Characterizing Small-Scale Matrix Multiplications on ARMv8-based Many-Core Architectures

Weiling Yang[§], Jianbin Fang[§], Dezun Dong
College of Computer Science, National University of Defense Technology
Changsha 410073, China
{w.yang, j.fang, dong}@nudt.edu.cn

***Abstract*—General Matrix Multiplication (GEMM) is a key subroutine in high-performance computing. There is a large body of work on evaluating and optimizing large-scale matrix multiplication, but how well the small-scale matrix multiplication (SMM) performs is largely unknown, especially for the ARMv8-based many-core architectures. In this work, we evaluate and characterize the performance of SMM subroutines on Phytium 2000+, an ARMv8-based 64-core architecture. The evaluation work is extensively performed with the mainstream open-source libraries including OpenBLAS, BLIS, BALSFEO, and Eigen. Given various experimental settings, we observe how well the small-scale GEMM routines perform on Phytium 2000+, and then discuss the impacting factors behind the performance behaviours of SMM. Built on such a basis, we shed light on the performance bottlenecks and practical optimizations on SMM from various angles: (1) mitigating the data packing overhead, (2) processing the edge cases properly, (3) selecting a suitable micro-kernel, and (4) adopting a right parallelization method. The result of our work facilitates users to develop efficient SMM optimizations on ARMv8-based many-core architectures, and embed them into real-world applications.**

***Index Terms*—Phytium 2000+, Small-scale GEMM, Performance evaluation and optimization**

I. INTRODUCTION

The high-performance computing (HPC) hardware is firmly moving towards many-core designs [1]–[6]. Among them, the ARMv8-based many-core processors, examplified by `Phytium 2000+` and `A64FX`, are emerging as interesting alternative building blocks to the conventional `x86` processors [7]–[12]. Phytium 2000+ has been used to build the prototype of China's new-generation exascale supercomputer [13], and `A64FX` is the building block of the `Fugaku` supercomputer [12]. Therefore, it is vital to understand the performance behaviours of typical HPC applications and kernels on such many-core architectures.

General matrix multiplication (GEMM) is one of the most fundamental linear algebra subroutines in big data analytics and scientific computing. Processor vendors have provided efficient BLAS (Basic Linear Algebra Subroutines) implementations that are highly optimized for their relevant micro-architectures, e.g., Intel MKL [14], AMD ACML [15], and NVIDIA cuBLAS [16]. The HPC community has also contributed several high-quality open-source BLAS implementations, e.g., ATLAS [17], GotoBLAS [18], OpenBLAS [19], and BLIS [20]. Both the vendor-specific BLAS libraries and the open-source libraries have been heavily optimized for large-scale matrices on the mainstream `x86` processors.

On the other hand, small-scale matrix multiplication (SMM) equally plays an important role in many practical applications. As an example, deep neural networks (DNN) are typically built based on SMMs [21], [22]. Blocksparse matrix formats such as Block Compressed Sparse Row Format can also substantially benefit from fast SMMs [23]. When GEMM is used for encoding checksums in the Algorithm-Bases Fault Tolerance applications, the input often involves a tall-and-skinny checksum weight matrix [24]. In this work, we refer *small-scale matrix multiplication (SMM)* to be that the three dimensions M, N, K of matrix multiplications are all very small or one dimension is significantly smaller than the other two (e.g., $M \ll N$ and $M \ll K$).

There is a large body of research work on large-scale matrix multiplications, but there is very little work for SMMs. `LIBXSMM` is a SMM library targeted the `x86` processors [23]. Prior work focuses on optimizing large-scale matrix multiplications on Phytium 2000+ [25]. But how well the SMM subroutines perform on ARMv8-based many-core architectures is largely unknown. Therefore, it is significant to evaluate SMM's performance to fully tap the computational potentials.

In this work, we use four mainstream open-source BLAS libraries (OpenBLAS [19], BLIS [20], BLASFEO [26] and Eigen [27]) to evaluate the performance of SMMs on Phytium 2000+. We characterize how well SMMs perform with various experimental settings, i.e., we use various sizes of small matrices and different number of hardware cores. We observe that there is a large gap between the achieved SMM performance and the machine's theoretical peak. Diving into the performance behaviours of SMMs with analytical models, we identity the impacting factors including (1) mitigating the data packing overhead, (2) processing the edge cases properly, (3) selecting a suitable micro-kernel, and (4) adopting a right parallelization method. Built on such a basis, we shed light on the performance bottlenecks and practical optimizations for SMMs.

Our main contributions are summarized as follows.

- We provide a systematic performance evaluation of SMMs on Phytium 2000+. We observe that the achieved performance of SMMs varies over BLAS libraries, and is far below the Phytium 2000+'s theoretical peak.

[§] Equal contribution

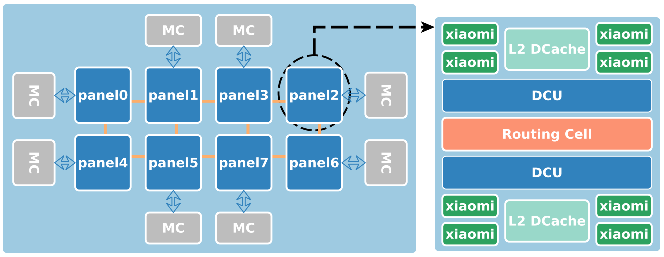


Fig. 1: A high-level view of the Phytium 2000+ architecture. Processor cores are groups into panels (left) where each panel contains eight ARMv8 based Xiaomi cores (right)

- We analyze the performance impacting factors of SMMs on Phytium 2000+. These factors include the data packing overhead, how to process edge cases, the selection of micro-kernels and the parallelization method.
- Detecting the performance bottlenecks, we discuss how to optimize SMMs and summarize a reference implementation of high-performance SMMs.

II. Background

A. Phytium 2000+ Architecture

Figure 1 shows that Phytium 2000+ integrates 64 ARMv8 based Xiaomi cores. The CPU chip has eight panels, each with eight cores running at 2.2 GHz [8], [9]. It can offer a peak performance of 563.2 Gflops for double-precision operations, with a maximum power consumption of 96 Watts.

Each Xiaomi core is an ARMv8-compatible core implemented as a superscalar, out-of-order, 4-decode/4-dispatch pipeline with a hybrid branch prediction. The front-end consists of a 32KB instruction cache and a prefetch, for instruction fetching and decoding. On hits, 2 cycles are required for retrieval of instructions from the L1. The back-end performs operations out-of-order for the most part and is in charge of queuing instructions, executing them and retiring them. Dispatch contains a 160-entry ReOrder Buffer (ROB) and can dispatch up to 4 instructions per cycle. From dispatch, out-of-order instructions go into 4 discrete scheduling queues: 2x Integer/SIMD, 1x FP/SIMD, and 1x Load/Store. The Int/FP queues are each 16-entry deep.

Each core has 32 128-bit vector registers, each of which is capable of storing four single-precision floating-point numbers, and a private 32KB L1 data cache. Four cores share a 2MB L2 cache. The panels are connected through two directory control units (DCU). Each panel connected to its own DDR4 memory module through the Memory Controller (MC), shared by eight cores of this panel.

B. GEMM Structure

GEMM performs a matrix-multiply-accumulate operation $C = \alpha AB + \beta C$, where A, B, and C are matrices of sizes $M \times K$, $K \times N$, and $M \times N$, respectively, and α, β are scalars. GEMM can be implemented with a straightforward three-nested loop, but achieving high performance is complicated. This is mainly due to the fact that modern many-core architectures have a hierarchical memory organization. Figure 4 illustrates the GEMM algorithm proposed by *Goto* [28], including its multiple layers for blocking (to maximize cache performance) and packing (to enable data to be moved efficiently to the registers). Each loop of the original three loop nests is tiled, resulting in a six-nested loop (referred to as Layers 1-6) [25].

The outermost loop at Layer 1 partitions C and B into (wide) column panels of sizes $M \times nc$ and $K \times nc$, respectively. The next loop at Layer 2 partitions A into column panels of size $M \times kc$, and further partitions a $K \times nc$ submatrix of B into row panels of size $kc \times nc$. At this loop level, *Goto's* packs the current $kc \times nc$ panel of B into a contiguous buffer $\tilde{B}$. When there is an L3 cache in the target processor, the buffer $\tilde{B}$ will reside completely in the L3 [29]; otherwise, it will have to be kept in the main memory. The outermost two loops partition matrix A and B to panel-panel multiplication (GEPP). The Layer 3 partitions the $M \times kc$ panels of A and a $M \times nc$ submatrix of C into $mc \times kc$ blocks and row panels of size $mc \times nc$. At this point, blocks of A are packed into a contiguous buffer $\tilde{A}$. The buffer is sized to occupy the majority of the L2 cache, leaving sufficient space to prefetch other data and the sliver of $\tilde{B}$. Up to now, we have $\tilde{A}$ in the L2 cache and $\tilde{B}$ in the L3 cache (or main memory).

The innermost three loops of Layers 4 – 6 is an architecture-specific kernel, known as GEBP, which updates an $mc \times nc$ panel of C by calculating the outer product of a block $\tilde{A}$ of $mc \times kc$ and a panel $\tilde{B}$ of $kc \times nc$. The GEBP algorithm is listed in Algorithm 1 [19]. The Layer 4 partitions a block $\tilde{B}$ into slivers of $kc \times nr$ and the Layer 5 partitions a block $\tilde{A}$ into slivers of $mr \times kc$. Layer 5 and Layer 6 are referred to as GEBS and GESS, respectively [29]. GESS, also known as the micro-kernel in BLIS, performs a sequence of rank-1 updates of an $mr \times nr$ sub-block of C using an $mr \times 1$ column sub-sliver of $\tilde{A}$ and a $1 \times nr$ row sub-sliver of $\tilde{B}$. Each rank-1 update is performed at Layer 7.

Figure 2 describes the data movements across all the levels of memory hierarchy. *Goto's* uses the outer-product formulation to update a $mr \times nr$ block of C in the registers [28]. Note that we have to load sub-sliver of $\tilde{A}$ and $\tilde{B}$ from the L1 cache for the next iteration (Lines 7-9 of Algorithm 1). A $kc \times nr$ sliver of $\tilde{B}$ resides in the L1 cache during all the iterations at Layer 5. The $mr \times kc$ sliver of $\tilde{A}$ is streamed from L2, and we have to reserve sufficient space in L1 for prefetching the sub-slivers of $\tilde{A}$.

C. GEMM Implementations in BLAS Libraries

This subsection compares four open-source BLAS libraries, including OpenBLAS [19], BLIS [20], BLASFEO [26], and Eigen [27], in terms of blocking algorithms, data packing, parallelization methods, and kernel routines.

Blocking algorithm. Figure 4 shows the blocking algorithm of GEMM in OpenBLAS and BLIS. The matrix is stored in the column-major format by default, so the algorithm starts to block from the N dimension. But in Eigen, the matrix is

ALGORITHM 1 GEBP

```
Input: $C$, $\tilde{A}$, $\tilde{B}$, $\alpha$
Output: $C = \alpha\tilde{A}\tilde{B} + C$
for $j = 1 \rightarrow nc$, step $= nr$ do
    for $i = 1 \rightarrow mc$, step $= mr$ do
        $TEMP\_C_{mr\times nr} = 0$
        $Load\ mr\ elements\ of\ \tilde{A}_{i1}$
        $Load\ nr\ elements\ of\ \tilde{B}_{1j}$
        for $k = 1 \rightarrow kc$, step $= 1$ do
            $Compute\ TEMP\_C_{mr\times nr} += \tilde{A}_{ik} \times \tilde{B}_{kj}$
            $Load\ next\ mr\ elements\ of\ \tilde{A}_{ik}$
            $Load\ next\ nr\ elements\ of\ \tilde{B}_{kj}$
        end for
        $Load\ mr \times nr\ elements\ of\ C_{ij}$
        $Compute\ C_{ij} += \alpha TEMP\_C_{mr\times nr}$
        $Store\ C_{ij}$
    end for
end for
```

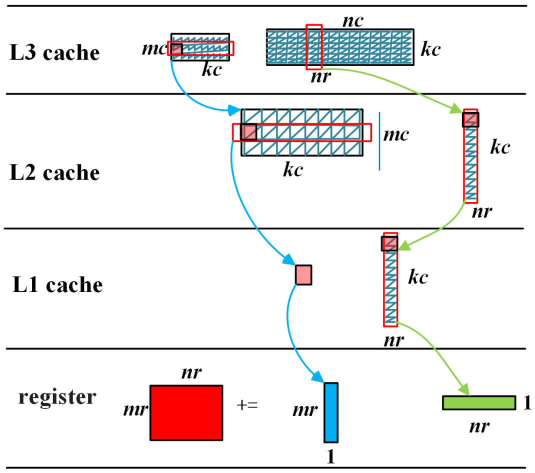


Fig. 2: Packed data storage for GEBP in GotoBLAS.

stored in the row-major format, and its implementation starts from the M dimension. BLASFEO is targeted for small-scale matrices, and skips Layers 1-3 in Figure 4 [26].

Data packing. To better exploit the caches, matrices are blocked and packed into continuous memory buffers that fit into the caches before the kernel execution. In the outer three loops of OpenBLAS, BLIS and Eigen, matrices A and B are packed into $\tilde{A}$ and $\tilde{B}$, and their storage format is shown in Figure 2. The width of the panel inside $\tilde{A}$ and $\tilde{B}$ is mr and nr, respectively. Because BLASFEO does not have the outer three-layer loop similar to OpenBLAS, the column-major format needs to be converted to the panel-major format before matrix multiplication. The panel size is fixed and denoted by ps. Both mr and nr are a multiple of ps [26]. The panel-major format is shown in Figure 3. The difference between the panel-major format and the format of $\tilde{A}$ is that the size of the former is fixed and we have to finish format convertion at the very beginning in Figure 4. Since data blocks are likely to be reused, it makes sense to store them in a panel-major format that is particularly favorable for the SMMs [26].

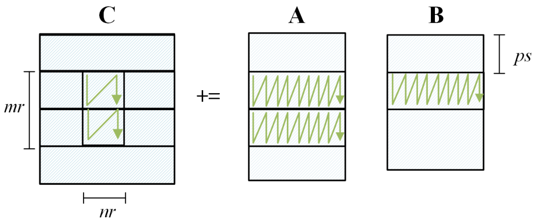


Fig. 3: Storage format of BLASFEO: panel-major.

TABLE I: A comparison of library kernels

	OpenBLAS	BLIS	BLASFEO	Eigen
Layers of assembly	Layer 4-7	Layer 6-7	Layer 6-7	none
unrolling factor	8	4	4	1
$mr \times nr$	16x4,8x8,4x4	8x12	16x4,8x8	12x4

Parallelization method. On multi-core CPUs, the GEMM routines use OpenMP or Pthreads to exploit the computational potentials. Marker *et al.* proposes a two-dimensional parallelization for GEMM [30], where the whole computing task is partitioned into a two-dimensional grid, and each thread works on updating a GEPP sub-task. Typically, all the sub-tasks in the same row are assigned to the same thread. OpenBLAS adopts such a parallel method. In contrast, BLIS uses a multi-dimensional GEMM parallelization method, which supports the parallel execution on various loop levels [31]. Specifically, this method enables the user to perform parallelization on any combination of jj-loops, ii-loops, j-loops and i-loops in Figure 4. Suppose that a processor has 64 threads. The BLIS approach will use 8 threads to parallelize the ii-loop, and let each thread perform a GEBP task. Within a GEBP task, it will further use 8 threads to parallelize the j-loop. This method helps to parallelize the task space in a more fine-grained manner. The BLASFEO currently provides only single-threaded routines for SMMs [26].

Kernel routine. At the core of in Figure 4 is the kernel routine (Layers 4-7). The kernel routine of OpenBLAS is implemented in assembly. Layers 6-7 is also known as *micro-kernel*. Both BLIS and BLASFEO use assembly to implement this micro-kernel. The size of micro-kernel (i.e. $mr \times nr$) and the loop unrolling factor has to be selected carefully according to hardware features. Table I compares the differences of kernel rountines among the BLAS libraries.

III. Empirical Evaluation

This section provides a comprehensive performance presentation of the small-scale matrix multiplications (SMM) on the Phytium 2000+ processor and discusses the impacting factors behind the performance behaviours.

Experimental settings. To investigate the factors that affect the performance of SMM, we evaluate the performance of OpenBLAS, BLIS, BLASFEO and Eigen with a single thread

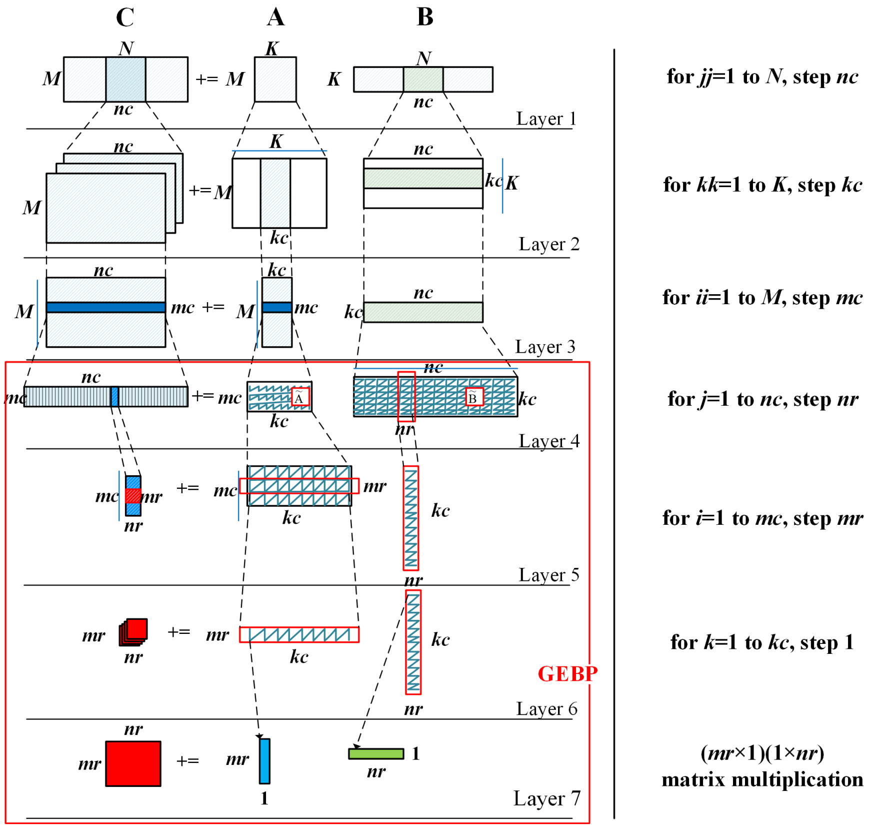


Fig. 4: Blocking algorithms used in implementing GEMM, where GEBP is the inner kernel highlighted inside a red box.

(Figure 5) and multiple threads (Figure 10), respectively, on Phytium 2000+. Note that, in Figure 5, the overall size of the input data should be smaller than the size of the L2 cache, which is to highlight the "small" feature of input matrices [23]. In Figure 5(a), the size of the square matrix ranges from 5 to 200, with a step of 5. In Figure 5(b), we aim to explore the effect of changing M, whose size ranges from 2 to 40, with a step of 2. We explore the effects of changing N in Figure 5(c) and that of K in Figure 5(d). To investigate the performance of various parallelization methods, we also evaluate the performance of SMM with multiple threads on Phytium 2000+. The small-scale matrices feature "irregular" shapes, where the size of one dimension is significantly smaller than the other dimensions. For each measurement, the execution time is calculated as the average of 20 runs.

A. The Data Packing Overhead

Observation. From Figures 5(a)–5(c), we see that BLASFEO performs significantly better than the other BLAS libraries, and using Eigen yields bad GEMM performance. In the best case, BLASFEO can reach 96% of the theoretical peak of the Phytium 2000+ processor, while Eigen can only reach 58%. In addition, the performance behaviors of SMM for small Ks (Figure 5(d)) differ significantly from those of small Ms (Figure 5(b)) or Ns (Figure 5(c)).

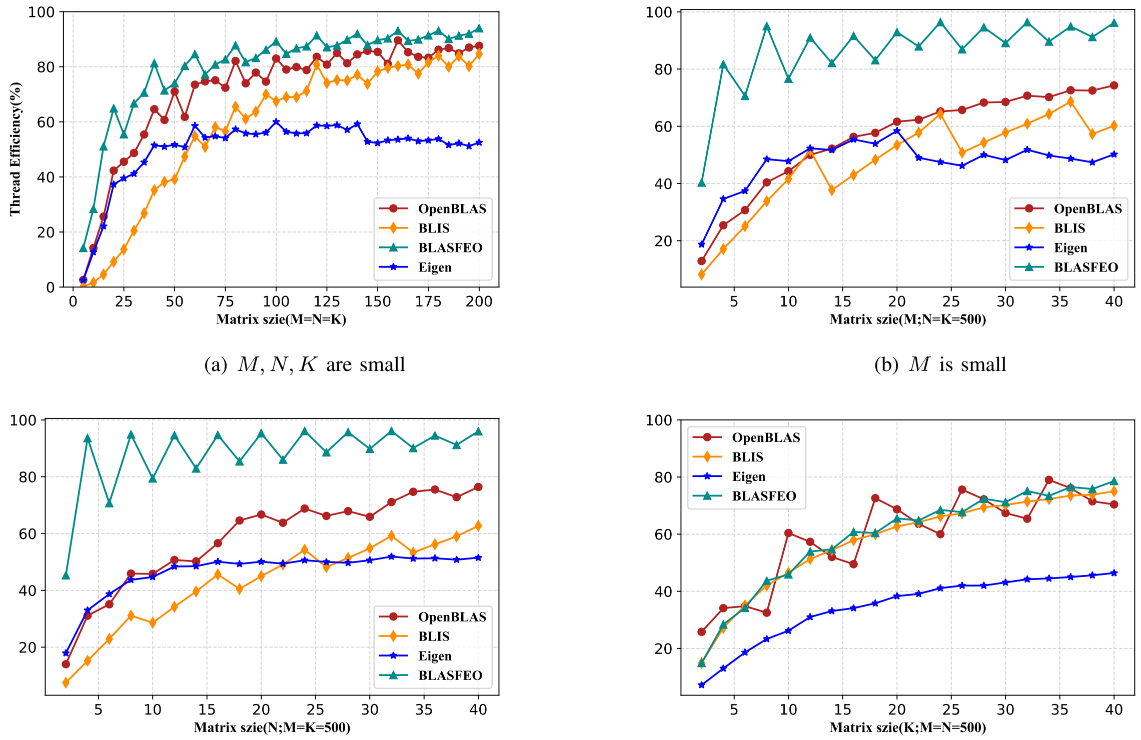
Analysis. The matrix used in BLASFEO is stored in a panel-major format, which has no data packing cost (Figure 3). In contrast, OpenBLAS, BLIS and Eigen have to load data into a panel data structure [19], [20], [28], which is essential to exploit the on-chip caches on modern multi-core CPUs. Further, the incurred overhead of data packing can be ignored for large-scale GEMM. Su *et al.* have shown that the packing overhead accounts for only around 3% of the total running time for large-scale GEMM [25].

However, the data packing step has a significant impact on the performance of SMM [26]. During data packing, we load data elements for a total of $O(M \times N + K \times N)$, and thereafter we run calculations of $O(M \times N \times K)$. We use packing-to-computing ratio ($P2C$) to quantify the impact of data packing. The number of load instructions required for data packing is:

$$Num\_Load = \frac{M \times N + K \times N}{Load\_width} \tag{1}$$

where the numerator denotes the total number of data elements for the matrix A and B, and $Load\_width$ denotes the number of floating-point data elements achieved by a load request. Because the vector register of Phytium 2000+ is 16 bytes, the $Load\_width$ is $16/sizeof(float) = 4$. The number of the required arithmetic FMA instructions for GEMM can be approximated as follows:

$$Num\_FMA = \frac{M \times N \times K}{FMA\_width} \tag{2}$$



(a) M, N, K are small

(b) M is small

(c) N is small

(d) K is small

Fig. 5: The SMM performance with a single thread.

where the FMA_width means the number of floating-point data a FMA instruction can compute. FMA_width of Phytium 2000+ is $2 \times 16/sizeof(float) = 8$. Therefore, the $P2C$ can be defined as follows:

$$P2C = Num\_Load/Num\_FMA = \frac{M+N}{2 \times M \times N} \tag{3}$$

A smaller $P2C$ means that the data packing overhead can be better amortized by computing operations. When M and N increase, the ratio decreases. Therefore, for GEMM with small M or N, the packing overhead has a great impact on it.

To illustrate the data packing impact, we measure the breakdowns of SMM with OpenBLAS. We see from Figure 6 that the smaller the M or N, the larger the proportion of the packing overhead. In the worst cases, it accounts for more than 50%. As a result, the data packing step becomes the performance bottleneck for SMM. Figure 6 also shows that, when K is very small, this overhead can be ignored. This is because $P2C$ is independent of K (Equation 3).

Summary. In the case of small Ms or Ns, the packing overhead should be avoided as much as possible. At this time, We recommend using the panel-major format implemented by BLASFEO. Rather than using a fixed kernel, its size can be calculated by the size of micro-kernel, i.e., $mr \times nr$. But when the BLASFEO panel format does not work, we have to redesign the SMM algorithm and implementation, aiming to hide the data packing overhead. The strategies such as avoiding data packing and choosing to pack the partial of the relevant matrix, can be used to achieve this goal.

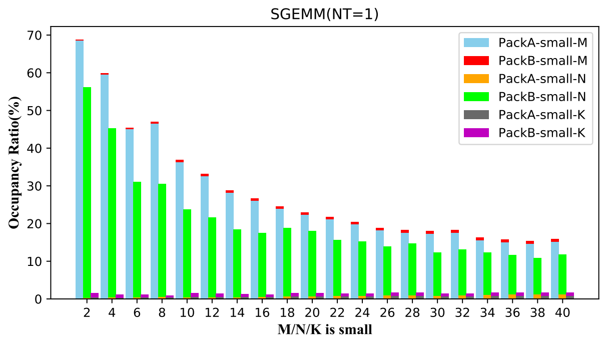


Fig. 6: Data packing overhead for SMMs.

B. Processing Method of Edge Cases

Observation. From Figures. 5(a)–5(c), we see that the performance of these BLAS libraries fluctuates over the matrix size. Taking OpenBLAS as an example, the performance of $M = N = K = 80$ is 83.5%, which is significantly better than its neighbouring matrix settings ($M = N = K = 75\ or\ 80$).

Analysis. When the matrix size is not a multiple of the micro-kernel (i.e., M is not a multiple of mr, and N is not a multiple of nr), we have to deal with the edge cases. There

```
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]
```

Fig. 7: The 8x4 micro-kernel in OpenBLAS.

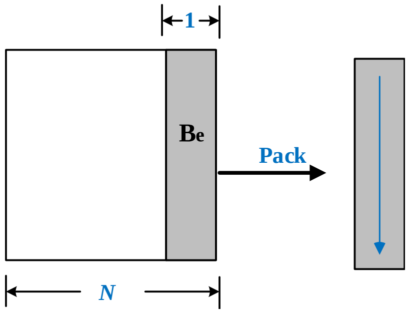


Fig. 8: Packing the edge to better use SIMD instructions.

are two main methods for edge case processing, which are data padding and using edge micro-kernels [19], [20], [26]. The data padding approach will fill the edge case with zeros, and thus introduce an additional overhead. This approach is applied in BLIS and BLASFEO. As an alternative, the edge micro-kernel is specially designed for computing edge cases, and applied in the OpenBLAS library. Both approaches suffer a loss in performance, when there occurs an edge case. Different from that in large-scale GEMM, processing edge cases in SMM takes a much larger percent. Consequently, it is significant to deal with the edge cases of SMM efficiently.

There are two main factors that determine the impact of the padding method on performance, i.e., the size of M and N, and the size of the edge case. The smaller the matrix means that more zeros need to be padded, and these zeros are also involved in the computation. When M and N are larger, these extra computational overheads can be better amortized.

The performance of the edge micro-kernel method depends on the size of an edge case and its implementation. We use GEMM $M = 75, N = K = 60$ as an example, and for this case, the size of the edge case is 11×4. Calculating the edge case requires a combination of the $8 \times 4, 2 \times 4, and\ 1 \times 4$ micro-kernels. By diving into the OpenBLAS's source code, we note that such micro-kernels are not highly optimized, as shown in Figure 7. We see that there are four `ldr` instructions next to each other, while Phytium 2000+ has only two load units. Moreover, the distance between the two dependent instructions is too close to completely hide the latency slots. There are many other inefficient edge micro-kernels in OpenBLAS, and some even have a poor utilization of the SIMD instructions.

To illustrate the impact of the edge kernel methods, we measure the kernel performance in OpenBLAS. In the experiment, we fix M or N or $K = 100$ respectively, which helps to characterize the performance impact of each dimension. From Figure 9, we can see that the kernel can reach the best performance (93.3% of the peak) when $M = 80$ and $N = 80$. In the worst cases, the performance is only 71.8%. This performance gap is mainly contributed to the usage of inefficient edge micro-kernels. Therefore, the processing of edge cases is critical to achieving high-performance SMMs.

Summary. Both the data padding method and the edge micro-kernel method are able to efficiently deal with the edges cases. But for SMMs, we prefer the latter one. There is a guiding principle for designing an efficient edge micro-kernel: to use aligned vector loads/stores and FMA instructions to facilitate vectorization. Although Section III-A summarizes that the data packing overhead should be avoided, we recommend hereby packing the small amount of edge data to better fit the SIMD unit. Suppose that the edge case size of N is 1 (i.e., $N\%nr = 1$), as shown in Figure 8. Without data packing, the accesses to the data elements of the edge case `Be` is discontiguous. In this case, the ARMv8-based processor can not fully use FMA instruction. Therefore, the data packing step is introduced here to deal with the edge case.

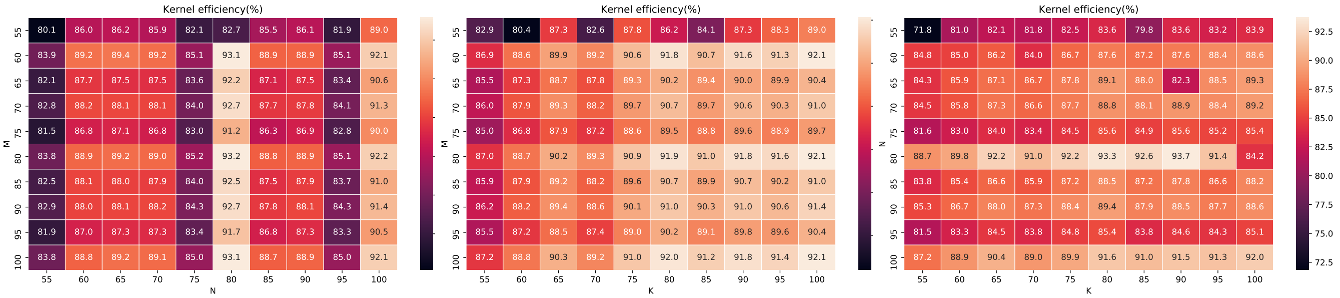
C. Micro-kernel Selection

Observation. From Figures 9(a)–9(c), we see that the best achieved efficiency of the SMM kernel is 93.3%. We argue that there is still room for performance improvement in terms of the micro-kernel design.

Analysis. The micro-kernels in OpenBLAS, BLIS, BLASFEO, and Eigen have not been optimized specifically for Phytium 2000+. For the best performance, each micro-kernel and its instruction layout need to be optimized according to the hardware specifics. Among them, the size of micro-kernel affects compute-to-memory ratio (CMR) [29], and the layout of instructions affect the distance between dependent instructions. These two indicators are closely related to the latency hiding among instructions.

The micro-kernel is calculated based on the outer-product approach, which produces the result block $(mr \times nr)$ in matrix C. The size of $(mr \times nr)$ is determined by the number of vector registers in the processor. The ARMv8-based processor offers 32 4-element-wide vector registers (called V0-V31), assuming single precision. To store this C-result buffer, $\frac{mr \times nr}{4}$ out of the 32 registers are needed. In addition, matrix A needs at least one vector register to stage the temporary result, and so does matrix B. Therefore, we have the following constraint when designing a performant micro-kernel:

$$\frac{mr \times nr}{4} \leqslant (32 - 2) \tag{4}$$



(a) $K = 100$

(b) $N = 100$

(c) $M = 100$

Fig. 9: Kernel efficiency for SMMs in OpenBLAS (Note that this does not include the overhead of data packing).

$mr \times nr$ also has to ensure a large CMR, which helps hide the memory access overhead. The CMR can be defined as:

$$CMR = \frac{2 \times mr \times nr}{mr + nr} \tag{5}$$

Theoretically, the larger the CMR, the greater the potential for the memory access overhead to be hidden.

Furthermore, we have to consider the distance between the Load and FMA instructions with a RAW dependency. In the case of $mr = 12, nr = 10$, only 1 vector register is available for A, and the same holds for B. At this time, the distance between the Load and FMA instructions with RAW dependency is 1. For the Phytium 2000+ processor, the latency of fetching a word from L1 is 3 cycles [7], while that of L2 and memory is much larger. In other words, the FMA instruction has to wait at least 3 cycles until the required data is available. Therefore, the instructions within a micro-kernel have to be scheduled carefully to eliminate latency bubbles. Also the loop within the micro-kernel has to be unrolled properly according to the capacity of the instruction cache.

Summary. When designing a performant micro-kernel, mr and nr has to be selected properly to statisfy the constraints (4), and (5). Once these parameters are specified, the instruction layout of the micro-kernel needs to be carefully designed to coordinate the computations and memory accesses.

D. Parallelization Method

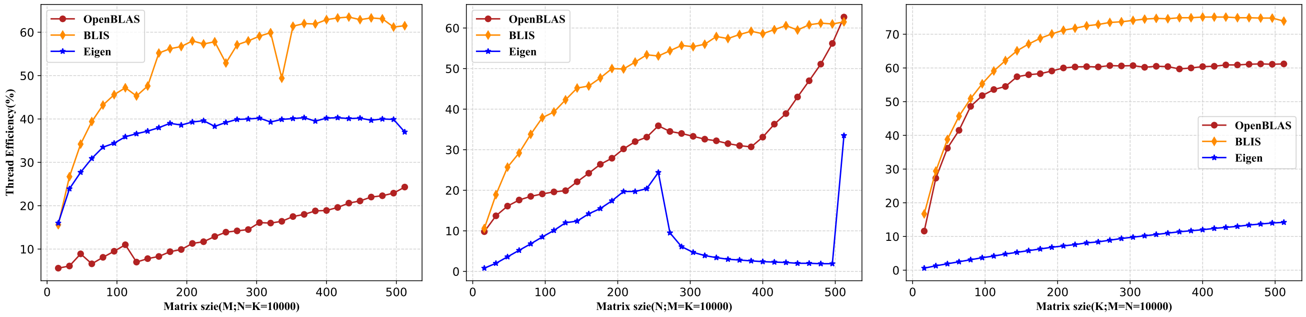
Observation. From Figures 10(a)–10(c), we see that BLIS performs the best for the small cases with 64 threads on Phytium 2000+. OpenBLAS has especially poor performance when M is small. When either M or N is very small, the performance of the three BLAS libraries is far below the theoretical peak of Phytium 2000+. In this case, BLIS is the best performer among them, peaking at around 60%.

Analysis. The BLAS libraries (OpenBLAS, BLIS and Eigen) use different parallelization methods. BLIS uses a multi-dimensional GEMM parallelization method [31], which chooses suitable loop levels to parallelize based on the matrix feature. When a dimension is particularly small, BLIS will choose not to parallelize this dimension, which helps to reduce the workloads of processing edge cases per core. Let us take SMM with $M = 64$ as an example. If using 64 threads to parallelize the ii or i loop in this case, we will obtain that $mc = mr = 1$. As we have discussed in Section III-B, this computing method has a significant negative impact on performance. But OpenBLAS and Eigen are not able to do so. They divide the task matrix C into a two-dimensional grid. Each thread uses the inner three loops in Figure 4 to update the task block. When M or N is very small, there would be a large number of edge cases.

Note that parallelizing the inner loops engenders better spatial locality. This is because by doing so can we have one large contiguous data block, rather than many small individual data blocks (i.e., $\tilde{A}$ or $\tilde{B}$) [31]. This helps to maintain the high performance of micro-kernels. When looking into the source code, we note that BLIS prefers parallelizing the j loop or the i loop when M and N are large enough. But OpenBLAS cannot parallelize these two loops. This further explains why BLIS performs better than OpenBLAS.

Synchronization is required when using multiple threads and it mainly occurs in three locations: packing $\tilde{A}$, packing $\tilde{B}$, and the end of the kk loop. Our goal is to minimize the number of synchronizations, and each thread needs to compute sufficient workloads before synchronization. The workload here refers to the computing and packing of GEBP tasks assigned to each thread. Considering the synchronization overhead, we should avoid using too many threads to parallel the j loop and i loop. Thus, there will fewer threads involved in the synchronization process. For SMM, BLIS can control the synchronization operation in a more fine-grained manner than OpenBLAS. Taking $M = 128$ as an example, BLIS can use 8 threads to parallelize the jj loop and 8 threads to parallelize the j loop. As a result, there are only 8 threads participating in each synchronization and each thread will have sufficient workloads (i.e., $mc \times \frac{nc}{8} \times kc$). But the number of threads used by OpenBLAS is 64 and the workload is $\frac{mc}{64} \times nc \times kc$, which creates a synchronization performance penalty. This is another reason why BLIS outperforms OpenBLAS for SMMs with 64 threads on Phytium 2000+.

To locate the reasons why BLIS's performance is still far below the hardware's theoretical peak, we measure the breakdowns of the multi-threaded SMM with small M. Table II



(a) M is small

(b) N is small

(c) K is small

Fig. 10: The performance comparison of OpenBLAS, BLIS and Eigen on SMMs with 64 threads.

TABLE II: The proportion of overhead of each part.(%)

M	Kernel	PackA	PackB	Sync	Kernel effic
16	35.5	2	56.9	4.2	43.6
32	45.1	2.1	47.7	4	59.3
48	50	5	38.4	5.6	68.6
64	57.9	4.5	31.2	5.6	73.6
80	57.4	5.6	30.4	5.8	74.9
96	64.5	4	25.1	5.7	71.8
112	68.4	3.9	21.6	5.5	72.8
128	70.2	10	17.4	1.7	67.7
144	74	10.8	12.5	2	71.1
160	74.4	7.5	15.3	2.2	67.6
176	74.4	8.8	13	3.1	72.8
192	79.6	5.5	14	0.3	73.5
208	77.3	5.9	13.8	2.5	73.6
224	79.8	6.9	10.5	2.4	75.2
240	78.2	6.4	10.4	4.5	74.7
256	82.2	6.5	9.7	1.2	74.6

shows that the main overhead comes from the kernel execution (`Kernel`) and the data packing for matrix B (`PackB`). Further, we note that the kernel efficiency is lower than that of the single-threaded SMM, ranging from 43% to 75%. We guess that there are three reasons that affect the performance of the multi-threaded kernel: (1) four cores share a non-LRU L2 cache on Phytium 2000+, which increases the L2 cache misses [25], (2) Phytium 2000+ is of an NUMA architecture, which affects the efficiency of load/store instructions. (3) to deal with edge cases, BLIS's approach will result in additional computational overheads. As for the overhead of PackB, its performance impact has been explained in Section III-A.

Note that we need to balance the efficiency of kernel, packing and synchronization. As we have aforementioned, we should use as many threads as possible to parallel j loop and i loop to achieve high kernel performance. Unfortunately, this also increases the overhead of synchronization and packing. Therefore, we have to tune the parallelization configuration in BLIS to balance them for high performance [20].

Summary. For SMMs, we recommend using the multi-dimensional parallelization method implemented by BLIS. That is, when the size of a dimension (M, N, or K) is small, we should choose not to parallelize that dimension. Besides, we should control the synchronization operation in a fine-grained manner to reduce the relevant overhead.

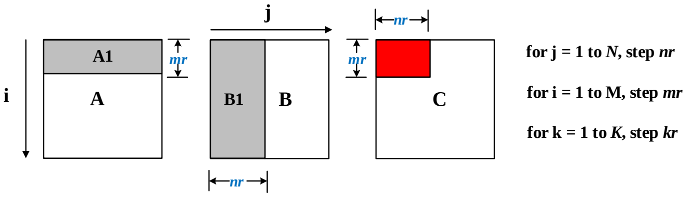


Fig. 11: Blocking algorithms used in implementing SMM.

IV. A Reference SMM Implementation

By characterizing SMM on Phytium 2000+, we have pinpointed the relevant performance bottlenecks. Built on the analytics, we propose that a reference implementation of high-performance SMMs should have the following features.

A packing-optional SMM. Section III-A shows that the step of data packing takes a large percent of SMMs. Also, the pipelining technique for large matrices is not applicable for small-scale matrices. Thus, we argue that a packing-optional implementation specially designed for SMM is required. To achieve this goal, the conventional GEMM implementation based on a six-nested loop should have to be restructured.

Having a set of optimal micro-kernels. Calculating GEMM falls into the calling of a combination of micro-kernels. Therefore, we have to design a set of optimal micro-kernels according to hardware features. A concept design is shown in Figure 11. We see that the sub-matrix $A1$ consistently resides in the L1 for k-loop, and sub-matrix $B1$ resides in the L1 for j-loop. Also, we pack $B1$ into continuous memory regions, which is integrated in the kernel execution. The micro-kernel here is different from the conventional implementation, which needs to be designed according to the characteristics of SMM.

Adaptive code generation. Sections III-B and III-C show that the best achieved performance for small matrices requires the usage of various micro-kernels and their combinations. Given an input matrix, we recommend using adaptive techniques such as JIT (Just-In-Time) to generate an optimal combination of micro-kernels. Further, the JIT technique helps to pre-calculate the offsets of memory accesses. In this way, we can achieve a match between SMMs and hardware architectures.

Multi-dimensional parallelization. Section III-D shows that using a fixed parallelization approach often leads to a poor performance. This is because SMM features an 'unpredictable' input shape, and performing parallelization on a wrongly selected dimension suffers a loss in performance. Thus, we have to use a multi-dimensional parallelization and make a run-time decision based on the input matrices.

V. Related Work

GEMM is an important operator for many applications in a broad range of domains. The GEMM routine based on X86 instruction has been thoroughly optimized and evaluated. Many research works focus on large-scale GEMMs [18]–[20] and SMMs [23], [32]. These optimized routines have been integrated into some deep learning frameworks based on X86 [33], [34].

Large-scale GEMM. In the memory access stage, Matrix A and matrix B are tiled and packed into a continuous memory according to the storage hierarchy of the processor [19], [28], [35]. In order to ensure that the data accessed each time resides in the cache. After GEMM enters the compution stage, it needs to call the kernel routine, which is usually written in assembly to keep the software pipeline busy [29], [36]. On multi-core CPUs, GEMM usually use threading APIs to exploit the computing potentials of processors [30], [31]. Their optimization work has achieved promising results for large-scale matrices. However, it is still unknown how well the small GEMM performs.

SMMs. To better integrate GEMMs into deep learning applications, researcher have optimized small-scale matrix multiplications. LIBXSMM is a dense linear library for SMMs on the X86 processors [23]. It uses the JIT technique to generate a customized kernel for each input matrix, and schedules instructions for the best performance. BLASFEO transforms the matrix storage format into panel-major, thus eliminating the data packing overhead in SMM [26]. But this panel-major format is not necessarily useful in practical applications. Kim *et al.* aims to make full use of the advantages of SIMD architecture, and proposes a new compact data layout that interleaves matrices in blocks according to the SIMD vector length [32]. To summarize, the prior works have not optimized and evaluated the SMMs on the ARMv8 architecture.

VI. CONCLUSIONS

This article evaluates the performance of GEMM routines in OpenBLAS, BLIS, BLASFEO and Eigen on Phytium 2000+. We mainly focus on small-scale matrix multiplications, analyzing the factors that affect the performance. We use a set of benchmarks to evaluate the performance of single thread, and multi-threaded SMMs. Then we investigate the performance gaps between the achieved performance and the hardware peak. At last, we propose a reference implementation of high-performance SMMs based on our analytics. For the next step, we plan to implement our reference SMM implementation and then integrate it into the deep learning frameworks.

VII. Acknowledgements

We thank the anonymous reviewers for their constructive comments. This work is partially supported by the National Key Research and Development Program of China under Grant No. 2018YFB0204301, the National Natural Science Foundation of China under Grant Nos. 61972408 and 61802146. For any correspondence, please contact Dezun Dong (Email: dong@nudt.edu.cn).

References

[1] M. A. Laurenzano, A. Tiwari, A. Cauble-Chantrenne, A. Jundt, W. A. W. Jr., R. L. Campbell, and L. Carrington, "Characterization and bottleneck analysis of a 64-bit armv8 platform," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016, Uppsala, Sweden, April 17-19, 2016*. IEEE Computer Society, 2016, pp. 36–45.

[2] N. Stephens, "Armv8-a next-generation vector architecture for HPC," in *2016 IEEE Hot Chips 28 Symposium (HCS), Cupertino, CA, USA, August 21-23, 2016*. IEEE, 2016, pp. 1–31.

[3] C. Zhang, "Mars: A 64-core armv8 processor," in *2015 IEEE Hot Chips 27 Symposium (HCS), Cupertino, CA, USA, August 22-25, 2015*. IEEE, 2015, pp. 1–23.

[4] X. Liao, K. Lu, C. Yang, J. Li, Y. Yuan, M. Lai, L. Huang, P. Lu, J. Fang, J. Ren, and J. Shen, "Moving from exascale to zettascale computing: challenges and techniques," *Frontiers Inf. Technol. Electron. Eng.*, vol. 19, no. 10, pp. 1236–1244, 2018.

[5] J. Fang, C. Huang, T. Tang, and Z. Wang, "Parallel programming models for heterogeneous many-cores: a comprehensive survey," *CCF Trans. High Perform. Comput.*, vol. 2, no. 4, pp. 382–400, 2020.

[6] P. Zhang, J. Fang, C. Yang, C. Huang, T. Tang, and Z. Wang, "Optimizing streaming parallelism on heterogeneous many-core architectures," *IEEE Trans. Parallel Distributed Syst.*, vol. 31, no. 8, pp. 1878–1896, 2020.

[7] W. Gao, J. Fang, C. Xu, and C. Huang, "Dissecting the phytium 2000+ memory hierarchy via microbenchmarking," in *Advanced Computer Architecture*, D. Dong, X. Gong, C. Li, D. Li, and J. Wu, Eds. Singapore: Springer Singapore, 2020, pp. 150–162.

[8] J. Fang, X. Liao, C. Huang, and D. Dong, "Performance evaluation of memory-centric armv8 many-core architectures: A case study with phytium 2000+," *J. Comput. Sci. Technol.*, vol. 36, no. 1, pp. 33–43, 2021.

[9] D. Chen, J. Fang, C. Xu, S. Chen, and Z. Wang, "Characterizing scalability of sparse matrix-vector multiplications on phytium FT-2000+," *Int. J. Parallel Program.*, vol. 48, no. 1, pp. 80–97, 2020.

[10] D. Chen, J. Fang, S. Chen, C. Xu, and Z. Wang, "Optimizing sparse matrix-vector multiplications on an armv8-based many-core architecture," *Int. J. Parallel Program.*, vol. 47, no. 3, pp. 418–432, 2019.

[11] S. Chen, J. Fang, D. Chen, C. Xu, and Z. Wang, "Adaptive optimization of sparse matrix-vector multiplication on emerging many-core architectures," in *20th IEEE International Conference on High Performance Computing and Communications, HPCC 2018, Exeter, United Kingdom, June 28-30, 2018*. IEEE, 2018, pp. 649–658.

[12] J. Dongarra, "Report on the fujitsu fugaku system," Tech. Rep. ICL-UT-20-06, June 2020.

[13] X. You, H. Yang, Z. Luan, Y. Liu, and D. Qian, "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*, ser. Lecture Notes in Computer Science, D. Abramson and B. R. de Supinski, Eds., vol. 11416. Springer, 2019, pp. 86–105.

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

[15] "Opencl blas," https://github.com/clMathLibraries/clBLAS.

[16] "Nvidia cublas," https://developer.nvidia.com/cublas.

[17] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the ACM/IEEE Conference on Supercomputing, SC 1998, November 7-13, 1998, Orlando, FL, USA*. IEEE Computer Society, 1998, p. 38.

[18] K. Goto and R. A. van de Geijn, "High-performance implementation of the level-3 BLAS," *ACM Trans. Math. Softw.*, vol. 35, no. 1, pp. 4:1–4:14, 2008.

[19] X. Zhang, Q. Wang, and Y. Zhang, "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, 2012, pp. 684–691.

[20] F. G. V. Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, pp. 14:1–14:33, 2015.

[21] Z. Jia, A. Zlateski, F. Durand, and K. Li, "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*, A. Krall and T. R. Gross, Eds. ACM, 2018, pp. 109–123.

[22] X. Li, Y. Liang, S. Yan, L. Jia, and Y. Li, "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*, J. K. Hollingsworth and I. Keidar, Eds. ACM, 2019, pp. 229–241.

[23] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "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*, J. West and C. M. Pancake, Eds. IEEE Computer Society, 2016, pp. 981–991.

[24] J. Chen, N. Xiong, X. Liang, D. Tao, S. Li, K. Ouyang, K. Zhao, N. DeBardeleben, Q. Guan, and Z. Chen, "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*, R. Eigenmann, C. Ding, and S. A. McKee, Eds. ACM, 2019, pp. 106–116.

[25] X. Su, X. Liao, H. Jiang, C. Yang, and J. Xue, "SCP: shared cache partitioning for high-performance GEMM," *TACO*, vol. 15, no. 4, pp. 43:1–43:21, 2019.

[26] G. Frison, D. Kouzoupis, T. Sartor, A. Zanelli, and M. Diehl, "BLASFEO: basic linear algebra subroutines for embedded optimization," *ACM Trans. Math. Softw.*, vol. 44, no. 4, pp. 42:1–42:30, 2018.

[27] "Eigen v3," http://eigen.tuxfamily.org.

[28] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 12:1–12:25, 2008.

[29] F. Wang, H. Jiang, K. Zuo, X. Su, J. Xue, and C. Yang, "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, 2015, pp. 200–209.

[30] B. Marker, F. G. V. Zee, K. Goto, G. Quintana-Ortí, and R. A. van de Geijn, "Toward scalable matrix multiply on multithreaded architectures," in *Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference, Rennes, France, August 28-31, 2007, Proceedings*, ser. Lecture Notes in Computer Science, A. Kermarrec, L. Bougé, and T. Priol, Eds., vol. 4641. Springer, 2007, pp. 748–757.

[31] T. M. Smith, R. A. van de Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. V. Zee, "Anatomy of high-performance many-threaded matrix multiplication," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*. IEEE Computer Society, 2014, pp. 1049–1059.

[32] K. Kim, T. B. Costa, M. Deveci, A. M. Bradley, S. D. Hammond, M. E. Guney, and et al, "Designing vector-friendly compact BLAS and LAPACK kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 12 - 17, 2017*, B. Mohr and P. Raghavan, Eds. ACM, 2017, pp. 55:1–55:12.

[33] E. Georganas, S. Avancha, K. Banerjee, D. D. Kalamkar, G. Henry, H. Pabst, and A. Heinecke, "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, 2018, pp. 66:1–66:12.

[34] E. Georganas, K. Banerjee, D. D. Kalamkar, S. Avancha, A. Venkat, M. J. Anderson, G. Henry, H. Pabst, and A. Heinecke, "Harnessing deep learning via a single building block," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*. IEEE, 2020, pp. 222–233.

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

[36] L. Jiang, C. Yang, Y. Ao, W. Yin, W. Ma, Q. Sun, F. Liu, R. Lin, and P. Zhang, "Towards highly efficient DGEMM on the emerging SW26010 many-core processor," in *46th International Conference on Parallel Processing, ICPP 2017, Bristol, United Kingdom, August 14-17, 2017*. IEEE Computer Society, 2017, pp. 422–431.