Optimization

This section explains our technique for optimizing the instruction pipeline automatically instead of relying on hand-optimization (as in [9], [14] for example). This involves overlapping the memory access operations with arithmetic computations.

1) Rotating Register Allocation: Rotating Register Allocation [53], [54], [55], [56] is a classic technique used to optimize pipelines in the field of software engineering and compiler design. It is also used in the main micro-kernel of the latest BLAS libraries [14], [26]. The goal of rotating register allocation in micro-kernel is to use redundant hardware registers to increase the volume of overlap between the FMA and load instructions. For the compute-bound micro-kernels (Section III-B1), this technique can help in overlapping the compute and load instructions of sub-matrices $A(m_r, k_c)$. For every $2 \cdot \sigma_{lane}$ iterations, the load instructions of sub-matrices $A(m_r, k_c)$ can be completely overlapped by the FMA instructions once. Eqn(6) can thus be updated as follows:

$$\begin{aligned} T^{comp}_{mainloop} =& m_r \cdot \vec{n_r} \cdot IPC_{fma} \cdot (\lfloor \vec{k_c} \rfloor \cdot \sigma_{lane}) \\ &+ \lceil \frac{\lfloor \vec{k_c} \rfloor}{2} \rceil \cdot (m_r \cdot IPC_{load} + L_{load}) \end{aligned} \tag{9}$$

Figure 3-(c) shows the details of the pipeline after rotating register allocation. Note that the 2 and 3 cycles in red come from using hardware redundant registers (3 registers for micro-kernel 5×16) to load sub-matrices $A(m_r, k_c)$ in advance. According to the parameters settings and updated model, besides the launch time, the projected runtime of the micro-kernel of tile size 5×16 will be $20 \cdot k_c + 13 \cdot \lceil \frac{\lfloor \vec{k_c} \rfloor}{2} \rceil + 65$ cycles. For the case of memory-bound micro-kernels (Section III-B2), this technique can allow us to overlap the load instructions of sub-matrices $B(k_c, n_r)$. In the prologue stage, we double the loading of multiple-iterations of sub-matrices $B(k_c, n_r)$ in the registers; the dependency of $FMA \rightarrow LOAD \rightarrow FMA$ in the pipeline of main loop stage is eliminated, and no bubbles appear between the FMA instructions. When load instructions are overlapped with FMA instructions, Eqn(8) can be optimized as:

$$\begin{aligned} T^{mem}_{mainloop} =& m_r \cdot \vec{n_r} \cdot IPC_{fma} \cdot (\lfloor \vec{k_c} \rfloor \cdot \sigma_{lane}) \\ &+ \lfloor \vec{k_c} \rfloor \cdot (m_r \cdot IPC_{load} + L_{load}) \end{aligned} \tag{10}$$

Figure 3-(d) illustrates this optimization based on Figure 3-(b). The projected main loop runtime for micro-kernel of tile size 2×16 now becomes $42 \cdot \lfloor \vec{k_c} \rfloor$ cycles.

2) Fusing the Epilogue with the Following Prologue: Latency hiding of the epilogue and prologue can also save a few cycles in the generated micro-kernel. When the k_c dimension is not large enough in some irregular matrix shapes, the time used for the epilogue and prologue will increase. For example, the micro-kernel with a shape of 5×16 and $k_c = 18$, $T_{prologue}$ and $T_{epilogue}$ account for 8.2% and 15.1% of the total micro-kernel projected runtime. We fuse the epilogue FMA with the ending store operations and load instructions in the next loop's prologue. This overlaps the load/store with the arithmetic instructions in the previous iteration and eliminates

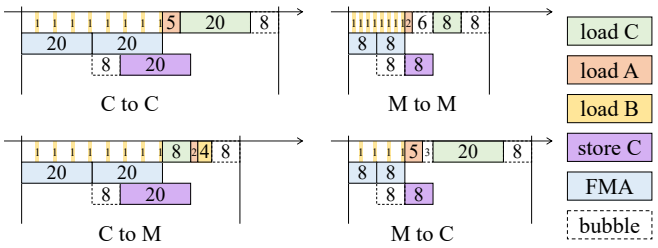


Fig. 4: Four fusion methods to fuse the current epilogue with the following prologue: c_to_c, m_to_m, c_to_m, m_to_c, based on whether the current and next micro-kernels are compute or memory-bound.

T_{launch} to improve the micro-kernel's efficiency. As shown in Figure 4, there are four types of fusion: c_to_c, m_to_m, c_to_m, m_to_c based on whether the current and next micro-kernel are compute or memory bound. Take the current and next micro-kernel to be both compute-bound as an example, the projected runtime can be calculated as:

$$\begin{aligned} T^{c\_to\_c}_{fuse\_epi\_pro} =& m_r \cdot \vec{n_r} \cdot IPC_{fma} \cdot (k_c - \lfloor \vec{k_c} \rfloor \cdot \sigma_{lane}) \\ &+ (m_r \cdot \vec{n_r} + m_r) \cdot IPC_{load} + L_{load} \end{aligned} \tag{11}$$

In summary, the traditional method of manually implementing the above optimizations on all chips, tile sizes (listed in Table II), and different numbers of epilogue iterations is not practical. The micro-kernel autogeneration with the above optimization approaches proposed in this section is a practical solution for irregular GEMM. It also serves as a prerequisite for enabling the parameterized tuning work in the following section.

IV. Auto-constructing and Tuning of the Kernel

After orchestrating the micro-kernels instruction pipeline, auto-constructing is used to comprise different micro-kernels into a kernel, after which auto-tuning is further used to achieve close-to-peak performance. In this section, we first propose a dynamic micro-tiling algorithm to split the sub-matrix $C(m_c, n_c)$ into multiple micro tiles. Then a performance analysis is presented in Section IV-B to explain the effectiveness of dynamic tiling on merging the sub-matrix $C(m_c, n_c)$. Finally, TVM is employed to tune all the algorithm parameters, on a pruned search space, as a final step to enhance performance.

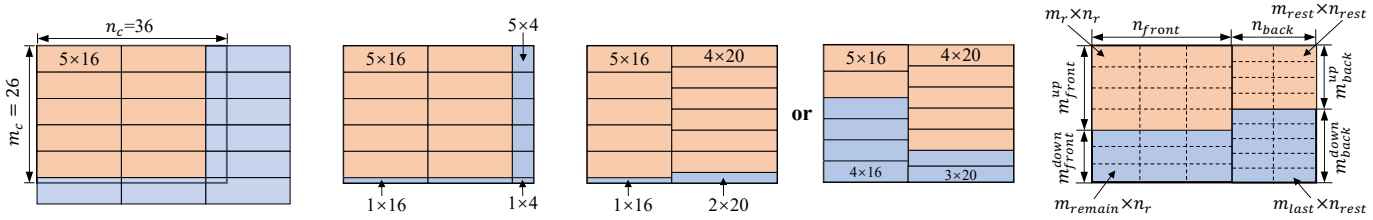
A. Dynamic Micro-Tiling

1) Static Micro-tiling: With two input matrices $A(M, K)$ and $B(K, N)$, we first divide matrices A, B and the output matrix C into sub-matrices that can fit in cache [12], [28] as follows:

$$C(m_c, n_c) = C(m_c, n_c) + A(m_c, k_c) \times B(k_c, n_c) \tag{12}$$

Each sub-matrix is the minimum scheduling unit executed by multiple threads. It can be further split into multiple micro-tiles for better utilization of registers. Thus the micro-tiling algorithm is critical to the performance of irregular GEMM.

Micro-tiling methods have been applied in several previous research works [9], [13]. An illustration of the different micro-tiling strategies is presented in Figure 5 with a sub-matrix



(a) OpenBLAS **(b)** LIBXSMM **(c)** DMT(ours): n_r=16 **(d)** DMT(ours): n_r

Fig. 5: Micro-tiling strategies for OpenBLAS, LIBXSMM, and DMT (ours) on a given sub-matrix of size $m_c \times n_c = 26 \times 36$: a) OpenBLAS uses a single fixed tile size and pads (blue tiles are padded), b) LIBSXMM uses a single fixed tile size and other sizes for edge columns and rows, c) DMT (ours) dynamically tiles the sub-matrix with high arithmetic intensity, and are balanced in size; (left) more suitable for hardware with high σ_{AI}; (right) suitable for hardware with low σ_{AI}, and d) the general case for DMT (ours) for any given sub-matrix size.

Algorithm 1 Dynamic Micro-Tiling (DMT) Algorithm

Input: : Sub-matrix to tile $C(m_c, n_c)$.
Output: : Minimum $T(m_c, n_c)$ and corresponding parameters.

1 $\mathcal{P}$ = $+\infty$
2 **for** $n_{front} = 0$ *to* n_c **do**
3 **for** $m^{up}_{front} = 0$ *to* m_c **do**
4 **for** $m^{up}_{back} = 0$ *to* m_c **do**
5 $n_{back} = n_c - n_{front}$
6 $m^{down}_{front} = m_c - m^{up}_{front}$; $m^{down}_{back} = m_c - m^{up}_{back}$
7 $\mathcal{P}_{new}$ = $T(m^{up}_{front}, n_{front})$ + $T(m^{down}_{front}, n_{front})$ + $T(m^{up}_{back}, n_{back})$ + $T(m^{down}_{back}, n_{back})$
8 $\mathcal{P}$ = min ($\mathcal{P}$, $\mathcal{P}_{new}$)
9 **return** minimum $\mathcal{P}$ and parameters set get minimum $\mathcal{P}$
10
11 **Function** T (m, n) **:**
12 $\mathcal{Q}$ = $+\infty$
13 **while** *(m_r, n_r) in Table II* **do**
14 $\mathcal{Q}_{new}$ = $(m/m_r) \cdot$(n / n_r) $\cdot$T$_r(m_r, n_r)$
15 $\mathcal{Q}$ = min ($\mathcal{Q}$, $\mathcal{Q}_{new}$)
16 **return** minimum $\mathcal{Q}$ and m_r, n_r get minimum $\mathcal{Q}$

of shape $C(26, 36)$. OpenBLAS (Figure 5-(a)) employs micro tiles with a single static shape of 5×16: for this example, 8 corner micro-tiles are generated with padding that will penalize performance. LIBXSMM (Figure 5-(b)) improves this by tiling the edge columns and rows with a different tile size. This results in micro-kernels at the edges that may have extremely low arithmetic intensity. For irregular matrices, the performance degradation can be significant for both redundant works due to padding (OpenBLAS) and edge tiles with low arithmetic intensity (LIBXSMM).

2) Dynamic Micro-Tiling Algorithm: To improve the efficiency of the calculations on the sub-matrix $C(m_c, n_c)$ of an irregular matrix, we propose a dynamic programming method, Dynamic Micro-Tilling (DMT), to find the optimal solution satisfying following conditions: 1) split the sub-matrix into micro-tiles that have high arithmetic intensity, 2) balance the different sizes of tiles to avoid extremely small tile with low arithmetic intensity, and 3) minimize the number of tiles. In Algorithm 1, DMT has 3 steps: the first step is to divide the sub-matrix into 4 parts using three parameters n_{front}, m^{up}_{front}, and m^{up}_{back}, in Line 2-4. In the second step, for each divided part, traverse all micro-kernel shapes in Table II and calculate the minimum projected runtime, in Line 11-16. Finally, we accumulate the projected runtime of the four parts, and record the minimum projected runtime, in Line 7-8.

We explain the DMT algorithm in Figure 5-(c) with the same example of sub-matrix shape $C(26, 36)$. We show two possible tilings using DMT for this specific sub-matrix shape, since the hardware σ_{AI} affects DMT results. For hardware with low σ_{AI}, micro-tiles of shapes 4×16 and 3×20 can also achieve close-to-peak performance, so the right one is a preferable solution for a sub-matrix without low arithmetic intensity tiles. While the left one is more suitable for hardware with high σ_{AI}, since it will generate the fewest low arithmetic intensity tiles. OpenBLAS and LIBXSMM would both have had 18 micro tiles, whereas DMT has 13 micro tiles in total. In terms of arithmetic intensity, LIBXSMM has 8 micro tiles with low arithmetic intensity, but DMT has at most 2.

B. Tile Size Search Space Pruning

For a given sub-matrix shape $m_c \times n_c$, the total number of FMA instructions used is fixed. If the load/store instructions can be fully overlapped with the FMA instructions, the best performance can be achieved regardless of the register block size chosen. However, the performance of the edge tiles is much lower than that of the non-edge tiles since there are not enough FMA instructions to hide the latency of the load/store instructions. We combine the projected runtime of all tiles in Figure 5-(d) to estimate the total runtime of sub-matrix $C(m_c, n_c)$:

$$\begin{aligned} T_c(m_c, n_c) =& (m^{up}_{front}/m_r) \cdot (n_{front}/n_r) \cdot T_r(m_r, n_r) \\ &+ (m^{down}_{front}/m_{remain}) \cdot (n_{front}/n_r) \cdot T_r(m_{remain}, n_r) \\ &+ (m^{up}_{back}/m_{rest}) \cdot (n_{back}/n_{rest}) \cdot T_r(m_{rest}, n_{rest}) \\ &+ (m^{down}_{back}/m_{last}) \cdot (n_{back}/n_{rest}) \cdot T_r(m_{last}, n_{rest}) \end{aligned} \quad (13)$$

With Eqn(13), we can do a theoretical estimation of the runtime with the parameters listed in Table III. Eqn(13) acts as a performance model in TVM to prune the search space by estimating the runtime for a certain combination of parameters.

C. Autotuning with TVM

TVM's [27] priority is to handle matrices of any shape, and then tune all available parameters by calling performance-aware kernels repeatedly. The tuning process can take hours or even days to get optimal parameters, using Eqn (13) to prune the search space can drop the tuning time dramatically.

1) Using TVM with autoGEMM: The generated kernel appears as an external library to TVM. Since TVM does not expose the parameterization for external libraries, we patched the TVM framework to embed directly the kernels

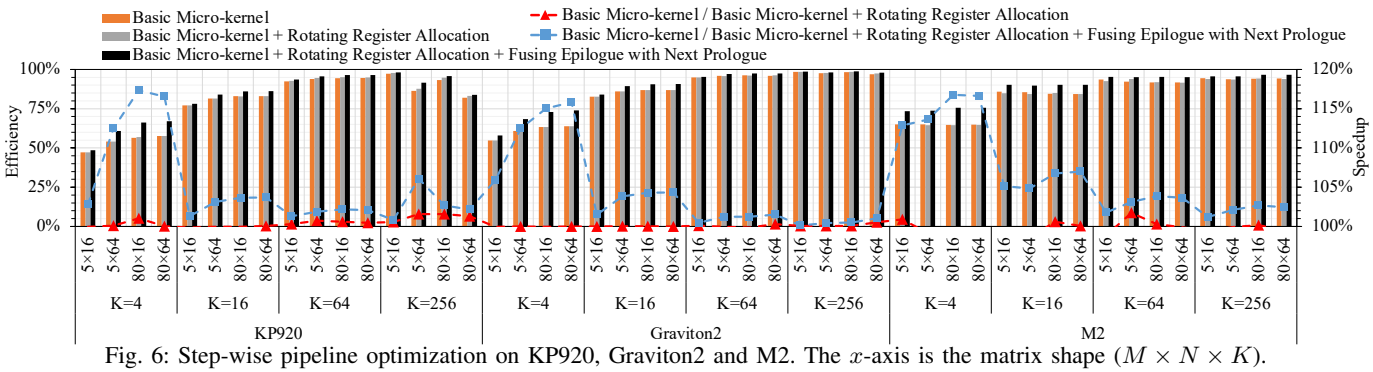


Fig. 6: Step-wise pipeline optimization on KP920, Graviton2 and M2. The x-axis is the matrix shape ($M \times N \times K$).

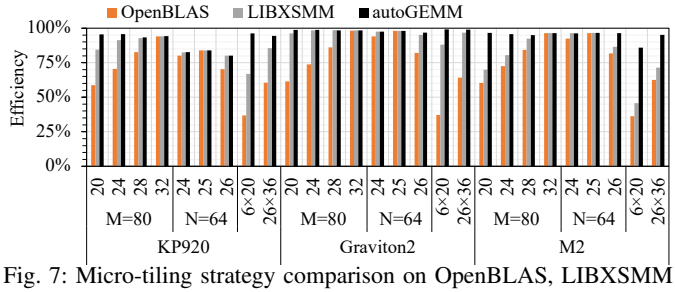


Fig. 7: Micro-tiling strategy comparison on OpenBLAS, LIBXSMM and autoGEMM (using DMT) with KP920, Graviton2, and M2.

TABLE IV: Hardware Specifications

	Huawei KP920	AWS Graviton2	Ampere Altra	Apple M2	Fujitsu A64FX
Cores	8	16	70	4(+4)	48(+2)
Frequency	8@2.60	16@2.50	70@3.0	4@3.49	48@2.20
L1i Cache	8@64K	16@64K	70@64K	4@192K	48@64K
L1d Cache	8@64K	16@64K	70@64K	4@128K	48@64K
L2 Cache	8@512K	16@1M	70@1M	16M-share	4@8M-share
L3 Cache	32M-share	32M-share	32M-share	None	None
SIMD(bit)	NEON(128)	NEON(128)	NEON(128)	NEON(128)	SVE(512)
SMP	1	1	2(NUMA)	1	1(ccNUMA)
Type	SoC	Datacenter	Datacenter	Consumer	Supercomputer

we generated using the code generation method proposed in Section III-A2, so that TVM could interweave the tuning of parameters directly into the assembly code.

2) Searching the Parameters Space: All the algorithm related parameters in our implementation are included in Table III. Among these parameters, M, N, K are the input problem size. m_c, n_c and k_c are cache blocking sub-matrix shape; and m_r, n_r are micro-tile size. Packing strategy $\sigma_{packing}$, and loop order σ_{order} are the tuning parameters related to the memory access pattern. **Micro-tile size** is an autoGEMM internal parameter used to select the main micro-kernels listed in Table II. **Cache blocking** yields a huge parameter search space in irregular matrix multiplication. We can search all possible combinations satisfying $0 < m_c \leq M, M\%m_c = 0$; $0 < n_c \leq N, N\%n_c = 0$ and $0 < k_c \leq K, K\%k_c = 0$. **Loop order** controls the memory access pattern: it includes all permutations of the above block parameters (m_c, n_c, k_c, m_r and n_r) to get $5! = 120$. **Packing strategy** permutes the possible transformations of sub-matrix's layout to get unit-strided access on the vectorized dimension. This may introduce additional read and store operations as an overhead. When the N dimension is relatively small, the performance benefits from data locality may not justify the packing time and hence we skip the packing step [26], [50]. Finally, three options for packing are provided: none, offline, and online.

V. Evaluation

Our framework autoGEMM relies on AOT code generation and TVM-based auto-tuning to achieve close-to-peak performance. The correctness of our implementation has been verified against all other libraries we compare with by ensuring the relative error is less than 1e-6. The performance of our implementation is compared against Eigen [10], OpenBLAS [9], LIBXSMM [13], LibShalom [14], [26], and Fugaku SSL2 [57].

A. Evaluation Environment and Data

We evaluated five Arm processors, which are detailed in Table IV. The data center servers and supercomputer used Linux kernel versions 5.4.0 and 4.18.0, as well as GCC versions 9.4.0 and 8.4.1, respectively. The MacBook M2 used Darwin Kernel Version 21.5.0 and clang version 13.1.6. TVM's version was v0.10 release, and LLVM's version was 10.0.0, for all five devices. We conducted evaluations on the following datasets: **(1) Small Matrices:** We investigated the effects of step-wise optimization on small matrices, varying in shape from (1,1,1) to (128,128,128). Our evaluation included three parts: a) a step-by-step assessment of micro-kernel generation and optimization; b) an evaluation of the dynamic micro-tiling algorithm; and c) a comparison of our work with other relevant studies on small matrices. **(2) Irregular Matrices:** To evaluate performance for deep learning applications, we used matrix shapes generated by Resnet-50 [22] and [58], [59], [60], [61]. These shapes include all three types of irregular matrices: tall and skinny, long and rectangular, and small matrices.

B. Step-wise Evaluation on Generated Micro-kernels

The GEMM code generation and pipeline optimizations presented in Section III are evaluated step-by-step. The code generated by micro-kernel generation (Listing 1 in Section III-A), rotating register allocation (Section III-C1), and fusing epilogue with next prologue (Section III-C2) are evaluated on three Arm CPUs: KP920, Graviton2, and M2. Then we compare our dynamic micro-tiling algorithm (DMT) with OpenBLAS and LIBXSMM strategy to confirm its performance benefits.

The step-by-step evaluation results are collected and presented in Figure 6. As can be seen from our two-step pipeline optimization, autoGEMM improves performance over Listing 1. When we increase the size of K-dimension, the efficiency increases from lower than 60% to upper than 95%, and

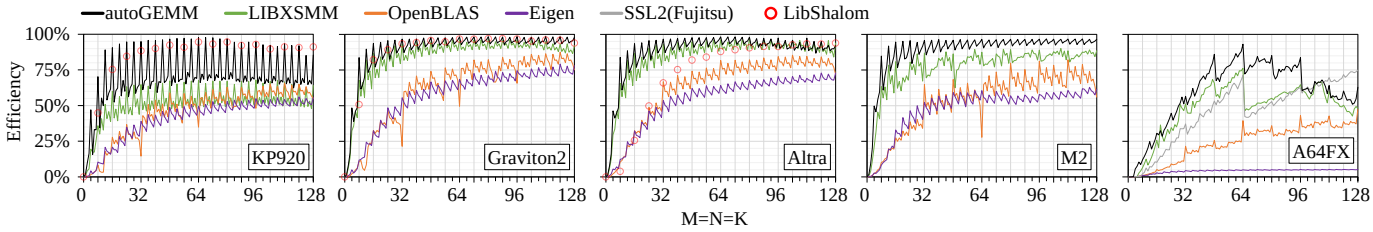


Fig. 8: Performance evaluation with single core for small GEMM. LibShalom can only correctly compute shapes when N and K are both divisible by 8, so its results appear as a few points. Additionally, LibShalom does not support M2 and A64FX. SSL2 is only used on A64FX.

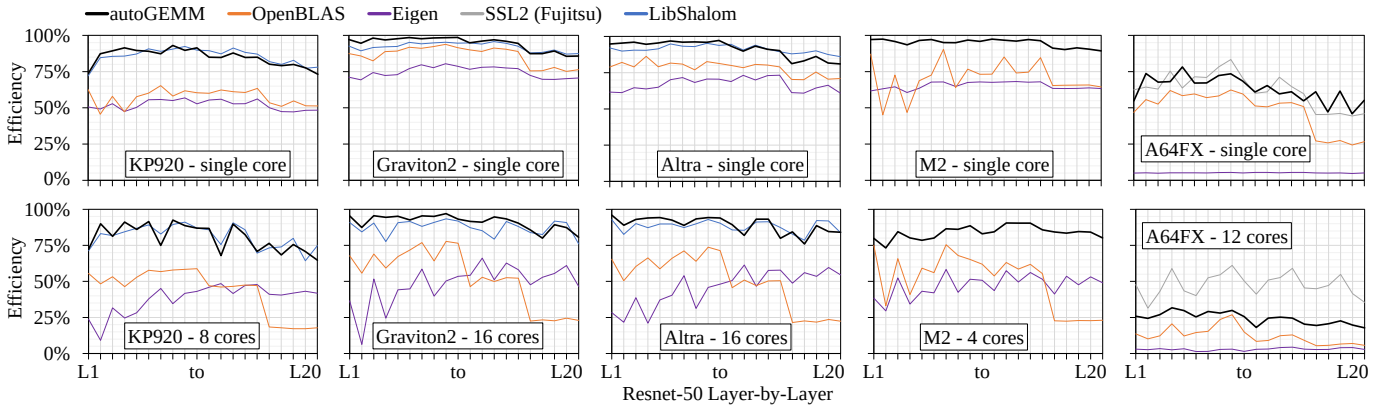


Fig. 9: Performance (upper: single core; lower: multi-cores) for irregular GEMM kernels corresponding to the 20 layers of Resnet-50 on five Arm processors. LibShalom does not support M2 and A64FX. SSL2 is only used on A64FX.

reaches 98.78% of the hardware peak efficiency on Graviton2. In the case when the K-dimension increases from 64 to 256 on $N = 64$, the efficiency on KP920 drops dramatically. As there is 64KB of L1 cache in KP920 and the size of matrix B is beyond 64KB, part of the data for the matrices A, B and C will be stored in L2 cache, and on KP920 the latency of loading instructions increases dramatically when accessing the data in the L2 cache. We highlight the following trends in the step-by-step performance evaluation results: 1) Individual optimizations have different effects on different processors. For example, the rotating register allocation optimization on KP920 has a 3% improvement, but Graviton2 and M2 do not benefit from it due to a larger hardware out-of-order execution window. 2) When using the fusing epilogue with next prologue optimization, when $K = 4$, we can see fairly consistent performance improvement on the three processors, 17.3%, 15.8%, and 16.7%. Our performance model (Eqn (11)) in Section III-C2 accurately projects that the setting $K = 4$ and $m_r = 5$, $n_r = 4$ can result in the projected improvements in the step-by-step performance results.

We compare the micro-tiling algorithm of the static strategies (OpenBLAS, LIBXSMM) with the dynamic strategy of autoGEMM using KP920, Graviton2, and M2. The comparison results are reported in Figure 7. When $M \times N = 80 \times 32$ or $M \times N = 25 \times 64$, the tiling results for these three algorithms use tiles of the same size, 5×16: there are no performance gains with autoGEMM. For the other cases, the dynamic micro-tiling of autoGEMM generates balanced tiles with higher arithmetic intensity, yielding a significant performance improvement over static strategies and always guarantees close to peak performance. The case of $M \times N = 26 \times 64$ on three hardware targets validates our analysis of the relationship between micro-kernel AI and hardware σ_{AI} in Figure 2, and the dynamic tiling strategy of autoGEMM in Figure 5-(c). On high-σ_{AI} hardware (KP920), our strategy is the same as LIBXSMM: generate the minimum number of low AI tiles. On low-σ_{AI} hardware (Graviton2 and M2), our performance improves in comparison to LIBXSMM. In particular, 4×16 shape micro-kernel at the edges can achieve peak performance, i.e. we completely eliminated tiles with low AI. Additionally, for Graviton2 with lower σ_{AI}, the penalty for calling low AI tiles of shape 1×16 is smaller. In conclusion, autoGEMM's dynamic micro-tiling plays an important role in improving the efficiency of irregular GEMM.

TABLE V: Irregular GEMM shapes from Resnet-50 [22].

Layer	M	N	K	Layer	M	N	K	Layer	M	N	K	Layer	M	N	K
L1	64	12544	147	**L6**	128	784	256	**L11**	256	196	512	**L16**	512	49	1024
L2	64	3136	64	**L7**	128	784	1152	**L12**	256	196	2304	**L17**	512	49	4608
L3	64	3136	576	**L8**	512	784	128	**L13**	1024	196	256	**L18**	2048	49	512
L4	256	3136	64	**L9**	512	784	256	**L14**	1024	196	512	**L19**	2048	49	1024
L5	64	3136	256	**L10**	128	784	512	**L15**	256	196	1024	**L20**	512	49	2048

C. Irregular and Small GEMM Evaluation

In Figure 8 we compare autoGEMM with five state-of-the-art libraries on five Arm processors using small matrices. For fairness, we exclude the time to generate the code JIT from the runtime and only the actual computation time was considered for LIBXSMM since it uses the JIT method. Additionally, LibShalom does not compile properly with clang and also does not support SVE (Scalable Vector Extension), and thus cannot be evaluated on the M2 and A64FX devices. For M=N=K$\leq$24, autoGEMM significantly outperforms other libraries, in comparison with LIBXSMM and LibShalom on

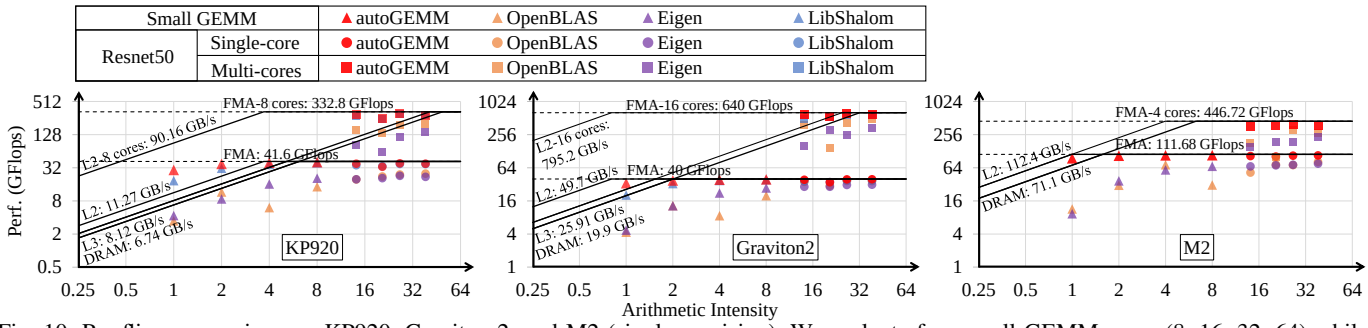


Fig. 10: Roofline comparison on KP920, Graviton 2, and M2 (single precision). We evaluate four small GEMM cases (8, 16, 32, 64) while keeping $M = N = K$. The irregular-shaped matrices are four layers (L4, L8, L10, L16) selected from Resnet-50 in Table V. autoGEMM running on a single-core is denoted as single-core, and with all available cores is denoted as multi-cores.

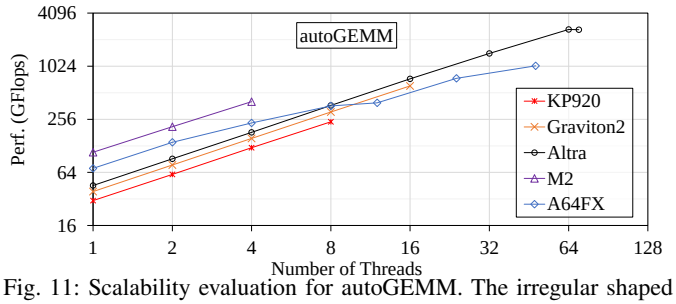


Fig. 11: Scalability evaluation for autoGEMM. The irregular shaped GEMM problem is L1 layer in Table V.

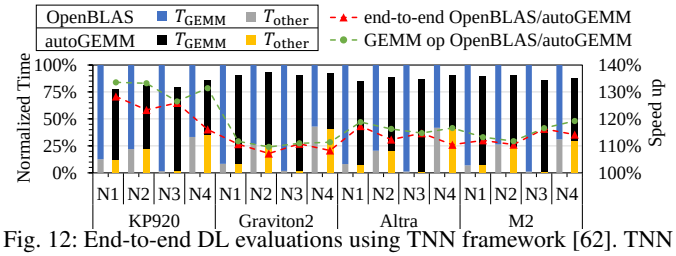


Fig. 12: End-to-end DL evaluations using TNN framework [62]. TNN is used with default components, while only replacing the calls of its GEMM routines from OpenBLAS to autoGEMM. The runtime for CONV and FC layers by calling GEMM operations is T_{GEMM}, and the time used on other non-GEMM operations is denoted as T_{other}.

five hardware, there is a 1.5x-2.0x speedup. That is mainly due to our more flexible combination of micro-kernels, and the pipeline optimization. As the matrices sizes continues to increase, we consistently achieve near-peak performance and have a minimum of 5% performance advantage over the best second-place library, on all different processors. We emphasize that autoGEMM achieves a near-peak efficiency of 97.6% on KP920, 98.3% on Graviton2, 98.4% on Altra, 96.5% on M2, and 93.2% on A64FX, when M=N=K=64. This indicates that load/store instructions are almost perfectly overlapped by FMA. However, when M=N=K=128 on KP920, our performance is lower than LibShalom, which benefits from hand-written data prefetching instructions. Our method does not employ L1 prefetch instructions, which is based on an assumption that there are no L1 cache misses in the micro-kernel, and only keeps L2 prefetch instructions in kernel.

We report single-thread and multi-threads evaluations on irregular-shaped matrices generated by the ResNet-50 deep neural network. The layer-by-layer shapes of matrices are presented in Table V. We compare the performance of autoGEMM with OpenBLAS, Eigen, SSL2(Fujitsu), and LibShalom, as shown in Figure 9. As LibShalom uses an offline packing routine for matrix B to improve reults on larger matrices. Similarly, autoGEMM is also flexible in enabling offline packing for near-peak performance, whereas traditional BLAS library do not support offline packing. In terms of single-thread evaluation, we outperform OpenBLAS and Eigen with an average improvement of 1.3x (up to 1.9x) and 1.5x (up to 2.0x), respectively. In comparison to LibShalom, autoGEMM achieves similar performance on KP920, and slightly outperforms LibShalom by 2-8% on Graviton2 and Altra. In terms of multi-threads evaluation, the result show that autoGEMM achieves comparable performance on KP920 and achieves an average of 8% (up to 20%) performance boost over LibShalom on Graviton2. However, note that when multi-cores autoGEMM implementation runs a matrix with a large K dimension (L7,L12,L17,L20), its performance may decrease. This is because TVM, as an external framework, does not support parallelism over the K dimension (k_c consistent with K). Consequently, autoGEMM cannot fully leverage the hardware capabilities on multi-cores. The multi-core performance on the A64FX is also not satisfactory (yet remains competitive at the single-core level). The possible explanations is that we transplant autoGEMM directly by replacing NEON vector intrinsic with A64FX's SVE intrinsic with no special optimizations. Further improvement is to be expected as we continue to optimize for SVE instructions on multi-cores.

D. Roofline Analysis

Figure 10 shows roofline [63] for both small and irregular-shaped matrices. The performance of small GEMM in most cases is not limited by memory bandwidth, although end-to-end execution of small matrices may be bounded by the DRAM bandwidth. As small GEMM becomes compute bound, the performance of autoGEMM is closer to peak GFLOPS than that of LibShalom, OpenBLAS, and Eigen. The shape extracted from Resnet50 has larger arithmetic intensity than small matrices and is typically in the compute bound realm. On a single core, autoGEMM is close to the peak of the roofline, while on multi-cores, autoGEMM can easily exceed the upper bounds of DRAM and L3 cache.

E. Scaling Evaluation

In Figure 11, we illustrate a strong scaling evaluation conducted on all five chips. The results demonstrate that autoGEMM achieves almost linear scaling on all chips with a parallel efficiency of 98%, 98.2%, 83.2%, 93.5%, and 30.3% on KP920, Graviton2, Altra, M2, and A64FX, respectively. The findings also reveal autoGEMM's poor scalability results on A64FX, which could be attributed to the performance tuning points being limited to 12 cores in one Core Memory Group (CMG), and the ccNUMA architecture with a ring bus between 4 CMGs that could affect scalability [64].

F. Deep Neural Network Evaluation

In this section, we present our experiments on performing end-to-end evaluations for popular DL networks. In Figure 12, we integrated autoGEMM or OpenBLAS into Tencent's TNN framework [62] and evaluated its performance on four DL models: ResNet50 (N1) [22], Inception-V3 (N2) [58], Mobilenet-V1 (N3) [59] and Squeezenet (N4) [60]. We denote the convolution (CONV) and fully-connected (FC) operators in the model as *GEMM*, while the non-GEMM operators are collectively denoted as *Other*. Figure 12 shows that the time consumed by *Other* is identical for both OpenBLAS and autoGEMM, but a noticeable decrease in normalized time for the T_{GEMM} part is observed with autoGEMM. We obtained the best results on KP920, with a speedup of 1.30x on all four models. On Graviton2, the speedup is between 1.08x to 1.15x.

VI. Conclusion

This paper presents autoGEMM, an open-source library for optimizing GEMM computation on Arm architectures. It automatically combines hand-optimized assembly code fragments, tunes register reuse, and overlaps data load/store for better performance. Dynamic tiling generates balanced tile shapes based on matrix shape. Evaluations on five different classes of Arm chips demonstrate the advantages of autoGEMM. For small matrices, autoGEMM achieves 98% of peak and a 1.5-2.0x speedup over state-of-the-art libraries such as LIBXSMM and LibShalom. For irregular matrices (i.e., tall skinny and long rectangle), autoGEMM is 1.3-2.0x faster than widely-used libraries such as OpenBLAS and Eigen.

Acknowledgments

This work was partly supported by the Key Research and Development Project of Guangdong Province under grant No. 2021B0101310002, Shenzhen-HongKong Joint Funding Project (A) under Grant No. SGDX20230116092056010, the National Key Research and Development Program of China under Grant No. 2021YFF0901102, 2021YFF1200100 and 2021YFF1200104, Shenzhen Key Laboratory of Intelligent Bioinformatics under Grant No. ZDSYS20220422103800001.

This work was also supported by Shanghai Zelixir Biotech Company by its Joint Lab of Zelixir-SIAT, and Tencent with 2 years continuous funding support.

This manuscript has been co-authored by ORNL, operated by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S.Department of Energy. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16 344–16 359, 2022.

[3] K. Hong, G. Dai, J. Xu, Q. Mao, X. Li, J. Liu, Y. Dong, Y. Wang *et al.*, "Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 148–161, 2024.

[4] P. Rostami, M. Sharifi, and M. Dejam, "Shape factor for regular and irregular matrix blocks in fractured porous media," *Petroleum Science*, vol. 17, pp. 136–152, 2020.

[5] M. Asri, D. Malhotra, J. Wang, G. Biros, L. K. John, and A. Gerstlauer, "Hardware accelerator integration tradeoffs for high-performance computing: A case study of gemm acceleration in n-body methods," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 8, pp. 2035–2048, 2021.

[6] A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc *et al.*, "Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.

[7] D. Wu, P. Chen, X. Wang, I. Lyngaas, T. Miyajima, T. Endo, S. Matsuoka, and M. Wahib, "Real-time high-resolution x-ray computed tomography," in *Proceedings of the 38th ACM International Conference on Supercomputing*, 2024, pp. 110–123.

[8] P. Chen, M. Wahib, S. Takizawa, R. Takano, and S. Matsuoka, "Ifdk: A scalable framework for instant high-resolution image reconstruction," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–24.

[9] Z. Xianyi, W. Qian, and Z. Chothia, "Openblas," https://github.com/OpenMathLib/OpenBLAS.

[10] G. Guennebaud, B. Jacob *et al.*, "Eigen," *http://eigen.tuxfamily.org*, vol. 3, 2010.

[11] Intel, "Mkl," https://software.intel.com/en-us/mkl.

[12] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, may 2008. [Online]. Available: https://doi.org/10.1145/1356052.1356053

[13] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "Libxsmm: Accelerating small matrix multiplications by runtime code generation," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 981–991.

[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*, 2021, pp. 1–14.

[15] S. Tang, J. Zhai, H. Wang, L. Jiang, L. Zheng, Z. Yuan, and C. Zhang, "Freetensor: a free-form dsl with holistic optimizations for irregular tensor programs," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 872–887.

[16] C. Rivera, J. Chen, N. Xiong, J. Zhang, S. L. Song, and D. Tao, "Tsm2x: high-performance tall-and-skinny matrix–matrix multiplication on gpus," *Journal of Parallel and Distributed Computing*, vol. 151, pp. 70–85, 2021.

[17] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 58–70.

[18] H. Li, K. Ota, and M. Dong, "Learning iot in edge: Deep learning for the internet of things with edge computing," *IEEE network*, vol. 32, no. 1, pp. 96–101, 2018.

[19] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019.

[20] J.-H. Lee, T. Xiao, and Q. Liu, "A 3-d spectral-element method using mixed-order curl conforming vector basis functions for electromagnetic fields," *IEEE Transactions on Microwave Theory and Techniques*, vol. 54, no. 1, pp. 437–444, 2006.

[21] S. Ausserhofer, O. Biro, and K. Preis, "Discontinuous galerkin finite elements in time domain eddy-current problems," *IEEE Transactions on Magnetics*, vol. 45, no. 3, pp. 1300–1303, 2009.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[23] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788.

[24] J. Devlin, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[26] W. Yang, J. Fang, D. Dong, X. Su, and Z. Wang, "Optimizing full-spectrum matrix multiplications on armv8 multi-core cpus," *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[27] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[28] H. Kung, V. Natesh, and A. Sabot, "Cake: matrix multiplication using constant-bandwidth blocks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[29] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," *ACM SIGOPS Operating Systems Review*, vol. 25, no. Special Issue, pp. 63–74, 1991.

[30] T. M. Smith, R. Van De Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee, "Anatomy of high-performance many-threaded matrix multiplication," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 1049–1059.

[31] 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, jun 2015. [Online]. Available: https://github.com/flame/blis

[32] E. Vasilakis, I. Sourdis, V. Papaefstathiou, A. Psathakis, and M. G. Katevenis, "Modeling energy-performance tradeoffs in arm big. little architectures," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE, 2017, pp. 1–8.

[33] J. Meng, C. Zhuang, P. Chen, M. Wahib, B. Schmidt, X. Wang, H. Lan, D. Wu, M. Deng, Y. Wei *et al.*, "Automatic generation of high-performance convolution kernels on arm cpus for deep learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 2885–2899, 2022.

[34] J. Aycock, "A brief history of just-in-time," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 97–113, 2003.

[35] G. Krylov, G. W. Dueck, K. B. Kent, D. Maier, and I. D'Souza, "Ahead-of-time compilation in omr: overview and first steps," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, 2019, pp. 299–304.

[36] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne *et al.*, "Jax: composable transformations of python+ numpy programs," 2018. [Online]. Available: http://github.com/google/jax

[37] 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," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[38] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[39] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen *et al.*, "Ansor: Generating high-performance tensor programs for deep learning," in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, 2020, pp. 863–879.

[40] Google, "Xla," https://github.com/openxla/xla.

[41] T. Chen, T. He, M. Benesty, V. Khotilovich, Y. Tang, H. Cho, K. Chen *et al.*, "Xgboost: extreme gradient boosting," *R package version 0.4-2*, vol. 1, no. 4, pp. 1–4, 2015.

[42] C.-C. Yang, Y.-R. Chen, H.-H. Liao, Y.-M. Chang, and J.-K. Lee, "Auto-tuning fixed-point precision with tvm on risc-v packed simd extension," *ACM Transactions on Design Automation of Electronic Systems*, vol. 28, no. 3, pp. 1–21, 2023.

[43] J. Mu, M. Wang, L. Li, J. Yang, W. Lin, and W. Zhang, "A history-based auto-tuning framework for fast and high-performance dnn design on gpu," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[44] V. Ferrari, R. Sousa, M. Pereira, J. P. de Carvalho, J. N. Amaral, and G. Araujo, "Improving convolution via cache hierarchy tiling and reduced packing," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2022, pp. 538–539.

[45] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu *et al.*, "The arm scalable vector extension," *IEEE micro*, vol. 37, no. 2, pp. 26–39, 2017.

[46] M. Sato, "The supercomputer "fugaku" and arm-sve enabled a64fx processor for energy-efficiency and sustained application performance," in *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE, 2020, pp. 1–5.

[47] M. Sato, Y. Ishikawa, H. Tomita, Y. Kodama, T. Odajima, M. Tsuji, H. Yashiro, M. Aoki, N. Shida, I. Miyoshi *et al.*, "Co-design for a64fx manycore processor and" fugaku"," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[48] W. Hohl and C. Hinds, *ARM Assembly Language: Fundamentals and Techniques*. Crc Press, 2014.

[49] J. Laukemann, J. Hammer, G. Hager, and G. Wellein, "Automatic throughput and critical path analysis of x86 and arm assembly kernels," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2019, pp. 1–6.

[50] W. Yang, J. Fang, and D. Dong, "Characterizing small-scale matrix multiplications on armv8-based many-core architectures," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 101–110.

[51] J. Li, C. Battaglino, I. Perros, J. Sun, and R. Vuduc, "An input-adaptive and in-place approach to dense tensor-times-matrix multiply," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[52] J. Huang, L. Rice, D. A. Matthews, and R. A. van de Geijn, "Generating families of practical fast matrix multiplication algorithms," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 656–667.

[53] X. Su, X. Liao, and J. Xue, "Automatic generation of fast blas3-gemm: A portable compiler approach," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017, pp. 122–133.

[54] M. Lam, "Software pipelining: An effective scheduling technique for vliw machines," in *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, 1988, pp. 318–328.

[55] C. Eisenbeis, S. Lelait, B. Marmol *et al.*, "The meeting graph: a new model for loop cyclic register allocation." in *PACT*, vol. 95, 1995, pp. 264–267.

[56] L. Wang, J. Xue, and X. Yang, "Reuse-aware modulo scheduling for stream processors," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, 2010, pp. 1112–1117.

[57] F. Inc., "Fujitsu ssl2," https://software.fujitsu.com/jp/manual/manualfiles/m200003/j2ul1903/01enz001/j2ul-1903-01enz0.pdf.

[58] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2818–2826.

[59] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[60] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2017.
[61] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Thirty-first AAAI conference on artificial intelligence*, 2017.
[62] Tencent, "Tnn," https://github.com/Tencent/TNN.
[63] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.
[64] R. Okazaki, T. Tabata, S. Sakashita, K. Kitamura, N. Takagi, H. Sakata, T. Ishibashi, T. Nakamura, and Y. Ajima, "Supercomputer fugaku cpu a64fx realizing high performance, high-density packaging, and low power consumption," *Fujitsu Technical Review*, pp. 2020–03, 2020.