LibShalom: Optimizing Small and Irregular-Shaped Matrix Multiplications on ARMv8 Multi-Cores

Weiling Yang[1,§], Jianbin Fang[1,§], Dezun Dong[1,*], Xing Su[1], Zheng Wang[2]
[1] College of Computer Science, National University of Defense Technology, China
[2] School of Computing, University of Leeds, United Kingdom
{w.yang, j.fang, dong, xingsu}@nudt.edu.cn, z.wang5@leeds.ac.uk

ABSTRACT

General Matrix Multiplication (GEMM) is a key subroutine in high-performance computing. While the mainstream linear algebra libraries can deliver high performance on large and regular-shaped GEMM, they are inadequate for optimizing small and irregular-shaped GEMMs, which are commonly seen in new HPC applications. Some of the recent works in this direction have made promising progress on x86 architectures and GPUs but still leave much room for improvement on emerging HPC hardware built upon the ARMv8 architecture. We present LibShalom, an open-source library for optimizing small and irregular-shaped GEMMs, explicitly targeting the ARMv8 architecture. LibShalom builds upon the classical *Goto* algorithm but tailors it to minimize the expensive memory accessing overhead for data packing and processing small matrices. It uses analytic methods to determine GEMM kernel optimization parameters, enhancing the computation and parallelization efficiency of the GEMM kernels. We evaluate LibShalom by applying it to three ARMv8 multi-core architectures and comparing it against five mainstream linear algebra libraries. Experimental results show that LibShalom can consistently outperform existing solutions across GEMM workloads and hardware architectures.

CCS CONCEPTS

• **Computer systems organization**; • **Software and its engineering** → **Compilers**;

KEYWORDS

Matrix Multiplication, Small and Irregular-Shaped, ARMv8 Multi-Core, Performance Optimization

ACM Reference Format:
Weiling Yang[1,§], Jianbin Fang[1,§], Dezun Dong[1,*], Xing Su[1], Zheng Wang[2]. 2021. LibShalom: Optimizing Small and Irregular-Shaped Matrix Multiplications on ARMv8 Multi-Cores. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3476217

1 INTRODUCTION

General matrix multiplication (GEMM)[1] is a fundamental building block for high-performance computing (HPC) applications - from traditional scientific simulations to emerging deep learning workloads. While GEMM optimization is a heavily studied field, existing linear algebra libraries mainly target GEMM operating on large matrices with regular shapes (i.e., when both dimensions of a matrix are more or less the same) [3, 7, 23, 56, 59].

Due to the diversity and the evolving nature of HPC workloads, the size and shape of the input matrices of a GEMM kernel can vary depending on the application algorithm used and input data. For example, new scientific simulation algorithms in computational fluid dynamics (CFD) like finite element methods and wave equations often adopt GEMM implementations operating on small matrices to achieve scalable performance on modern multi-core systems [28]. For example, the implementation of CP2K [28], a popular molecular dynamics simulator, extensively uses GEMMs performed on matrices of sizes 5×5 and 23×23. As another example, kernels of the Nek5000 high-order solver for CFD heavily rely on GEMMs computing on 8×8 matrices. In addition to these conventional HPC applications, new HPC workloads like deep learning and machine learning methods are often built upon small GEMM kernels [28]. Some of these data analytic algorithms also need to operate on irregular-shaped matrices [14, 32] where the magnitude of both matrix dimensions has a significant difference. For example, GEMMs used by the convolution kernels of the ResNet deep neural network [27] computes on matrices with one dimension equal to 64 while the other is greater than 3000.

These new HPC workload characteristics challenge how we optimize GEMM computation. Although the traditional linear algebra libraries like OpenBLAS [59] and BLIS [56] can deliver near-optimal performance on large and regular-shaped GEMMs, they often give poor performance on small-sized GEMMs. This is an issue reported by recent studies [28] on the x86 architecture and observed in our evaluation on ARMv8 platforms (Section 3). As we will show later in the paper, while OpenBLAS can deliver over 70% of the peak performance on large GEMMs, it gives less than 20% of the peak performance on some representative small and irregular-shaped GEMMs. As small and irregular-shaped GEMMs are now common in HPC, there is a critical need to optimize such workloads.

[§] Equal contribution
[*] Corresponding author
[1] GEMM is a matrix-multiply-accumulate operation, defined as $\mathbf{C} = \alpha\mathbf{A} \cdot \mathbf{B} + \beta\mathbf{C}$, where $\mathbf{A}$ and $\mathbf{B}$ are matrix inputs, α and β are scalar inputs, and $\mathbf{C}$ is a pre-existing matrix which is overwritten by the output. Following the naming convention of linear algebra libraries, in this work, matrix $\mathbf{A}$ is denoted as a $M \times K$ matrix with M rows and K columns, matrix $\mathbf{B}$ is sized of $K \times N$, and $\mathbf{C}$ is sized of $M \times N$.

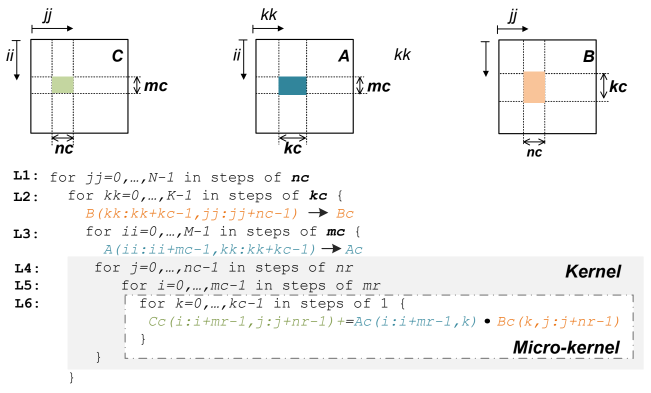
Recently, efforts have been made to optimize small GEMMs [28] on CPUs or irregular-shaped GEMMs on GPUs [12]. BLASFEO was among the first attempts to optimize small and irregular-shaped GEMMs within a single framework [17, 18]. While delivering promising results on x86 and GPU architectures, existing solutions are inadequate for optimizing small and irregular-shaped GEMMs on the ARMv8 based CPU architecture. As we will show in the paper, existing approaches leave much room for performance improvement on ARMv8 multi-cores due to their strategies of data packing (that maps the input matrix elements to a linear buffer), processing edge cases of matrix elements and parallelization. Since multi-core CPUs built upon the ARMv8 architecture and instruction set are quickly emerging as an alternative to the x86-based HPC hardware [38, 44], it is highly attractive to have a library dedicated to optimizing small and irregular-shaped GEMMs on ARVMv8 [52].

This paper presents LibShalom[2], an open-source BLAS library designed to optimize small and irregular-shaped GEMMs on ARMv8 multi-cores. As a departure from existing BLAS libraries, LibShalom takes different approaches for data packing, edge-case processing and parallelization. Like mainstream BLAS libraries, LibShalom builds upon the classical *Goto* GEMM algorithm [22], but it tailors this algorithm for optimizing small and irregular-shaped GEMMs on ARMv8. Unlike existing solutions that process data packing and GEMM computation in a sequential manner, LibShalom leverages the SIMD instruction hide memory latency by carefully overlapping memory accesses incurred by data packing with computation operations within a GEMM kernel. Unlike conventional BLAS libraries that always apply data packing, LibShalom determines, *at runtime*, if data packing is beneficial by taking into consideration the input matrix size and the GEMM computation mode. We show how simple yet effective analytic models can be developed to determine the GEMM loop tiling parameters to enhance instruction scheduling, cache locality and efficiency of parallelization and edge-case processing. We show that our analytical methods, in combination of our new, carefully optimized micro-kernel implementations, lead to significantly better performance over existing BLAS libraries when processing small and irregular-shaped GEMM on ARMv8.

Results. We demonstrate the benefit of LibShalom by applying it to three representative ARMv8 multi-core CPUs, Phytium 2000+ [16], Kunpeng 920 [4] and ThunderX2 [34]. We evaluate LibShalom on both small and irregular-shaped GEMMs as well as computation kernels from real-life applications. We compare it against five GEMM libraries that have an optimizing back-end for ARM architectures [1, 18, 28, 56, 59]. We show that LibShalom consistently outperforms the competing schemes across hardware architectures, GEMM workloads, computation modes for both single-threaded and parallel executions. We showcase that, despite being a library-based approach, LibShalom can outperform techniques built upon just-in-time compilation [28]. The result is a new way for implementing and optimizing kernels for small and irregular-shaped GEMMs.

Contributions. The technical contributions of this paper are:

[2]LibShalom =**Lib**rary for **S**mall Irregular-S**ha**ped M**a**trix Mu**l**tiplicati**o**ns on AR**M**



```
L1: for jj=0,…,N-1 in steps of nc
L2:   for kk=0,…,K-1 in steps of kc {
        B(kk:kk+kc-1,jj:jj+nc-1) → Bc
L3:     for ii=0,…,M-1 in steps of mc {
          A(ii:ii+mc-1,kk:kk+kc-1) → Ac
L4:        for j=0,…,nc-1 in steps of nr
L5:           for i=0,…,mc-1 in steps of mr
L6:             for k=0,…,kc-1 in steps of 1 {
                 Cc(i:i+mr-1,j:j+nr-1)+=Ac(i:i+mr-1,k) • Bc(k,j:j+nr-1)
                }
           }
      }
```

Figure 1: A typical implementation of GEMM.

- It demonstrates how the memory accessing overhead of data packing can be hidden with computations through SIMD instructions and scheduling (Section 4).
- It presents a new way to implement the GEMM computation kernels that achieves better performance over existing solutions (Section 5).
- It shows how analytical methods can be developed to determine the GEMM kernel optimization parameters for the ARMv8 architecture (Sections 4, 5 and 6).

Online material. We provide our code, test programs and full result data sets online at: https://github.com/AnonymousYWL/LibShalom.

2 BACKGROUND

2.1 Problem Scope

Our work focuses on optimizing GEMM performed on small and irregular-shaped matrix inputs on ARMv8 CPUs. We consider a GEMM matrix input to be small if two of its dimensions (M, K, or N) are of a similar size that can fit into the last-level data cache (LLC) of the CPU. By contrast, an irregular-shaped matrix is where one dimension is significantly smaller than the other, e.g., a $64 \times 50,176$ convolutional kernel in a deep neural network [41]. This type of matrices is also known as tall-and-skinny matrices [12, 15]. The dimensions here usually refer to M and N dimensions, and the K dimension is usually not considered [12, 39]. While recent efforts have been made to optimize small and irregular-shaped matrix multiplications, current solutions mainly target x86 architectures or GPUs. It remains unclear how small and irregular-sized GEMMs can be best optimized on emerging ARMv8 multi-core CPUs. Our work aims to close this gap.

2.2 General Matrix Multiply Algorithm

Figure 1 gives a high-level overview of the *Goto* GEMM algorithm [22] used by mainstream linear algebra libraries, including OpenBLAS [59] and BLIS [56]. The algorithm computes $\mathbf{C} = \alpha\mathbf{A} \cdot \mathbf{B} + \beta\mathbf{C}$ by first partitioning and packing matrices A, B, and C into submatrices, so that matrix multiplications can be performed on the submatrices to improve cache locality. The process of partitioning, packing and computing is performed within nested loops outlined in Figure 1, described as follows.

Partitioning. The outermost loop (L1) of Figure 1 groups C and B along the column direction into submatrices of sizes $M \times nc$ and $K \times nc$ respectively. The second level loop (L2) partitions A into submatrices on the column dimension of size $M \times kc$. It also further partitions the $K \times nc$ submatrix of B into row panels of size $kc \times nc$. Essentially, the outermost two loops translate matrix multiplication $\mathbf{A} \cdot \mathbf{B}$ to a panel-to-panel multiplication (GEPP). Then, the third level loop, L3, partitions the $M \times kc$ panels of A into $mc \times kc$ blocks, and partitions a $M \times nc$ submatrix of C into row panels of size $mc \times nc$. The choice of mc and nc is important for maximizing the cache locality after the packing stage, described next.

Data packing. The outermost two loops of the GEMM algorithm packs the $kc \times nc$ panel of B into a linear buffer, Bc. The algorithm will try the largest panel size while the entire Bc can be stored in the last level data cache [47]. Similarly, at loop L3, the algorithm packs submatrices of A generated at this loop level to a linear buffer Ac to fit into the L2 data cache. Data packing is vital for achieving high-performance GEMM by reducing memory access latency through cache locality optimization [26, 33]. However, as we will show later, the existing packing implementation is ill-suited for processing small and irregular-shaped matrices on ARMv8 multi-cores.

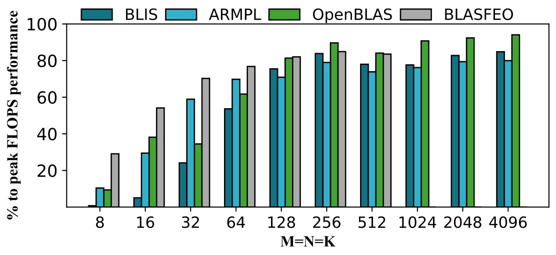
Kernel. Matrix multiplication is performed at the kernel level using a three-level loop, known as general block-times-panel multiply (GEBP) in BLAS. The kernel updates an $mc \times nc$ panel of C by calculating the outer product (via dot to vector multiplications) of a block of Ac of size $mc \times kc$ and a panel Bc of size $kc \times nc$. Specifically, the outermost loop of the kernel (L4) partitions a block Bc into *slivers* (i.e., micro-panels) of $kc \times nr$ and the second-level loop of the kernel (L5) partitions a block Ac into slivers of $mr \times kc$.

Micro-kernel. The innermost loop of the GEBP kernel performs a sequence of updates of an $mr \times nr$ sub-block of C using an $mr \times 1$ column sub-sliver of Ac and a $1 \times nr$ row sub-sliver of Bc. This innermost loop is also known as the micro-kernel in BLIS [56].

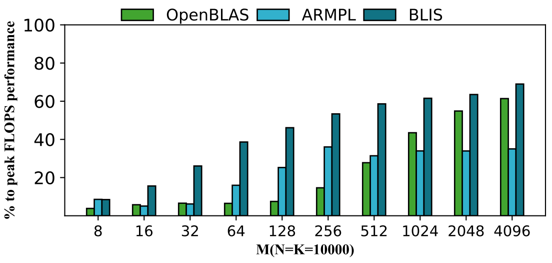
Edge cases. When the matrix size is not a multiple of the micro-kernel size (e.g., M is not a multiple of mr, or N is not a multiple of nr), we have to process the remaining elements outside the partitions. Existing approaches either pad the matrices with zeros to match the kernel size [56], or use another dedicated routine to process the remaining edge elements [59].

3 MOTIVATION AND OVERVIEW

Existing GEMM libraries like OpenBLAS [59] and BLIS [56] are designed to optimize GEMM operating on large matrices. While they can obtain near peak hardware performance on large matrices, they deliver low performance on small and irregular-shaped matrices. A few libraries like BLASFEO [18] offer certain optimizations for small GEMMs but do not take full advantage of a multi-core design and the workload characteristics. As concrete examples, we evaluate the GEMM performance of four representative linear algebra libraries, OpenBLAS, BLIS, ARMPL [1] and BLASFEO, on Phytium 2000+, a 64-core ARMv8 multi-core [11, 16, 54].



(a) Small GEMM on Phytium 2000+



(b) Irregular-shaped GEMM on Phytium 2000+

Figure 2: GEMM performance on (a) small square and (b) irregular-shaped matrices on the ARMv8-based Phytium 2000+ processor. Existing libraries are ineffective in processing small or irregular-shaped GEMMs.

3.1 Motivation Results

In Figure 2, we normalize the measured performances (FLOPS) to the theoretical peak CPU computational performance.

Small-sized GEMM. Figure 2a shows the GEMM performance on square matrices (i.e., $M = N = K$), and we see that existing GEMM libraries also give poor performance when the matrix size is small[3]. For example, even the best-performing library only achieves around 60% of the peak performance when the matrix size is 32. By contrast, they can achieve over 80% of the CPU peak performance when the matrix size is 256 or larger.

Irregular-shaped GEMM. Figure 2b shows the performance of irregular-shaped GEMMs when we fixed N and K to 10,000 while varying M. Matrices of this scale and size are often seen in scientific simulation kernels like high-order FEM codes and sparse direct solvers using super-block [28]. For this type of GEMM matrices, the highly optimized BLIS library can achieve 70% of the peak performance when M is 4096. However, all libraries deliver less than 40% of the peak performance when M is smaller than 128. When $M = 16$ and $N = 50000$, a representative setting for certain neural network workloads [29, 30], all evaluated libraries achieves less than 25% of the theoretical CPU peak performance.

3.2 Optimization Opportunities

As can be seen from the motivation results, there is much room for improvement for small and irregular-shaped GEMM on ARMv8 multi-cores, and none of the test libraries is effective at both small

[3]Since BLASFEO is designed to optimize GEMM on matrices that can entirely fit into the L2 data cache [17, 18], it is excluded from irregular-shaped GEMM in Figure 2b.

and irregular-shaped matrices. After close examinations, we identify three missing opportunities of current linear algebra libraries.

First, although the packing overhead is small ($< 3\%$) on large matrices [45], it can account for 50% of the execution time for small GEMMs (e.g., when $M = 32$ in Figure 2) and cannot be ignored. Existing GEMM libraries always pack data even when it is not beneficial to do so. When the packing overhead outweighs the benefit small-sized GEMMs, existing solutions give a poor performance.

Secondly, we observe around 10% drop in the FLOPS when processing edge cases for small-sized matrices. This performance degradation is observed for all testing GEMM libraries, regardless of which of the two edge case strategies described in Section 2.2 is used. While the overhead of handling edge cases is negligible for large matrices (less than 1% of the execution time when $M = N = K = 5000$), the cost can be significant for small and irregular-shaped matrices.

Thirdly, we found that the existing parallelization scheme for GEMM is ineffective for irregular-shaped matrices. For example, when performing GEMM on matrices of sizes $M = 32, N = K = 10000$, OpenBLAS and BLIS only deliver 6% and 14% of the peak performance on Phytium 2000+. This is because when distributing the work across parallel threads, they ignore the workload characteristics of irregular-shaped GEMMs [43], creating many edge cases to be processed. These edge cases in turn bring in extra overhead that could otherwise be avoided.

3.3 Overview

In light of these observations, our work aims to design a better approach for packing, handling edge cases and parallelization, specifically targeting small and irregular-shaped GEMMs on the ARMv8 architecture for HPC. To this end, we develop LibShalom, an open-source optimizing library for small and irregular-shapped GEMM. Following the common practice of low-level systems libraries, LibShalom provides APIs in C and C++ to be used by the applications, but implements its underlying GEMM kernels in assembly for performance reasons.

GEMM modes. Like most BLAS libraries [1, 17, 56, 59], LibShalom supports four types of GEMM kernels, NN, NT, TN and TT. Here, T and N respectively stand for a transposed and not transposed matrix. For example, GEMM for matrices $\mathbf{A} \cdot \mathbf{B}$ under the NT mode means matrix B is transposed (T) but matrix A is not (N).

Algorithm implementation. Algorithm 1 outlines the LibShalom's GEMM implementation under the NN mode. Like mainstream BLAS libraries, our implementation follows the *Goto* algorithm described in Figure 1, but introduces several optimizations. Firstly, LibShalom removes the always-executed packing steps, i.e., converting matrices B and A to linear buffers Bc and Ac, respectively from Figure 1. For cases that needed to be packed, we perform packing at the micro-kernel level rather than the kernel level. Secondly, we exchange the $L2$ loop and the $L3$ loop from Figure 1 to yield a more contiguous access on matrix A, and use loops $L1$ and $L3$ for parallelization (Section 6). Note that we mainly use the outer-product formulation (scalar-vector multiplication) at the micro-kernel, which has greater computation-to-memory ratio (CMR) than the inner-product formulation (vector-vector multiplication), to update matrix C. Here, the CMR is computed as the ratio of arithmetic instructions to memory load and store instructions (see Section 5.2.1). A larger CMR

Algorithm 1: NN mode GEMM implementation

```
Input: Matrix $A$,$B$, Buffer $Bc$
Output: Matrix $C$
for $jj = 0 \rightarrow N$ step = $nc$ do
    for $ii = 0 \rightarrow M$ step = $mc$ do
        for $kk = 0 \rightarrow K$ step = $kc$ do
            for $j = 0 \rightarrow nc$ step = $nr$ do
                if $size(B) > L1$ then
                    for $k = 0 \rightarrow kc$ step = 1 do
                        $C(ii : ii + mr, jj + j : jj + j + nr) += A(ii : ii + mr; kk + k) \times B(kk + k, jj + j : jj + j + nr)$;
                        $Bc(k, 0 : nr) = B(kk + k, jj + j : jj + j + nr)$
                    for $i = mr \rightarrow mc$ step = $mr$ do
                        for $k = 0 \rightarrow kc$ step = 1 do
                            $C(ii + i : ii + i + mr, jj + j : jj + j + nr) = A(ii + i : ii + i + mr, kk + k) \times Bc(k, 0 : nr)$
                else
                    for $i = 0 \rightarrow mc$ step = $mr$ do
                        for $k = 0 \rightarrow kc$ step = 1 do
                            $C(ii+i : ii+i+mr, jj+j : jj+j+nr) = A(ii+i : ii+i+mr, kk+k) \times B(kk+k, jj+j : jj+j+nr)$
```

indicates that more arithmetic instructions are available to overlap with memory accesses to hide the memory latency.

Roadmap. In the following sections, we present the three key optimizations of LibShalom for minimizing the overhead of small and irregular-shaped GEMMs, by redesigning the kernel (Section 4), micro-kernel (Section 5) and parallelization strategy (Section 6). Without losing generalization, we describe our approach under the NN and NT modes using single-floating point (FP32) operations. However, our optimizations are equally applicable to the other GEMM modes and double-floating points (FP64), which all are supported by LibShalom. We also assume the matrices are stored in the row-major format in our discussions.

4 GEMM KERNEL DESIGN

4.1 Design Principals

For kernel computation $\mathbf{A} \cdot \mathbf{B}$, existing BLAS libraries convert each matrix to a linear buffer at the packing stage, regardless of the mode (N or T) and the matrix size. Our insight is that packing is unnecessary for small matrices or those being sequentially accessed in the micro-kernel because they can be accessed in a cache-friendly manner. For example, as matrix A is accessed rows by rows under the NN mode, the cache prefetching mechanism can largely hide the main memory access latency. For this reason, it is unnecessary to pack a large matrix A and pay the potentially expensive cost of data packing. For scenarios where packing the matrix can be profitable, LibShalom tries to overlap the memory loads and stores incurred by packing with computation instructions inside the micro-kernel (Section 5.3). Therefore, LibShalom only performs packing when ❶ the data cannot be accessed continuously in the micro-kernel (i.e. cache-unfriendly), or ❷ the CMR of the micro-kernel is too low to hide the memory latency without packing.

4.2 NN Mode Packing Strategy

Depending on the size of matrix B, we apply two packing strategies in the NN mode, described as follows.

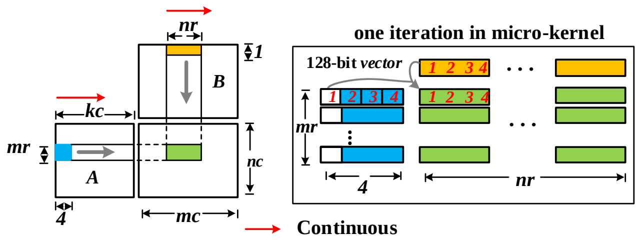


Figure 3: Kernel design using the FP32 NN kernel mode as a working example.

Algorithm 2: Main micro-kernel of all modes

```
// Main micro-kernel of all modes
for $k = 0 \rightarrow kc$ step= 4 do
    $(V0 - V6) \leftarrow A(0:6, k:k+3)$;
    $(V7 - V9) \leftarrow Bc(*, 0:11)$;
    $(V11 - V31) \leftarrow$ FMA $(V0 - V6)[0]$ , $(V7 - V9)$ /* scalar-vector multiply
       */
    ...;
    $(V7 - V9) \leftarrow Bc(* + 3, 0:11)$;
    $(V11 - V31) \leftarrow$ FMA $(V0 - V6)[3]$ , $(V7 - V9)$
```

No packing. If the size of matrix B is smaller than the L1 data cache (i.e., $size(B) < L1$ at line 12 of Algorithm 1), we skip the packing step. Instead, we go straight to divide matrix A into multiple tiles of size $mr \times K$, and then update matrix C at lines 13 - 15 in Algorithm 1.

Packing large B. If matrix B is larger than the L1 data cache capacity, we pack the tiled B into a linear buffer, Bc, and, at the same time, we update parts of matrix C in first distributed loop (lines 6–8 of Algorithm 1). Our algorithm utilizes the fused-multiply-add (FMA) instructions[4] to perform the outer-product computation at line 7 of Algorithm 1. As the FMA instruction can be executed concurrently with independent load and store instructions (thanks to out-of-order instruction scheduling), we use it to hide the packing overhead when packing matrix B at line 8 of Algorithm 1. After packing, Bc is used to update the $mr \rightarrow mc$ rows of C during lines 9 - 11 in Algorithm 1. With a carefully designed micro-kernel (Section 5.3), we ensure that the number of CPU cycles for executing the FMA instructions can hide the overhead for filling Bc. Furthermore, since our kernel reuses Bc across computation iterations for the second distributed loop at lines 9 - 11 in Algorithm 1, we increase the chance for Bc to be kept in the L1 data cache.

Packing choice. For the scenarios where both matrices A and B exceed the L1 data cache size, we will pack B instead of A. Our design choice can be justified using Figure 3. From the diagram, we see that for *any* given row of A, the CPU can continuously access the $0 \rightarrow kc$ elements within the same row. By contrast, the CPU can only do so for the $0 \rightarrow nr$ elements at each row of B. Since our implementation uses a small nr (12 or 6; see Section 5) to promote the use of the vector registers, accessing to a none-packed, large B would exhibit poor cache locality. For this reason, we prioritize the packing of the matrix B at the NN mode. Because accessing matrix A is nearly continuous, we do not pack A even it is the only matrix that is larger than the L1 data cache at the NN mode.

4.3 Other Kernel Modes

In the NT mode, we always pack matrix B because computation is performed on the transposed (T) matrix where elements cannot be accessed along the N dimension with aligned vectorization instruction. This is depicted in Figure 3 where the continuously stored nr elements of B are transposed to be stored at discontinuous memory locations (assuming the row-major storage). The outer-product is ineffective under this setting, because this formulation requires at least one of the M dimension of A and the N dimension of B to be continuously stored in memory. To meet this requirement, LibShalom chooses to pack elements from matrix B to a linear buffer Bc so that matrix elements are stored in continuous memory space. Here we also overlap computation and packing. Similarly, for the TT mode, we pack matrix A as accessing to matrix B is nearly continuous (like how we access matrix A in the NN mode). Like the NN mode kernel, we use the FMA instruction to concurrently update parts of matrix C while preforming data packing.

[4]The FMA instruction computes $a \times b + c$ using one single rounding step.

5 MICRO-KERNEL DESIGN

LibShalom has three types of micro-kernels, designed to minimize memory access latency and edge case processing. The first type of micro-kernels is the main routine for computing $\mathbf{A} \cdot \mathbf{B}$, corresponding to lines 10-11 of Algorithm 1. The second type of micro-kernels is used at the initialization stage to perform packing while updating parts of matrix C. This corresponds to lines 6-7 of Algorithm 1 at the NN mode. The third type of micro-kernels is used to process the edge cases; see Section 2.2.

5.1 Design Principals

Our micro-kernel implementations aim to maximize the CMR, as prior studies have shown that optimizing this metric is important for small and irregular-shaped GEMM to achieve high-performance [28, 30]. We achieve this by taking advantages of the instruction parallelism of GEMM and the vector registers of the ARMv8 architecture, which provides 32 128-bit-wide vector registers (referred to as $V0 - V31$). The key challenge here is to find the right loop tiling parameters, mr and nr, to best utilize the vector registers to maximize the CMR. To this end, we use analytic methods to determine the tiling parameters for the three types of micro-kernels, described in the next subsections. We remark that our distributed micro-kernel design is different from OpenBLAS [59] and BLIS [56] where the packing step and micro-kernel are completely separated like Figure 1.

Working example. In the following subsections, we describe our micro-kernel design using the FP32 NN kernel mode depicted in Figure 3 as a working example. However, our design methodology is equally applied to other kernel modes and FP64 GEMMs.

5.2 Main Micro-kernel

5.2.1 Optimization constraints. For the NN mode micro-kernel, as show in Figure 3, we need mr, nr/j and $mr \times nr/j$ vector registers to store elements from matrices A, B, and C respectively, where j is 4 and 2 for FP32 and FP64 GEMM respectively. In addition, like [47], we reserve one vector register to prefetch the elements of A or B. To make sure that the matrix elements can fit into the number

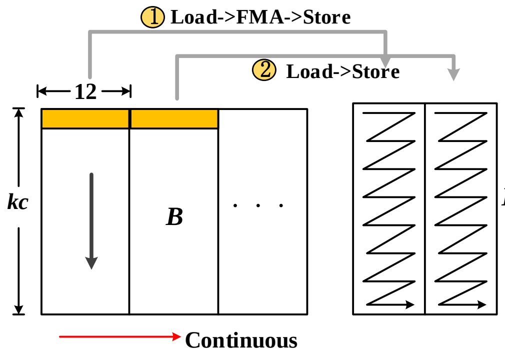


Figure 4: Packing steps of B in FP32 NN mode

of available vector registers (i.e., 31), mr and nr have to satisfy:

$$\begin{cases} mr + \frac{nr}{j} + \frac{mr \times nr}{j} \leqslant (32-1) \\ nr\%j = 0 \end{cases} \tag{1}$$

Since each vector register stores j elements, we wish to set nr to be a multiple of j, i.e., $nr\%j = 0$, so that we do not waste a vector register to store fewer than j elements from matrix B.

5.2.2 *Optimization goal.*

In our NN mode micro-kernel, vector registers store elements of matrix B are released after exiting from the current iteration. By contrast, vector registers for storing matrix A elements will be freed every j iterations after all the j elements on the K direction (e.g., 1, 2, 3, 4 in Figure 3 for FP32 GEMM) have been used. Therefore, for every j iterations, we need mr load instructions to load elements from matrix A. Additionally, we need $nr = nr/j \times j$ loads to fetch elements from matrix B. For computation, we apply the scalar-vector FMA instruction to $mr \times nr$ matrix elements, which translates to $2 \times mr \times nr$ computational operations as each FMA instructions contains two operations, addition and multiplication. Putting it together, the average CMR of our micro-kernels is:

$$CMR = \frac{2 \times mr \times nr}{mr + nr} \tag{2}$$

5.2.3 *Solving the equations.*

To find an integer value of mr and nr that can maximize the CMR, we apply the Lagrange multiplier method [25] to solve the constraints defined in Equation 1 with the goal to maximize the CMR defined in Equation 2. This gives us $mr = 7$ and $nr = 12$ to use in our main micro-kernel implementation for the ARMv8 architecture. Not only NN mode, but we also use micro-kernel of this size for other mode GEMMs. The general process is shown in Algorithm 2.

5.3 Micro-kernel for Packing

The packing micro-kernel (lines 6-8 of Algorithm 1) will only be invoked if the relevant matrix is larger than the L1 data cache.

5.3.1 *Medium-sized matrix.*

If matrix B is larger than the L1 data cache but smaller the LLC, we only need to pack the nr elements of B used in the current iteration of micro-kernel; after that, the elements that are continuous with these nr elements would be prefetched into the data cache.

5.3.2 *Larger and Irregular-shaped matrices.*

We now describe how we pack matrices that cannot fit into the LLC cache.

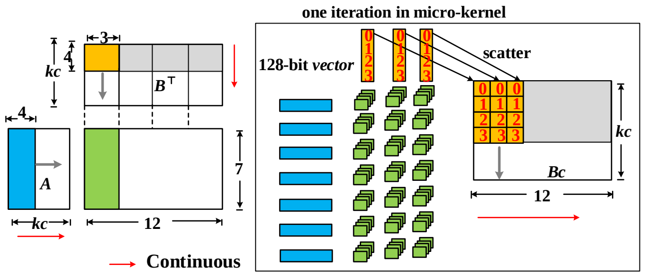


Figure 5: Micro-kernel for FP32 NT mode packing.

NN mode. To reduce cache and TLB misses, when accessing the $0 \rightarrow nr$ elements of B at the current iteration of j loop in Algorithm 1, we pack the next batch of elements of B as required by the next iteration into another part of the linear buffer, Bc in line 8 of Algorithm 1. This is because when these elements are used in the next iteration of the j loop, cache and TLB misses may occur frequently. As we iterate over this micro-kernel, we pack more elements into Bc. As a result, we will have already packed $t \times nr$ of such elements when executing the t^{th} iteration of j loop, where $t = 0, 1, 2...n$. Note that mr and nr in the packing kernel are set to the same values as the main kernel (i.e., $mr = 7$ and $nr = 12$ for FP32). In implementation, we set t to be 0 and 1 for small and irregular-shaped GEMMs, respectively. This means that the former only performs step 1 in Figure 4 in each iteration, while the latter performs steps 1 and 2.

Algorithm 3: Micro-kernel for NT mode data packing

```
for $j = 0 \longrightarrow 12$ step= 3 do
    for $k = 0 \longrightarrow kc$ step= 4 do
        $(V0 - V6) \leftarrow A(0:6, k:k+3)$;
        $(V7 - V9) \leftarrow B(0:2, k:k+3)$;
        $(V10 - V31) \leftarrow$ FMA $(V0 - V6)$ , $(V7 - V9)$ /* vector-vector
            multiply                                                  */
        $(V7 - V9)$ scatter to $Bc(k:k+3, j:j+2)$
    Reduce $(V10 - v31)$ to $V10.[0] - V31.[0]$;
    Store to $C$
```

NT mode. In this case, B is not continuous in the N dimension, which affects the use of the 7×12 main micro-kernel to update the $mr \rightarrow mc$ rows of C. To overcome it, we design a 7×3 packing micro-kernel for NT mode GEMM, as show in Figure 5. In the computation process, we use the inner-product formulation to update C, and the processor accesses A and B along the K dimension. In each iteration, we use seven loads to fetch the elements of A to V0 − 6, and use three loads to fetch the elements of B to V7 − 9. The packing micro-kernel performs 21 (7×3) vector-vector FMAs to produce 84 (4×21) elements, which are stored in V10-31. At the same time, four elements of V7-V9 are scattered to Bc, and the distance between the elements is 12. Additionally, the elements in the same position of vectors are scattered to adjacent positions. For example, in Figure 5, the distance between 0 and 1 in the same vector is 12 elements, but 0 in different vectors are next to each other in Bc. At the end of the micro-kernel, the four elements of V10-31 need to be reduced to one as using vector-vector FMAs. To get a complete Bc, we need to call packing micro-kernel four times (12/3). The micro-kernel uses the same $7 \times kc$ tiled matrix A, but uses different $3 \times kc$ tiled matrix B.

```
ldp     s12, s13, [pB], #8
ldp     s14, s15, [pB], #8

ldr     q4, [pA], #16
ldr     q5, [pA], #16

fmla    v16.4s, v4.4s, v12.s[0]
fmla    v17.4s, v5.4s, v12.s[0]
fmla    v20.4s, v4.4s, v13.s[0]
fmla    v21.4s, v5.4s, v13.s[0]
fmla    v24.4s, v4.4s, v14.s[0]
fmla    v25.4s, v5.4s, v14.s[0]
fmla    v28.4s, v4.4s, v15.s[0]
fmla    v29.4s, v5.4s, v15.s[0]
```

(a)

after schedule

```
ldr     q4, [pA], #16
ldr     q5, [PA],  #16
...
ldr     q0, [PB], #16
...
fmla    v16.4s, v4.4s, v0.s[0]
fmla    v17.4s, v5.4s, v0.s[0]
ldr     q6, [pA], #16
fmla    v20.4s, v4.4s, v0.s[1]
fmla    v21.4s, v5.4s, v0.s[1]
ldr     q7, [PA],  #16
fmla    v24.4s, v4.4s, v0.s[2]
fmla    v25.4s, v5.4s, v0.s[2]
ldr     q1, [PB], #16
fmla    v28.4s, v4.4s, v0.s[3]
fmla    v29.4s, v5.4s, v0.s[3]
```

(b)

Figure 6: OpenBLAS' 8×4 edge-case processing micro-kernel (a) and a better instruction schedule used by LibShalom (b).

The storage format of *Bc* is the same as that of Figure 4. The general process of packing micro-kernel is shown in Algorithm 3, where the vector-vector FMAs and scatter instructions occur interchangeably.

TN and TT modes. Following the discussion in Section 4.3, for TN mode, we apply the same strategy used for the NT mode to pack matrix *A*. Similarly, for TT mode, we apply the strategy used for the NN model to pack matrix A, depending on its size.

We want to highlight that our implementation interleaves the memory load and store instructions required in the packing step with FMA computation instructions like Figure 6. This is the key difference between LibShalom and all existing GEMM implementations (like OpenBLAS and BLASFEO), where the packing and micro-kernel routines are executed in a sequential order. As we show later in the paper, by overlapping packing with the computation instructions, LibShalom gives significantly better performance for small and irregular-shaped GEMMs.

5.4 Edge Processing Micro-kernel

Our edge-case processing kernel adapts the OpenBLAS implementation [59], but enhances it with better instruction scheduling designed for small and irregular-sized GEMM. Considering the OpenBLAS 8×4 edge micro-kernel for ARMv8 architectures shown in Figure 6a, this implementation has two drawbacks on the ARMv8 architecture. Firstly, it fails to hide the memory latency with computation instructions. That is, the load instructions are scheduled in a batch fashion. Secondly, there is no sufficiently large instruction distance between two dependent instructions. As illustrated in Figure 6b, our implementation overcomes these two drawbacks by prefetching the matrix elements required by the current iteration in the previous one and insert the load instructions between FMA instructions to hide the latency. Our experimental results show that this strategy significantly improves the OpenBLAS implementation.

5.5 Other Hardware Architectures

Our approach is generally applicable and can be easily ported to other architectures. All our discussions so far target the 128-bit vector register supported by our evaluation platforms. Some new ARM-based many-cores, like the FUJITSU ARMv8-based A64FX [40] and future ARMv9 processors (e.g., NVIDIA Grace) support the latest ARM Scalable Vector Extension (SVE) [2]. This extension allows the CPU implementation to choose a vector length that is any multiple of 128 bits between 128 and 2048 bits. Our approach can be applied to a longer vector length with an revised *mr* and *nr* computed according to the available number and length of vector registers. In addition to ARM-based CPUs, our techniques can also be ported to modern x86 architectures with vectorization extensions and FMA-like instructions. Doing so will require changing the constraints of Equation 1 to match the hardware parameters to derive *mr* and *nr*. Furthermore, to adapt to different cache sizes, we can adjust the values of *mc*, *nc* and *kc* [33]. Other than these parameter adjustments, we believe our analytical methods and instruction scheduling optimizations can remain unchanged.

6 PARALLELIZATION METHODS

Small-sized GEMM is typically executed with a single thread, but irregular-shaped GEMM can benefit from parallel execution. LibShalom applies a static work partitioning scheme to parallelize irregular-sized GEMM by using the *fork-join* operating system primitives. By default, we use all available cores of the CPU. For a CPU with T cores, we will spawn T parallel threads.

6.1 Work Partitioning

To ensure work balance among parallel threads, LibShalom adopts a two-level parallelization strategy. It first divides matrix C into a grid of sub-blocks, where each thread updates one of the sub-blocks. Since we partition the work across T parallel threads, each parallel thread will perform $\frac{M\times K}{T_m} \times \frac{N\times K}{T_n}$ computation operations for $\mathbf{A} \cdot \mathbf{B}$. Similarly, the number of memory accesses required by each parallel thread is $\frac{M\times K}{T_m} + \frac{N\times K}{T_n}$, where T_m and T_n are the number of threads (or cores) assigned to the M and N dimensions respectively, where $T_m \times T_n = T$. Therefore, the CMR for updating a sub-block is:

$$CMR = \frac{M \times N}{M \times T_n + N \times \frac{T}{T_n}} \tag{3}$$

Like our main micro-kernel design (Section 5.2), we wish to maximize the CMR. By applying the inequality of arithmetic and geometric mean method, we have:

$$CMR \le \frac{M \times N}{2 \times \sqrt{T \times M \times N}} \tag{4}$$

where both sides of the equation will equal if $M \times T_n = \frac{N\times T}{T_n}$. In other words, when $T_n = \sqrt{\frac{T\times N}{M}}$, CMR would reach its maximum value. By taking into consideration the overhead of the packing micro-kernel, we take the up-bound value of T_n, i.e., $T_n = \lceil\sqrt{\frac{T\times N}{M}}\rceil$, to maximize the CMR. We note that $T \bmod T_n = 0$ to ensure the number of cores can be equally divided among parallel threads. For example, for parallelzing GEMM with $M = 2048$ and $N = 256$ on a 64-core processor, we would set $T_n = 4$, which leaves us with $T_m = 16$ (as $T_m \times T_n = T$). To minimize the thread synchronization overhead, we choose to parallelize two outer loops of the GEMM kernel (i.e., $L1$ and $L3$ in Figure 1) , instead of the inner loops [19].

Table 1: Hardware evaluation platform

	Phytium 2000+	KP920	ThunderX2
Peak perf. (FP32 GFLOPS)	1126.4	2662.4	1280
Number of Cores	64	64	32
Frequency	2.2 GHz	2.6 GHz	2.5 GHz
L1 cache	32KB	64KB	32KB
L2 cache	2MB	512KB	256 KB
L3 cache	None	64MB	32MB
RAM	64 GB	64 GB	64 GB

7 EXPERIMENTAL SETUP

7.1 Evaluation Platforms

Hardware. We evaluate LibShalom on three representative ARMv8 multi-core architectures: Phytium 2000+ [16], Kunpeng 920 (KP920) [4] and ThunderX2 [34]. Table 1 lists the specification of the hardware platforms used in our evaluation. Note that on Phytium 2000+, the L2 cache is shared between a cluster of four cores, while on KP920 and ThunderX2, the L2 cache is private to a processor core.

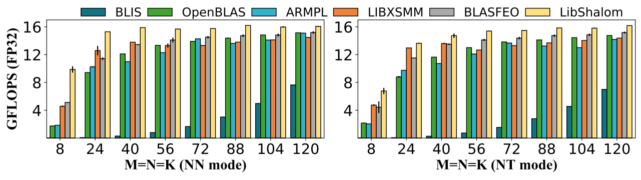
Systems software. Our evaluation platforms run Linux kernel version 4.19.46. We compile the benchmarks using gcc version 8.2.1 with the "-O3" compiler option. LibShalom uses OpenMP to parallelize irregular-shaped GEMMs.

7.2 Workloads

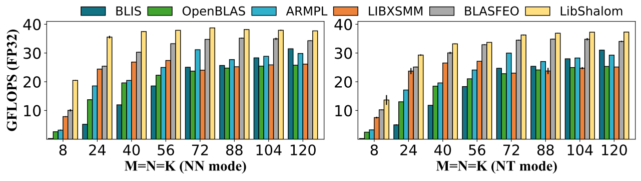
We evaluate LibShalom by applying it to both small and irregular-sized matrices. The size ($M \times N \times K$) of the small matrices ranges from $8 \times 8 \times 8$ to $120 \times 120 \times 120$, which are the typical matrix sizes seen in applications like scientific simulation workloads like SeisSol [8] and Nekbox [6]. The M or N of the irregular-sized matrices used in our evaluation ranges from 32 to 256. These types of irregular-sized matrices are commonly seen in convolution neural networks (CNN) [37, 41]. Like prior work [29], we initialize the matrices by populating them with random floating-point numbers (0 to 1). In addition to the synthetic matrix inputs, we also apply LibShalom to the computational kernels from CP2K (an open-source molecular dynamics simulator) [28] and the VGG CNN [30]. We report the results for running GEMM under the NN and NT modes, but we also observe similar performance trends under the TN and TT modes. More experimental results can be found on the project website [5].

7.3 Competitive Approaches

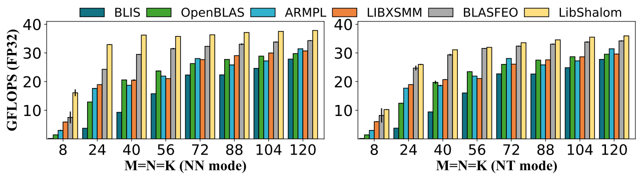
We compare LibShalom against five GEMM libraries that have a back-end specifically tuned for ARMv8. These include OpenBLAS [59], BLIS [56] and ARMPL [1], which are designed to optimize large GEMM, as well as LIBXSMM [28] and BLASFEO [18], which specifically target small-matrix GEMM. Note that LIBXSMM uses just-in-time (JIT) compilation to optimize the GEMM kernel on the underlying architecture and uses a code cache to minimize the compilation overhead across different runs of the same kernel. Unless stated otherwise, we always run LIBXSMM on the target GEMM kernel to warm up its code cache so that the JIT compilation overhead is not included in its execution time measurement.



(a) Phytium 2000+



(b) KP920



(c) ThunderX2

Figure 7: Small GEMMs on our evaluation platforms.

We also note that ARMPL is the official ARM performance library, which is heavily optimized for BLAS by ARM.

7.4 Evaluation Methodology

For small-sized GEMM, we measure the single-threaded performance because the small matrix size does not benefit from parallel CPU execution. This is standard practice when processing small-sized matrices where parallelism is achieved by running multiple GEMM kernels to process independent matrices. For irregular-sized matrices, we report the multi-threaded performance using all the cores of a CPU. Note that because BLASFEO does not support multi-threaded execution, it is excluded from the irregular-sized matrix experiments to ensure fairness.

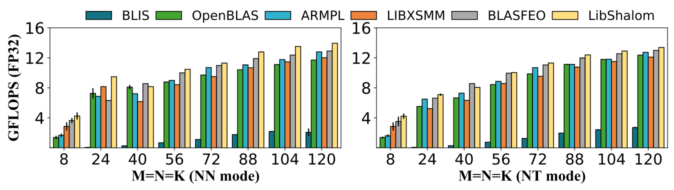
Performance report. We run each GEMM kernel 10 times and report the *geometric mean* of the runtime. We show the variations across different runs as a min-max bar.

8 EXPERIMENTAL RESULTS

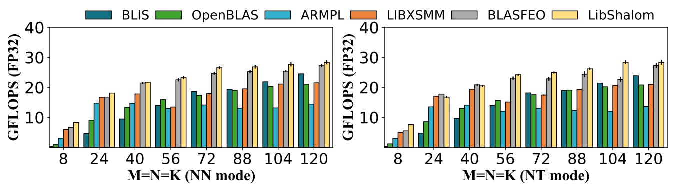
8.1 Single-threaded Small GEMM

In this experiment, we show the FP32 throughput for running small GEMMs on the NN and NT modes. We also observe similar trends for TN and TT modes. We also note that we obtain similar performance when applying LibShalom to double-precision workloads, where the throughput is roughly half of the FP32 performance across all test methods.

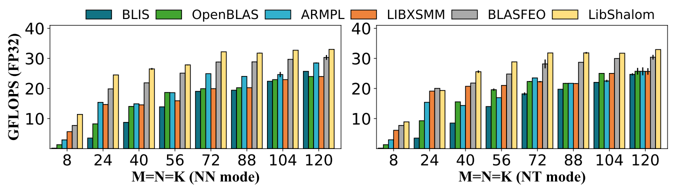
Figure 7 shows the GEMM performance by first warming up the cache - a typical scenario where the *small* matrices data has been preloaded into a certain level cache before launching the GEMM kernel. This is the evaluation methodology adopted by



(a) Phytium 2000+



(b) KP920



(c) ThunderX2

Figure 8: Small GEMMs starting with a cold cache.

the source publications of LIBXSMM [28] and BLASFEO [18]. In this scenario, LibShalom consistently outperforms the competing methods across benchmarks and evaluation platforms. The advantage of LibShalom is noticeable on smaller matrices. For example, when $M = N = K = 8$, LibShalom delivers 2× higher throughput than BLASFEO, the best-performing alternative approach. We note that this GEMM kernel size is widely used in scientific simulation algorithms, including the NekBox CFD solver [28]. When the matrix size increases to 120, LibShalom still gives at least 5% (up to 10%) higher throughput compared to the alternative approach. This benefit mainly comes from the optimized micro-kernels used by LibShalom. We also observe that LibShalom gives higher performance for GEMM running on the NN mode than that of the NT mode, especially on smaller sized matrices. This is because, unlike in the NT mode, NN mode GEMM under LibShalom does not pack matrices that can fit into the L1 data cache; see Section 5. Overall, LibShalom gives the highest throughput across the matrix settings by giving 1.05 ∼ 2× higher throughputs across hardware platforms and manifests more significant advantages on smaller matrices.

Figure 8 shows the results when the GEMM kernel was launched from a cold cache where the matrix data are not presented in the data cache. In this evaluation scenario, LibShalom demonstrates a similar performance trend as Figure 7, outperforming alternative schemes on most of the test cases. On a few matrix sizes, LibShalom does not give noticeable advantages over BLASFEO - the best-performing alternative method. These matrix sizes are a (or nearly) multiple of BLASFEO's 8×8 micro-kernel; as such, there is no or little edge-case processing overhead incurred by BLASFEO, where our edge-case optimization does not demonstrate a benefit. Nonetheless, LibShalom delivers the highest overall throughout and outperforms other schemes for most of the matrix settings.

8.2 Parallelized Irregular-shaped GEMM

Figure 9 shows the results on irregular-shaped GEMM using all the CPU cores for parallelization on Phytium 2000+. Due to the space constraint, we show the results under the NT mode, but we observe similar performance trends in other modes. Like prior work [42], we set K to a sufficiently large number (5000 in our evaluation) to drive the last run data out of the last level data cache to avoid the artificially good performance due to a hot data cache across multiple runs. Note that we omit the results of LIBXSMM and BLASFEO in this experiment as they are tuned for small GEMMs and give a poor performance on irregular-shaped GEMMs.

LibShalom significantly outperforms the alternative approaches across our evaluation platforms, yielding on average 1.8× performance improvement over the second-best performing method, BLIS. The performance benefit of LibShalom tends to be more significant for smaller matrix sizes (i.e., when M or N are smaller). For example, in Figure 9, for GEMMs with $M = 32$, LibShalom gives 2.6x higher GFLOPS over BLIS. This is largely due to the more efficient packing strategy adopted by LibShalom when processing small matrices. LibShalom also demonstrates better performance over OpenBLAS and ARMPL because its parallelization strategy can minimize the overhead of processing edge cases. Although ARMPL is an official BLAS library developed by ARM for parallel GEMMs, it delivers lower performance compared to LibShalom. Once again, LibShalom's advantage is greater when M and N are small, suggesting that LibShalom is highly effective in handling irregular-shaped matrices.

Figure 10 reports irregular-shaped GEMM performance on KP920 and ThunderX2 under NN and NT mode. Compared with the best-performing baseline, LibShalom improves the performance by 1.6× and 1.3× respectively, on average on KP920 and ThunderX2. We also observe that LibShalom gives higher performance for irregular-shaped GEMM running on the NT mode than that of the NN mode. This is because, unlike in the NT mode, the elements of B cannot be continuously accessed along K dimension under NN mode. Overall, LibShalom is generally applicable and can deliver portable performance across representative ARMv8 processors.

8.3 Scalability

Figure 11 shows the scalability when performing an irregular-shaped GEMM kernel of $\{M \times N \times K\}$=$\{64 \times 50176 \times 576\}$ from the VGG convolutional neural network [29]. The results are normalized to the performance obtained by the single-threaded OpenBLAS execution. As can be seen from the diagram, LibShalom not only outperforms other approaches but also exhibits the best scalability as the number of threads used increases. And the maximum speedup is 49× for Phytium 2000+, 82× for KP920, and 35× for ThunderX2.

8.4 L2 Data Cache Locality

In this experiment, we measure the L2 data cache miss count using the hardware performance counter. The experiment was performed on an irregular-shaped, NT mode GEMM with input matrix sizes of $M = 64, N = 50176$, where K ranging from 576 to 3744, with a step of 128. The setting ensures that the data required by the GEMM kernel can fit into the L2 data cache in an ideal scenario.

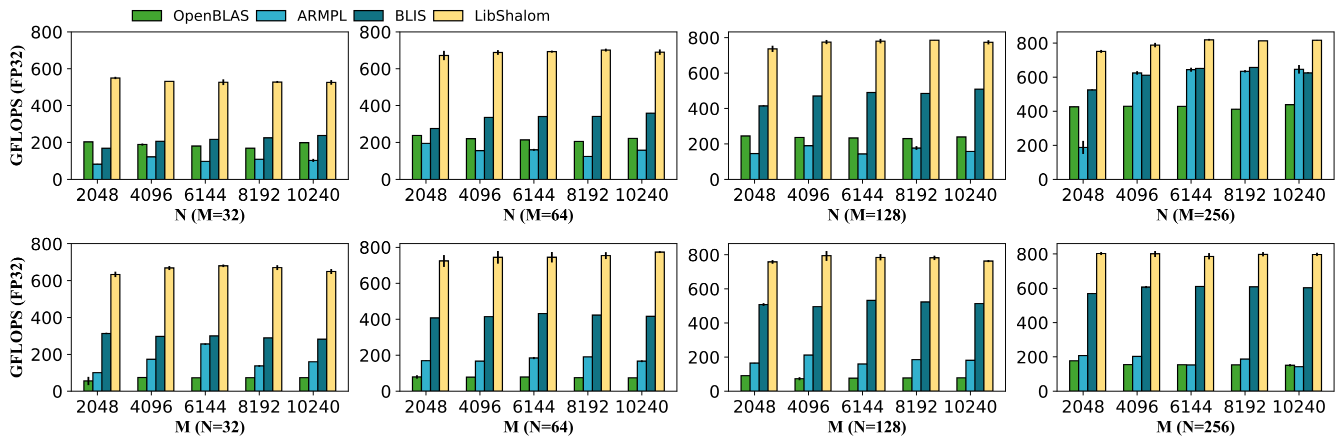


Figure 9: Performance of irregular-shaped GEMM on Phytium 2000+ under the NT mode ($K = 5000$).

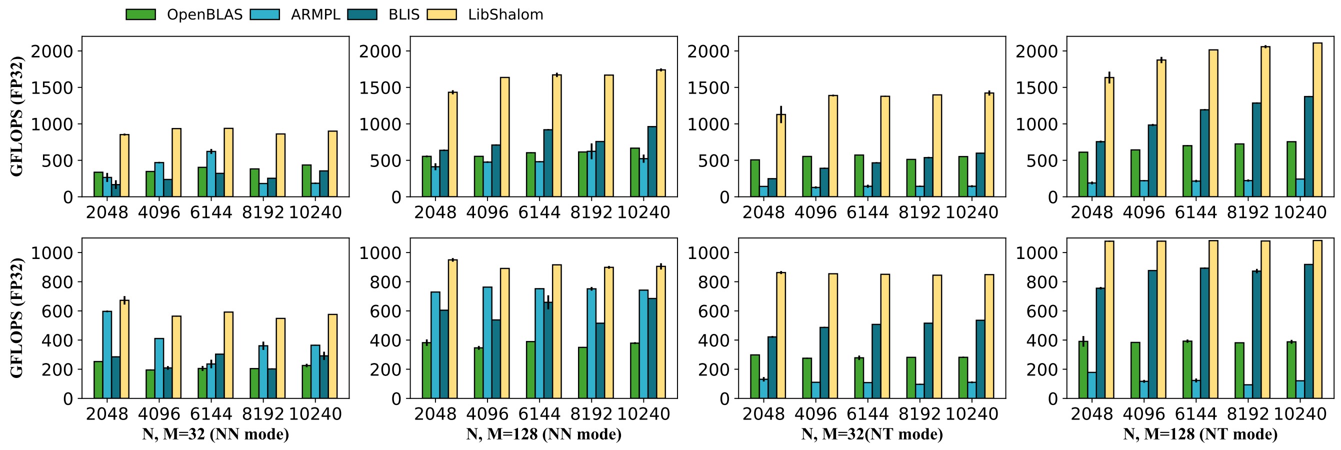


Figure 10: Irregular-shaped GEMMs on KP920 (top row) and ThunderX2 (bottom row) under NN and NT mode ($K = 5000$).

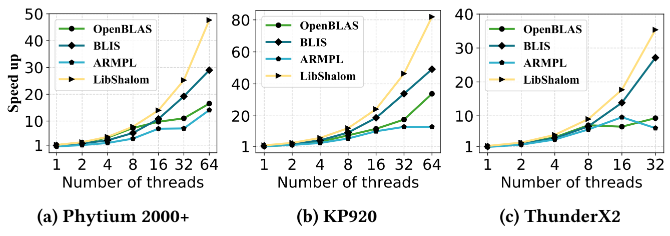


Figure 11: Scalability on the VGG irregular-shaped GEMM.

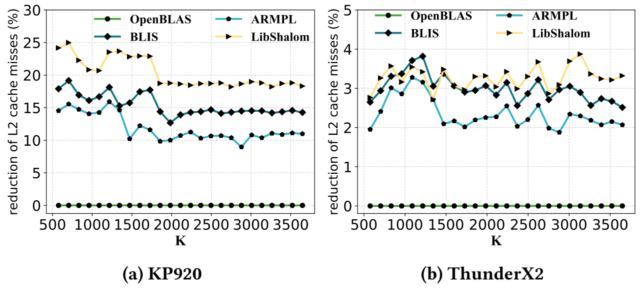


Figure 12: Reduction of the L2 data cache misses over OpenBLAS for irregular-shaped NT mode GEMMs.

Hence, a good GEMM routine should have low L2 data cache misses. The results are shown on KP920 and ThunderX2, because we can access the performance counter through the Linux `perf` profiler on these two platforms. Figure 12 shows the reduction of L2 cache misses using the OpenBLAS measurement as the baseline - the higher the reduction is, the better L2 cache locality an approach has. LibShalom experiences the least frequent cache misses across all matrix sizes, representing around 20% reduction in the cache misses on KP920. This is because LibShalom chooses to not pack matrix A under the NT mode; instead, it exchanges loops $L2$ and $L3$ in Figure 1 to improve the locality when accessing matrix A within the GEMM kernel. By eliminating the memory loads and stores introduced by data packing, LibShalom improves the computation kernel's cache locality, leading to less frequent cache misses.

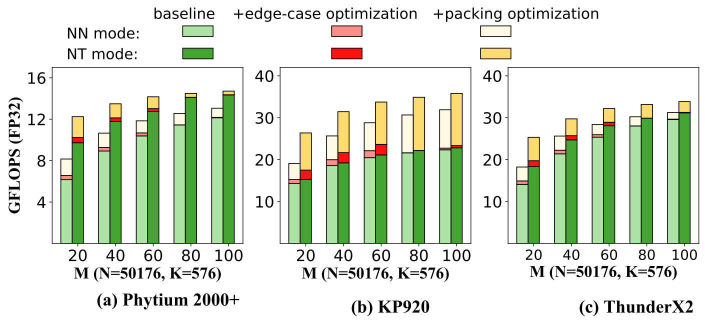


Figure 13: Breakdown of Optimizations on single-threaded irregular-shaped GEMM on three platforms.

8.5 Breakdown of Optimization Techniques

In this experiment, we measure how the proposed packing and edge-case-processing optimizations contribute to performance improvement. Here, we use OpenBLAS as a baseline to show the speedup contribution brought by each of the two optimization techniques. The experiment was performed on single-threaded irregular-shaped, NT mode GEMM. In the experiment, we fix N and K to the VGG CNN kernel size of 50,176 and 576, but we vary M from 20 to 100 with a step of 20.

As can be seen from Figure 13, our data packing optimization can have a significant contribution to performance improvement because this technique can overlap the expensive memory accesses with computation through FMA instructions. Our optimizations also demonstrate various degrees of benefits on different architectures. When $M = 20$, our two optimizations give a 1.25x and 1.6x improvement on Phytium 2000+ and KP920 respectively. The reason for the more noticeable advantage on KP920 over Phytium 2000+ is described as follows. KP920 runs a higher clock frequency over Phytium 2000+ (2.6 GHz vs 2.2 GHz), and it has more FMA units, allowing KP920 to execute more arithmetic instructions per time unit. In other words, KP920 requires more intensive FMA instructions than Phytium 2000+ to keep FMA units busy. It is more difficult for OpenBLAS' implementation strategy to achieve a good CMR to hide the memory latency on a faster CPU where our approach gives stronger benefit.

8.6 Evaluation on Application Kernels

In this evaluation, we apply the tested methods to the GEMM kernels extracted from real-life application workloads. In the first experiment, we apply each approach to the FP64 small GEMM kernels from the CP2K simulation package [9]. The matrix sizes involved range between $4-32$ [28]. As can be seen from Figure 14, LibShalom gives the best performance across matrix sizes and evaluation platforms. Once again, LibShalom demonstrates noticeable advantages when the input matrices are small. For example, it gives up to 2× improvement over LIBXSMM for the input matrix sizes ($M \times N \times K$) is $5 \times 5 \times 5$. Considering LIBXSMM uses a just-in-time compilation back-end to aggressively optimize the GEMM computation, this is an impressive improvement obtained by LibShalom as a library-based solution.

In the second experiment, we evaluate the irregular-shaped GEMM performance using the typical FP32 convolutional kernels from the widely-used VGG16 image classification network [29, 30]. Like prior work [29], we consider five convolution layers from VGG16, namely conv1.2, conv2.2, conv3.3, conv4.2 and conv5.2 of VGG16. These kernels perform GEMM on matrice sizes of $M = \{64, 128, 256, 512, 512\} \times N = \{50176, 12544, 3136, 784, 196\} \times K = \{576, 1152, 2304, 4608, 4608\}$, where the N dimension of the matrices are significantly larger than M. In this experiment, we use all the CPU cores to execute a GEMM kernel. The results are given in Figure 15. Once again, LibShalom consistently outperforms all alternative approaches across GEMM kernels and evaluation platforms. LibShalom's advantage is significant for certain kernels, like conv1.2 and conv5.2, where it improves the second-best-performing approach by up to 1.6x.

9 RELATED WORK

High-performance linear algebra libraries are a vital component of the HPC systems software stack. A range of linear algebra or so-called BLAS libraries were developed to optimize the execution of linear algebra kernels, including GEMMs [1, 3, 56, 59]. Most of these BLAS libraries are designed to optimize GEMM operating on large and regular-shaped matrices.

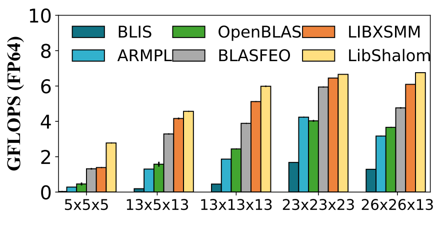
Recent studies report that many new HPC workloads use small GEMM to exploit fine-grained parallelism for better scalability [17, 28]. Other works also highlight the importance of optimizing irregular-shaped GEMMs seen in machine learning workloads [29, 36]. Recent works target optimizing small GEMMs on x86 [28] or irregular-shaped GEMM on GPUs [12, 39]. Some of these techniques have been integrated into deep learning frameworks [20, 21]. However, as we have shown in the paper and our prior work [52], existing approaches give a sub-optimal performance on the ARMv8 architecture, leaving much room for improvement.

BLASFEO is designed to optimize both small and irregular-shaped GEMMs. It provides two optimization routines [17, 18]. The first covert the input matrices to the panel-major format to improve the cache locality. The second selectively packs the input matrices based on some pre-defined heuristics. For example, it does not pack a small matrix A. Likes existing BLAS libraries, BLASFEO performs data packing and computation sequentially. It also does not support parallel execution of irregular-shaped GEMMs. LibShalom overcomes these limits by overlapping the memory access instructions introduced by data packing with computations and provides a highly optimized kernel for parallel execution.

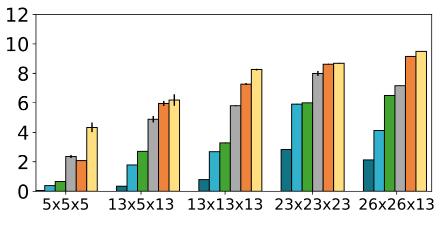
LIBXSMM uses JIT code compilation technology to generate assembly code for small GEMMs [28]. This technique allows aggressive instruction-level optimization. LIBXSMM uses code cache to reuse the compilation results to reduce the overhead of JIT. However, it is designed to optimize tiny GEMM kernels where $(MNK)^{1/3} <= 64$. We empirically show that LIBXSMM is ineffective for optimizing the commonly used small GEMMs where the matrix sizes do not fit its design scope. Our experimental results show that despite being a library-based approach, LibShalom is able to outperform LIBXSMM across GEMM workloads and evaluation platforms.

10 CONCLUSIONS AND FUTURE WORK

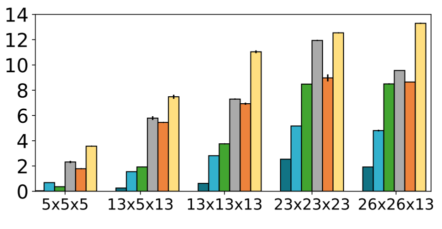
We have presented LibShalom, an open-source library for optimizing small and irregular-shaped GEMMs on ARMv8 multi-cores.



(a) Phytium 2000+

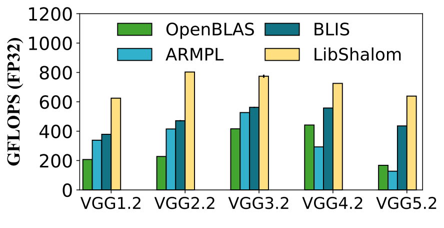


(b) KP920

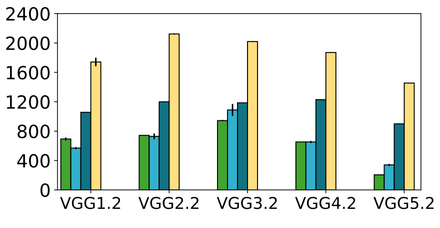


(c) ThunderX2

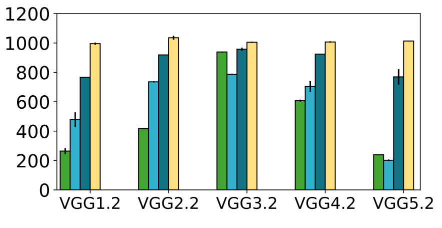
Figure 14: Small FP64 GEMM performance on CP2K kernels.



(a) Phytium 2000+



(b) KP920



(c) ThunderX2

Figure 15: Performance of FP32 irregular-shaped GEMM kernels from the VGG convolutional neural network.

LibShalom determines if data packing is beneficial, and when packing is deemed necessary, it uses the FMA SIMD extension to hide the non-trivial data packing overhead through instruction scheduling. We show how simple analytical models can be developed to derive the tuning parameters of a GEMM kernel. We evaluate LibShalom by applying it to small and irregular-shaped GEMMs on three ARMv8 multi-core architectures and compare it against five mainstream BLAS libraries. Experimental results show that LibShalom delivers consistently better overall performance across all hardware evaluation platforms.

In the future, we will look into how to extend our optimization techniques to sparse matrix computation [10, 11, 31, 55]. Another interesting direction is to open up the kernel parameters to allow an auto-tuning framework to search for the optimal parameters for optimizations like instruction scheduling and compiler options [13, 24, 35, 46, 48–51, 53, 57, 58]. Our future work will consider integrating our techniques with a performance tuning framework to fine-tune the kernel parameters.

ACKNOWLEDGMENTS

We thank Xiaoyu Wang and Yanghai Wang for their helpful discussions. This work was partially supported by the National Key R&D program of China under grant agreement 2018YFB0204300, the Excellent Youth foundation of Hunan Province under grant number 2021JJ10050, the National Natural Science Foundation of China (NSFC) under grant agreements 61972408 and 61872294, and a UK Royal Society International Collaboration grant.

REFERENCES

[1] [n. d.]. ARM PERFORMANCE LIBRARIES. ([n. d.]). https://www.arm.com/products/development-tools/server-and-hpc/allinea-studio/performance-libraries.
[2] [n. d.]. ARMv9. ([n. d.]). https://www.arm.com/company/news/2021/03/arms-answer-to-the-future-of-ai-armv9-architecture.
[3] [n. d.]. Intel MKL. ([n. d.]). https://software.intel.com/en-us/mkl.
[4] [n. d.]. Kunpeng 920. ([n. d.]). https://www.hisilicon.com/en/products/Kunpeng/Huawei%20Kunpeng%20920.
[5] [n. d.]. LibShalom. ([n. d.]). https://github.com/AnonymousYWL/MYLIB.
[6] [n. d.]. Nek5000/NekBox. ([n. d.]). https://github.com/NekBox/NekBox.
[7] [n. d.]. OpenCL BLAS. ([n. d.]). https://github.com/clMathLibraries/clBLAS.
[8] [n. d.]. A scientific software for the numerical simulation of seismic wave phenomena and earthquake dynamics. ([n. d.]). http://www.seissol.org/.
[9] Mauro Calderara, Sascha Brück, Andreas Pedersen, Mohammad H. Bani-Hashemian, Joost VandeVondele, and Mathieu Luisier. 2015. Pushing back the limit of *ab-initio* quantum transport simulations on hybrid supercomputers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015.* ACM, 3:1–3:12.
[10] Donglin Chen, Jianbin Fang, Shizhao Chen, Chuanfu Xu, and Zheng Wang. 2019. Optimizing sparse matrix–vector multiplications on an armv8-based many-core architecture. *International Journal of Parallel Programming* 47, 3 (2019), 418–432.
[11] Donglin Chen, Jianbin Fang, Chuanfu Xu, Shizhao Chen, and Zheng Wang. 2020. Characterizing Scalability of Sparse Matrix-Vector Multiplications on Phytium FT-2000+. *Int. J. Parallel Program.* 48, 1 (2020), 80–97.
[12] Jieyang Chen, Nan Xiong, Xin Liang, Dingwen Tao, Sihuan Li, Kaiming Ouyang, Kai Zhao, Nathan DeBardeleben, Qiang Guan, and Zizhong Chen. 2019. TSM2: optimizing tall-and-skinny matrix-matrix multiplication on GPUs. In *Proceedings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, June 26-28, 2019.* ACM, 106–116.
[13] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT).* IEEE, 219–232.
[14] Marat Dukhan. 2019. The Indirect Convolution Algorithm. *CoRR* abs/1907.02129 (2019).
[15] Dominik Ernst, Georg Hager, Jonas Thies, and Gerhard Wellein. 2021. Performance engineering for real and complex tall & skinny matrix multiplication kernels on GPUs. *The International Journal of High Performance Computing Applications* 35, 1 (2021), 5–19.
[16] Jianbin Fang, Xiangke Liao, Chun Huang, and Dezun Dong. 2021. Performance Evaluation of Memory-Centric ARMv8 Many-Core Architectures: A Case Study with Phytium 2000+. *J. Comput. Sci. Technol.* 36, 1 (2021), 33–43.
[17] Gianluca Frison, Dimitris Kouzoupis, Tommaso Sartor, Andrea Zanelli, and Moritz Diehl. 2018. BLASFEO: Basic Linear Algebra Subroutines for Embedded Optimization. *ACM Trans. Math. Softw.* 44, 4 (2018), 42:1–42:30.

[18] Gianluca Frison, Tommaso Sartor, Andrea Zanelli, and Moritz Diehl. 2020. The BLAS API of BLASFEO: Optimizing Performance for Small Matrices. *ACM Trans. Math. Softw.* 46, 2 (2020), 15:1–15:36.

[19] Wanrong Gao, Jianbin Fang, Chun Huang, Chuanfu Xu, and Zheng Wang. 2021. Optimizing Barrier Synchronization on ARMv8 Many-Core Architectures. In *2021 IEEE International Conference on Cluster Computing (Cluster).*

[20] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj D. Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy of high-performance deep learning convolutions on SIMD architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018.* IEEE / ACM, 66:1–66:12.

[21] Evangelos Georganas, Kunal Banerjee, Dhiraj D. Kalamkar, Sasikanth Avancha, Anand Venkat, Michael J. Anderson, Greg Henry, Hans Pabst, and Alexander Heinecke. 2020. Harnessing Deep Learning via a Single Building Block. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 222–233.

[22] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.* 34, 3 (2008), 12:1–12:25.

[23] Kazushige Goto and Robert A. van de Geijn. 2008. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw.* 35, 1 (2008), 4:1–4:14.

[24] Dominik Grewe, Zheng Wang, and Michael FP O'Boyle. 2013. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE, 1–10.

[25] Gabriel Haeser, Oliver Hinder, and Yinyu Ye. 2021. On the behavior of Lagrange multipliers in convex and nonconvex infeasible interior point methods. *Math. Program.* 186, 1 (2021), 257–288.

[26] Qingchang Han, Yongmin Hu, Fengwei Yu, Hailong Yang, Bing Liu, Peng Hu, Ruihao Gong, Yanfei Wang, Rui Wang, Zhongzhi Luan, and Depei Qian. 2020. Extremely Low-bit Convolution Optimization for Quantized Neural Network on Modern Computer Architectures. In *ICPP 2020: 49th International Conference on Parallel Processing, Edmonton, AB, Canada, August 17-20, 2020.* ACM, 38:1–38:12.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 770–778.

[28] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016.* IEEE Computer Society, 981–991.

[29] Zhen Jia, Aleksandar Zlateski, Frédo Durand, and Kai Li. 2018. Optimizing N-dimensional, winograd-based convolution for manycore CPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018.* ACM, 109–123.

[30] Haidong Lan, Jintao Meng, Christian Hundt, Bertil Schmidt, Minwen Deng, Xiaoning Wang, Weiguo Liu, Yu Qiao, and Shengzhong Feng. 2020. FeatherCNN: Fast Inference Computation with TensorGEMM on ARM Architectures. *IEEE Trans. Parallel Distributed Syst.* 31, 3 (2020), 580–594.

[31] Daniel Langr and Pavel Tvrdik. 2015. Evaluation criteria for sparse matrix storage formats. *IEEE Transactions on parallel and distributed systems* (2015).

[32] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. 2019. A coordinated tiling and batching framework for efficient GEMM on GPUs. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019.* ACM, 229–241.

[33] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Ortí. 2016. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Trans. Math. Softw.* 43, 2 (2016), 12:1–12:18.

[34] Filippo Mantovani, Marta Garcia-Gasulla, José Gracia, Esteban Stafford, Fabio Banchelli, Marc Josep-Fabrego, Joel Criado-Ledesma, and Mathias Nachtmann. 2020. Performance and energy consumption of HPC workloads on a cluster based on Arm ThunderX2 CPU. *Future Gener. Comput. Syst.* 112 (2020), 800–818.

[35] William F Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2014. Active learning accelerated automatic heuristic construction for parallel program mapping. In *Proceedings of the 23rd international conference on Parallel architectures and compilation.* 481–482.

[36] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020.* IEEE, 58–70.

[37] Tran Minh Quan, David G. C. Hildebrand, and Won-Ki Jeong. 2016. FusionNet: A deep fully residual convolutional neural network for image segmentation in connectomics. *CoRR* abs/1612.05360 (2016).

[38] Nikola Rajovic, Alejandro Rico, Filippo Mantovani, Daniel Ruiz, Josep Oriol Vilarrubi, Constantino Gomez, Luna Backes, Diego Nieto, Harald Servat, Xavier Martorell, et al. 2016. The Mont-Blanc prototype: an alternative approach for HPC systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 444–455.

[39] Cody Rivera, Jieyang Chen, Nan Xiong, Jing Zhang, Shuaiwen Leon Song, and Dingwen Tao. 2021. TSM2X: High-performance tall-and-skinny matrix-matrix multiplication on GPUs. *J. Parallel Distributed Comput.* 151 (2021), 70–85.

[40] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, Kouichi Hirai, Atsushi Furuya, Akira Asato, Kuniki Morita, and Toshiyuki Shimizu. 2020. Co-design for A64FX manycore processor and "Fugaku". In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020.* IEEE/ACM, 47.

[41] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings.*

[42] Tyler M. Smith and Robert A. van de Geijn. 2019. The MOMMS Family of Matrix Multiplication Algorithms. *CoRR* abs/1904.05717 (2019).

[43] Tyler M. Smith, Robert A. van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. 2014. Anatomy of High-Performance Many-Threaded Matrix Multiplication. In *2014 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 1049–1059.

[44] Nigel Stephens. 2016. Armv8-a next-generation vector architecture for HPC. In *2016 IEEE Hot Chips 28 Symposium (HCS).* IEEE, 1–31.

[45] Xing Su, Xiangke Liao, Hao Jiang, Canqun Yang, and Jingling Xue. 2019. SCP: Shared Cache Partitioning for High-Performance GEMM. *TACO* 15, 4 (2019), 43:1–43:21.

[46] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael FP O'Boyle. 2009. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation.* ACM, 177–187.

[47] Feng Wang, Hao Jiang, Ke Zuo, Xing Su, Jingling Xue, and Canqun Yang. 2015. Design and Implementation of a Highly Efficient DGEMM for 64-Bit ARMv8 Multi-core Processors. In *44th International Conference on Parallel Processing, ICPP 2015, Beijing, China, September 1-4, 2015.* IEEE Computer Society, 200–209.

[48] Zheng Wang, Dominik Grewe, and Michael FP O'boyle. 2014. Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 4 (2014), 1–26.

[49] Zheng Wang and Michael FP O'Boyle. 2010. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques.* 307–318.

[50] Zheng Wang and Michael O'Boyle. 2018. Machine learning in compiler optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901.

[51] Zheng Wang, Georgios Tournavitis, Björn Franke, and Michael FP O'boyle. 2014. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 1 (2014), 1–26.

[52] Weiling Yang, Jianbin Fang, and Dezun Dong. 2021. Characterizing Small-Scale Matrix Multiplications on ARMv8-based Many-Core Architectures. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 101–110.

[53] Guixin Ye, Zhanyong Tang, Huanting Wang, Dingyi Fang, Jianbin Fang, Songfang Huang, and Zheng Wang. 2020. Deep program structure modeling through multi-relational graph-based learning. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques.* 111–123.

[54] Xin You, Hailong Yang, Zhongzhi Luan, Yi Liu, and Depei Qian. 2019. Performance Evaluation and Analysis of Linear Algebra Kernels in the Prototype Tianhe-3 Cluster. In *Supercomputing Frontiers - 5th Asian Conference, SCFA 2019, Singapore, March 11-14, 2019, Proceedings (Lecture Notes in Computer Science)*, Vol. 11416. Springer, 86–105.

[55] Raphael Yuster and Uri Zwick. 2005. Fast sparse matrix multiplication. *ACM Transactions On Algorithms (TALG)* 1, 1 (2005), 2–13.

[56] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* 41, 3 (2015), 14:1–14:33.

[57] Peng Zhang, Jianbin Fang, Tao Tang, Canqun Yang, and Zheng Wang. 2018. Auto-tuning streamed applications on intel xeon phi. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 515–525.

[58] Peng Zhang, Jianbin Fang, Canqun Yang, Chun Huang, Tao Tang, and Zheng Wang. 2020. Optimizing Streaming Parallelism on Heterogeneous Many-Core Architectures. *IEEE Trans. Parallel Distributed Syst.* 31, 8 (2020), 1878–1896.

[59] Xianyi Zhang, Qian Wang, and Yunquan Zhang. 2012. Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor. In *18th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2012, Singapore, December 17-19, 2012.* IEEE Computer Society, 684–691.