Improving BLIS Parallelism performance through targeting loop priorities

Khanin Udomchoksakul
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, U.S.
kudomcho@andrew.cmu.edu

***Abstract*—BLIS was introduced as an instantiation of the BLAS-like library, which introduces five nested loops that handle packing on different data dimensions to different memory hierarchy levels to compute dense linear algebra operations on a multi-core system. Accelerating BLIS's performance relies on multi-threading, where distributing threads among five loops is essential to achieve high performance. As multi-core systems have become more modern and complex than ever, from dual sockets to multiple NUMA domains, applying traditional thread configurations is no longer optimal. Thus, considering new hardware topology for deriving new parallelization schemes is imperative and essential to scale up a better performance. We present our approach of determining thread-parallelization priorities on general matrix-matrix multiplication (GEMM) that takes cache memory structure, dual-socket constraint, and NUMA domain on modern multi-core systems into account. We reproduce substantially better performance than the auto-configuration generated by the BLIS framework across different architectures. Specifically, we show that our approach can deliver from 15% to 193% performance improvement.**

***Index Terms*—Linear Algebra, High Performance Computing, Multi-threading, Matrix Multiplication, BLAS, Multi-Core Processing, Machine Learning**

I. Introduction

Rises of high-performance computing have emerged throughout decades due to the needs for scientific discovery and overcoming engineering challenges. Specifically, hardware resources have become increasingly more complex and abundant in Multi-Core systems. Thus modern hardware topology plays a role in structuring memory and physical cores. Understanding the architectures of the multi-core system on specific machines is essential to accelerate computing efficiency. General matrix-matrix multiplication (GEMM) is one of the core computations within a Basic Linear Algebra Software (BLAS) responsible for various scientific computing, including machine learning applications. Being able to accelerate this computation essentially translates to faster computation time and lower power consumption required.

Historically, van de Geijin et al. [1] has widely been regarded as the most efficient BLAS implementation, usually called the GotoBlas approach. BLAS-like Library Instantiation Software [2], BLIS, is a framework that expands the existing GotoBlas implementation with two key features that differentiate it from other libraries. It has nested for loops responsible for packing different data dimensions to different levels of the memory hierarchy and a micro-kernel that acts as the smallest unit of the computation written in Assembly to optimize specific registers of a target architecture.

Recent research attempted to analyze the mechanism of each loop on BLIS in detail. T. M. Smith et al.[3] described properties and mechanism variation on each loop when the cache is either private or shared and explained how the number of iterations is a significant factor for performance. However, as resources per core are more abundant and the number of processors increases over time, we cannot expect traditional parallelization schemes to be optimal on newer hardware. Our work expands this research by 1). reexamining parallelization of each loop to best align with modern architectures where we emphasize targeting commonalities of more recent hardware across architectures and dual-socket systems 2). proposing loop priorities to decide which loop to parallelize first from a small to large number of threads on the system.

We demonstrate our approach by showing BLIS performance results on both Intel and AMD processors. Our results show a substantial performance improvement from the automatic default configuration provided by BLIS in multiple cases, giving from 15% to 193% better Gigaflops per core.

II. BLIS Implementation

Our focus on the BLIS computation is the dense matrix-matrix multiplication case $C+ = AB$, where the dimension of A, B, and C are $m \times k$, $k \times n$, and $m \times n$, respectively. To perform this computation, BLIS [4] introduces five nested for loops that iterate through problem size dimensions. Figure 1 shows the nested for loops starting with the most outer loop called the JC loop, PC loop, IC loop, JR loop, and IR loop.

BLIS has five parameters that determine the data size on each loop for targeting the different sizes of caches available. kc, mc, and nc are parameters that target cache memories, which can be modified.

mr, and nr target register memory, which can only be modified in the micro-kernel Assembly code. Low et al.[4] showed that modification of these 3 parameters can be achieved for performance tuning. However, our work assumes fixed values of five parameters on BLIS.

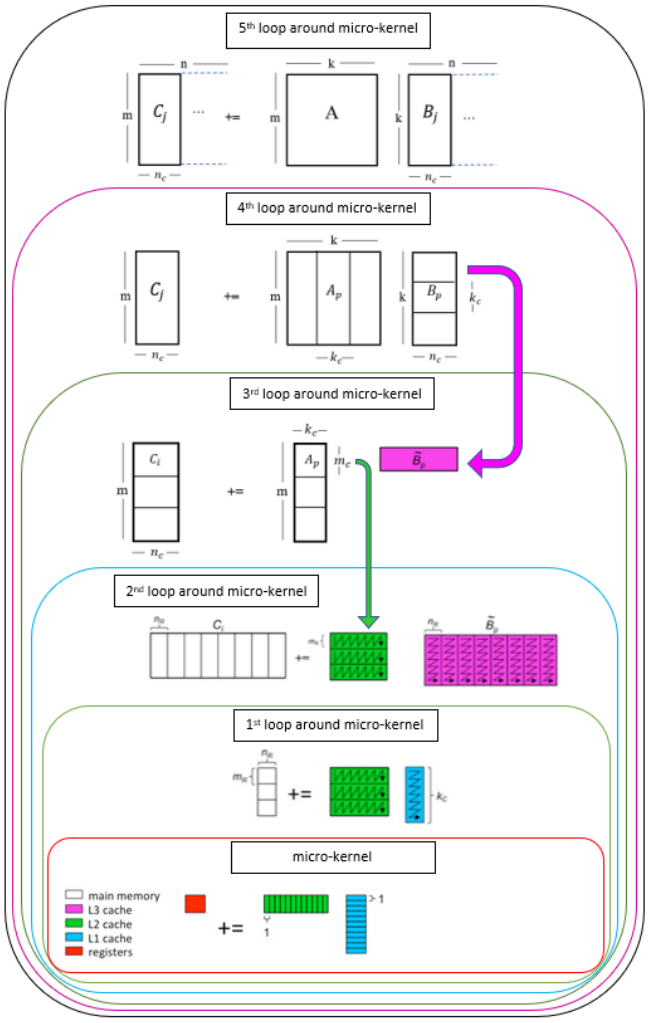

Fig. 1. BLIS 5 nested for loops diagram

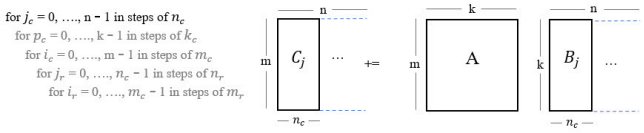

Fig. 2. Parallelizing JC loop on N dimension

A. Parallelizing 5th loop around micro-kernel indexed by JC

The JC loop partitions the matrices B and C in a column dimension with the size of parameter nc. In Figure 2, parallelizing this loop means each thread obtains an individual B column in the main memory. Each thread shares matrix A and performs the remaining loop independently until matrix C is synchronized. An essential consideration for this loop is that when the value of the n dimension after being divided by the number of threads is less than the default nc parameter value, that value becomes a new nc parameter instead.

B. Parallelizing 4th loop around micro-kernel indexed by PC

The next loop is the PC loop. Figure 3 shows that the A and the B columns are partitioned into column panels and row

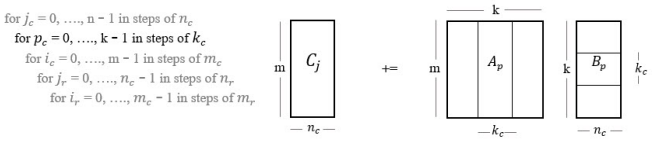

Fig. 3. Parallelizing PC loop on N dimension

panels by the size of parameter kc, respectively, to allow the sequence of rank-k updates[5] the C column panel. Then, a row-column panel of B with a size of kc by nc will be placed onto the L3 cache. Row B panel is collaboratively packed by threads on the inner loops. Parallelizing this loop means distributing individual column blocks of A and blocks of B to a new team of threads so that each new thread will have a single column block of A and row block of B. However, since this loop needs to compete to update C, race condition is not what we desire, particularly when having big sizes of A and B. Therefore, we do not consider parallelizing this loop for these two reasons.

C. Parallelizing 3rd loop around micro-kernel indexed by IC

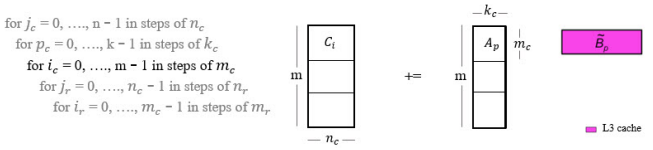

Fig. 4. Parallelizing IC loop on M dimension

Figure 4 shows an IC loop where each column of A panel is partitioned on the M dimension by the size of mc parameter, resulting in having multiple of macro-panels of A sized mc by kc. For a single core case, one macro panel of A is brought into the L2 cache. For a multi-threaded case, multiple macro panels of A are brought to individual L2 caches on each thread because all threads have private L1 and L2 caches.

D. Parallelizing 2nd loop around micro-kernel indexed by JR

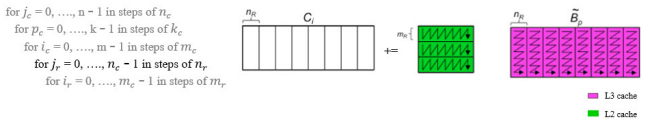

Fig. 5. Parallelizing JR loop on nc dimension

There is a JR loop that partitions a row column of B from the L3 cache by the size of nr parameter on the nc dimension on Figure 5. It brings a piece of data called sliver B sized nr by kc to the L1 cache. This sliver is used to compute the micro-kernel at the register level.

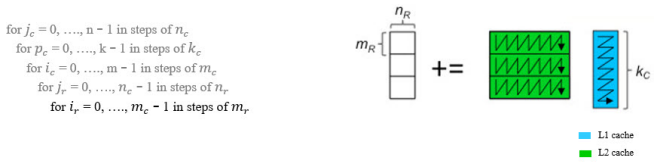

Fig. 6. Parallelizing IR loop on mc dimension

E. Parallelizing 1st loop around micro-kernel indexed by IR

The last loop around the micro-kernel is an IR loop shown in Figure 6. It partitions each macro panel of A by the size of mr parameter on the mc dimension, producing a sliver of A sized mr by kc that resides on the L2 cache. This sliver will also be used to compute the micro-kernel. After this loop is the micro-kernel that computes a sequence of rank-1 updates (outer products) from columns from the sliver A and the sliver of B, producing a subset of block C sized mr by nr.

F. Multi-threading on BLIS framework

BLIS uses either OpenMP or Pthreads to allow Multi-core parallelism. Our scenario uses OpenMP because it allows some control to bind threads to cores by using an environment variable before executing the program. We do binding to ensure that data placement on each thread will not move to other threads' cache memory, which could cause cache misses and impact performance overall.

To determine how many threads are on each loop is to specify the number on that loop. Parallelizing BLIS is hierarchical, meaning that if we parallelize multiple loops, the total number of threads is the product of the threads on each loop. Also, parallelizing multiple loops means that the outer loop will have a team of threads on the inner loops. For instance, stating $JC = 2$, $IC = 8$, $JR = 1$ means that we specify 2 threads on the JC loop and 8 threads on the IC loop, which indicates that each thread on the JC loop will have a group thread of 8 in parallel. Thus the total number of threads used in this configuration is 16. Specifying multi-threading can be achieved automatically or manually.

In a manual way, we determine the number of threads on each loop before runtime. In an automatic way, we specify how many total physical threads are available on this system to BLIS, and BLIS will generate the configuration based on the number of threads and problem sizes.

III. Our approach on Parallelizing BLIS

We take our understanding of architecture topology, advantages of hardware on processors, and the iterations of each for loop to derive for loop priorities to parallelize. Our approach helps select which loop to parallelize first, given the limited number of threads. At the same time, our approach also helps determine which loops we should prioritize when working with a large number of threads.

A. Priority 1: Bringing data to cache level memory

Memory subsystems on multi-core systems are hierarchical, and caches residing closer to the registers are generally faster but smaller. For example, on the AMD EPYC 7742 processor, the latency on the L1, L2, and L3 caches and DRAM memory are 1.18ns, 3.86ns, 10.87ns, and 122ns, respectively [6]. At the same time, modern architectures now have private L1 and L2 caches on each core, allowing us to maximize the use of a large number of private caches.

IC loop is the first loop that stores macro A panels to the private L2 cache and collaboratively packs the B column panel from DRAM memory to the L3 cache. The number of iterations to pack macro$A$ panels will increase as the dimension M is larger. This loop provides an opportunity to reduce the iterations of packing macro A panels and maximize the number of private L2 caches available. Therefore, It is reasonable to have many threads bringing data to cache-level memory to reduce iterations of packing macro A panels from DRAM memory. Packing from memory is also expensive, so parallelizing this loop helps reduce the time packing data to the cache memory. Because each thread has private L1 and L2 caches, scaling out this approach is also favorable. Therefore, the IC loop should be the first to be parallelized.

B. Priority 2: Parallelizing on packing data to cache

The next consideration is whether we can parallelize the IC loop using all available threads. We provide several factors that will determine the IC parallelization.

First, more time will be required to transfer data when only parallelizing IC loop on the dual socket. When parallelizing only IC loop with many threads that exceed the total number of threads on one socket, the second socket will need to communicate with the first socket to obtain a column of B. Thus, the column of B resides in the socket that first spawns threads, which causes communication overhead over cross socket. For this reason, we do not parallelize the IC loop with threads exceeding the number of threads on one socket.

Second, depending on the M dimension, the IC loop iteration is M/mc. On the system level, having each thread perform some iterations allows the system to reach a steady state where each thread can work at its full capacity. Therefore, it is optional to target parallelizing the IC loop such that each thread only has one iteration to pack A. Unless, there are an abundant numbers of threads that can be used to scale up packing A close to a single iteration per thread. For these reasons, we consider incorporating the IC loop with the JC loop for a performance boost.

Collaboratively parallelizing IC and JC loops can help speed up because we use the JC loop to partition B into columns, allowing multiple teams of threads to work independently. In contrast, each team packs macro A panels in parallel, and this helps reduce the iterations on the JC loop that BLIS needs to perform on the N dimension sequentially. However, because the IC loop does pack macro A panels from DRAM to cache memory, we still desire to maintain the majority of threads to be on the IC loop. Thus, the ratio of the IC loop to the JC loop will be that the IC loop has more threads

than those in the JC loop. Therefore, we will use JC loop to parallelize as the second priority.

C. Priority 3: Splitting data to the faster memory

The following loop we would like to consider is the JR loop. Parallelizing this loop splits the row B panel into a number of smaller panels based on the number of threads on the JR loop. Each thread has a different row B panel with a smaller iteration on the nc dimension, so the thread has less iteration to bring the sliver of B to its L1 cache. This results in each thread bringing different slivers of B in parallel. Also, partitioning B into columns to the L1 cache is less expensive than previous loops because it is on the cache level memory, where latency is significantly less than those in the main memory to the L3 cache. Therefore, we consider parallelizing this loop when we sufficiently allocate threads for JC and IC loops.

D. Priority 4: Reducing iterations on the macro A panels

For the IR loop, parallelizing is beneficial based on whether double buffering occurs on the micro-kernel. Because there are enough registers on the Intel machine, it allows us to hide the computational cost. Thus, we can bring all data from the L2 to registers without stalling. Traditionally, we often desire to size the data such that part of the micro panel of A and B are stored on the L1. However, because double buffering [7] occurs, it allows us to bring data from the L2 cache directly, so the space on the L1 cache can be used to store the larger sliver B instead. Therefore, enlarging the size of kc parameter can achieve more speedup. However, modification of the BLIS parameters is not our focus in this work. Thus, it should be discussed in future work instead. Parallelizing the IR loop may be beneficial when no double buffering occurs on the memory or modification of the kc parameter needs to be calibrated for better performance. Therefore, it is the fourth priority for our approach to parallelization.

E. Summary on our approach

Once we understand the importance of each priority, our approach will work as follows: 1). on a few threads, we will parallelize IC loop and then JC loop such that the majority of the threads remain on the IC loop. 2). When packing A and B are sufficient, we either scale up the ratio of JC and IC loops or parallelize JR loop. 3). Once JC, IC and JR loops are sufficiently parallelized, we can parallelize IR loop.

IV. Experimentation

Our experiments are on the AMD EPYC 7742 processor with 128 physical cores dual socket system and 8 NUMA domains system, and Intel Xeon 6128 processor with 12 physical cores dual socket and 2 NUMA domains system. Details on the hardware properties are presented in Table 1. We use square matrices starting from a problem size of 240 to 6000, incremented by 240. For every problem size, we ensured that we set the number of the experiment run by at least 10 for consistent performance. We also isolated the system to ensure no external processes were active during our experiment.

For each machine we experimented with, we will show parallelizing BLIS from a small to a large number of threads to observe that parallelism priorities are essential to consider in multi-core processing. For each performance plot, we show problem sizes on the X axis and Gigaflops per core on the Y axis. We present the default configuration from the BLIS and our configurations that reflect our reasoning. We use double precision for experimenting on BLIS.

TABLE I
PROCESSOR HARDWARE DETAILS

	Intel Xeon 6128	***AMD EPYC 7742***
total cores	12	128
total sockest	2	2
L1 cache / core	32 KB	32 KB
L1 cache / core	1 MB	512 KB
total L3 cache	38.5 MB	512 MB
Core frequency	2.9-3.4 GHz	2.25-3 GHz
SIMD Instruction	AVX-512	AVX-2
mc	240	96
kc	256	256
nc	3752	4080
mr	14	8
nr	16	6

V. Intel Xeon 6128 Results

On this machine [8], with BLIS parameters, data on the L1 cache will utilize 28 KB and 480 KB on the L2 cache, so this ensures that data will not exceed the cache capacity. The maximum data usage on the L3 cache is 7508 KB, but because the L3 cache is shared, this will not exceed the cache. We need to present an expected single-core performance and the empirical single-core performance we obtained from running on the system.

For the expected single core, we have 2 fused multiply-add (FMA) for floating-point operations per instruction, from addition and multiplication. The FMA instruction has a throughput of 2 instructions per cycle, and this computation uses AVX-512 as the SIMD instruction, consuming 8 64-bit couple registers per operation. Our ideal performance for each thread should be approximately 32 Flop/cycle. With a core frequency is 2.9 GHz for multi-threading and AVX-512 enabled, we expect approximately 92.8 Gigaflops per core. On average, the empirical single-core performance on this machine is 93 Gigaflops per core.

A. Intel Xeon 6128 using 6 threads

We demonstrate 6 threads on the single socket in Figure 7. With 6 threads, we have 2 options for factorization, which are 2 and 3, and 1 and 6. Using our method, we prioritize IC loop first and the JC loop second. We want factorization of 2 and 3, so 3 threads parallelize on the IC loop while 2 threads parallelize on the JC loop. Using 6 threads on the IC loop only can be effective as well, given the limited threads available. From the dimensions of 240 to 3600, configuration $JC = 1$, $IC = 6$, $JR = 1$, $IR = 1$ peaked 83 Gigaflops

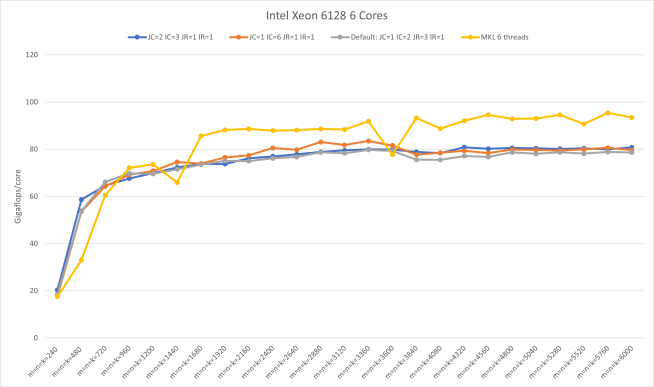

Fig. 7. 6 threads paralllesim using JC and IC loops on Intel Xeon 6128

per core, approximately 4-5 Gigaflops better than the other 2 configurations. This is due to having more threads to help pack data from DRAM to the cache memory.

Configuration $JC = 1$, $IC = 6$, $JR = 1$, and $IR = 1$ also have another advantage that packing data to cache quickly can speed up the computation once the data is closer to registers at once. Hence, the performance is instantly higher shown at the beginning of Figure 7. However, once the problem size of N exceeds the nc parameter, configurations$JC = 1$, $IC = 6$, $JR = 1$, $IR = 1$ and $JC = 1$, $IC = 2$, $JR = 3$, $IR = 1$ will have another iteration on the JC loop as they have the remainder data of $N - nc$. Thus the performance for both configurations are reduced, while configuration $JC = 2$, $IC = 3$, $JR = 1$, $IR = 1$ already partitions B columns initially, so it does not compromise the performance dip from this. As the problem size exceeds nc onwards, configuration $JC = 2$, $IC = 3$, $JR = 1$, $IR = 1$ can scale up performance to saturation at 80 Gigaflops per core while the other two configurations reach 79 Gigaflops.

We compare the BLIS results with the MKL performance for 6 threads. MKL achieves 95 Gigaflops per core, a 13% better performance than the peak of BLIS.

B. Intel Xeon 6128 using 12 threads

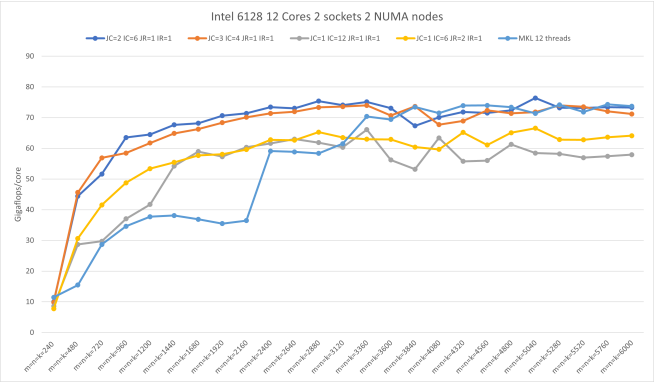

Fig. 8. 12 threads paralllesim using JC and IC loops on Intel Xeon 6128

Figure 8 shows that with 12 threads, we have more factorization pairs which are 2 and 6, 3 and 4, and 1 and 12. We do not want to parallelize 12 threads on the IC loop due to threads communication overhead across the sockets for requesting the B row panel. Configuration $JC = 2$, $IC = 6$, $JR = 1$, $IR = 1$ is optimal because we allocate most threads to the IC loop to pack data to cache memory from DRAM. Configuration $JC = 3$, $IC = 4$, $JR = 1$, $IR = 1$ can be comparable to $JC = 2$, $IC = 6$, $JR = 1$, $IR = 1$, despite losing some resources for packing A for partitioning more B columns.

Results on Figure 8 show that our prioritization approach gives the best result, peaking at 77 Gigaflops per core, approximately 15% better performance than the default configuration at the same problem size. Particularly on configuration $JC = 2$, $IC = 6$, $JR = 1$, $IR = 1$, rapid performance increase at the starting problem sizes is benefited from both packing pack multiple macro A panels in parallel and partitioning B columns into 2 that allows this occurrence in parallel.

Configuration $JC = 3$, $IC = 4$, $JR = 1$, $IR = 1$ gives a comparable performance to configuration $JC = 2$, $IC = 6$, $JR = 1$, $IR = 1$, but is slightly below throughout most problem sizes. Configuration $JC = 1$, $IC = 12$, $JR = 1$, $IR = 1$ shows a relatively large fluctuation and poor performance due to communication overhead across two sockets.

Configuration $JC = 1$, $IC = 6$, $JR = 2$, $IR = 1$ shows a consistent performance increase. However, its peak value is below the configuration $JC = 2$, $IC = 6$, $JR = 1$, $IR = 1$ and $JC = 3$, $IC = 4$, $JR = 1$, $IR = 1$ because it only partitions the B row panel and only maintains one threads group on the IC loop. The MKL performance peaks at 74 Gigaflops per core on this problem size range, a 4% difference from the peak of the best performing BLIS configuration.

We observe that MKL's Gigaflops per core is more significantly dropped as more cores are being used, from 95 Gigaflops per core on 6 threads to 74 Gigaflops per core on 12 threads which is a 12% drop, while BLIS shrinks from 83 to 77 Gigaflops per core respectively. The dual-socket effect outweighs the NUMA domain effect in this system because each socket contains one NUMA domain, so we cannot measure the effects of NUMA on the performance.

VI. AMD EPYC 7742 Results

This machine has 64 physical cores and 4 NUMA domains on a single socket such that each NUMA node has 16 threads [9]. With 2 sockets, there are 128 physical cores available and 8 NUMA domains. The 512 MB L3 cache is distributed so 4 threads have their own 16 MB L3 cache pool. The base core frequency is 2.25 GHz, but the boost frequency can reach 3.0 GHz. BLIS parameters for our expeirment are kc=256, mc=96, nr=8, mr=6, nc=4080.

For the expected single core, we have 2 fused multiply-add (FMA) for floating-point operations per instruction, from addition and multiplication. The FMA instruction has a throughput of 2 instructions per cycle, and this computation uses AVX-2 as the SIMD instruction, consuming four 64-bit couple registers per operation. Our ideal performance for each thread should be approximately 16 Flop/cycle.

With core frequency between 2.25 to 3.0 GHz for multi-threading, we expect approximately 36-48 Gigaflops per core. Our empirical performance on average throughout the problem sizes is 42.57 Gigaflops per core. We will demonstrate our approach on 8, 16, 32, 64, and 128 threads scenarios.

A. AMD EPYC 7742 using 8 threads

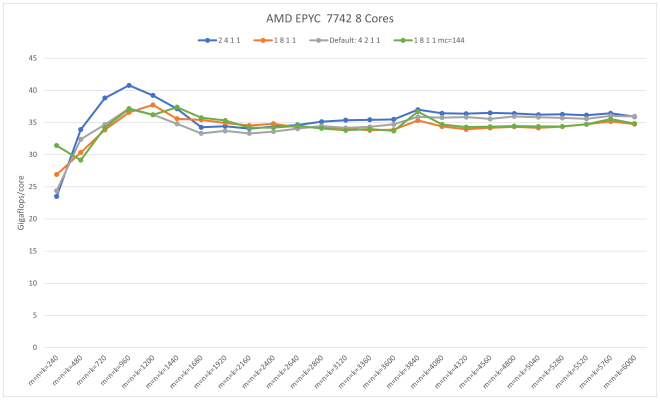

Fig. 9. 8 threads paralllesim using JC and IC loops on AMD EPYC 7742

We show 3 configurations which are $JC = 1$, $IC = 8$, $JR = 1$, $IR = 1$, $JC = 2$, $IC = 4$, $JR = 1$, $IR = 1$, and the default $JC = 4$, $IC = 2$, $JR = 1$, $IR = 1$ for 8 threads on the Figure 9. Configuration $JC = 2$, $IC = 4$, $JR = 1$, $IR = 1$ performs the best overall because it partitions the B matrix into 2 columns, and each column has the thread of 4 to pack different macro A panels to the individual L2 caches in parallel.

Performance on the configuration $JC = 1$, $IC = 8$, $JR = 1$, $IR = 1$ is slightly lower for 2 reasons. First, it does not split the column of B to allow the threads group to work on different B columns in parallel. Second, the parameter mc on this processor is smaller than the mc from the Skylake processor, which results in more iterations to pack macro A panels that are smaller than those of the Intel processor.

The default configuration lies between the two configurations, which shows comparable performance to the configuration $JC = 4$, $IC = 2$, $JR = 1$, $IR = 1$ at the large problem size range but lower on the small problem range.

B. AMD EPYC 7742 using 16 threads

Figure 10 shows the 16 threads parallelization. We again show 3 configurations: $JC = 2$, $IC = 8$, $JR = 1$, $IR = 1$, $JC = 1$, $IC = 16$, $JR = 1$, $IR = 1$, and the default configuration $JC = 4$, $IC = 4$, $JR = 1$, $IR = 1$. We select $JC = 2$, $IC = 8$, $JR = 1$, $IR = 1$ based on our approach from section 3, and $JC = 1$, $IC = 16$, $JR = 1$, $IR = 1$ to show that the effects of the cross socket on the Intel Xeon 6128 will not happen on this scenario.

We demonstrate that our approach on the configuration $JC = 2$, $IC = 8$, $JR = 1$, $IR = 1$ gives the best performance overall with a peak of 36.7 Gigaflops per core. The configuration $JC = 1$, $IC = 16$, $JR = 1$, $IR = 1$ does

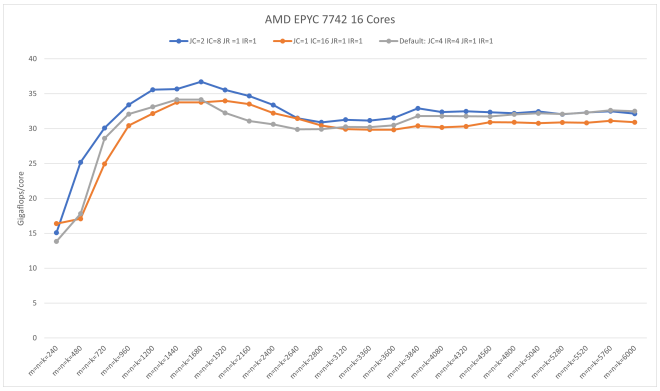

Fig. 10. 16 threads paralllesim using JC and IC loops on AMD EPYC 7742

not encounter performance degradation for the same reason as the configuration $JC = 1$, $IC = 12$, $JR = 1$, $IR = 1$ on the Xeon 6128 processor because this parallelization scenario is on the single socket.

This indicates that parallelizing the IC loop with large number of threads can be plausible as long as all threads on the IC loop remain on the same socket. The default configuration distributes 16 threads on the JC and IC loops equally, so the performance is slightly lower than our approach at the small problem size range but similar to the large problem range.

C. AMD EPYC 7742 using 32 threads

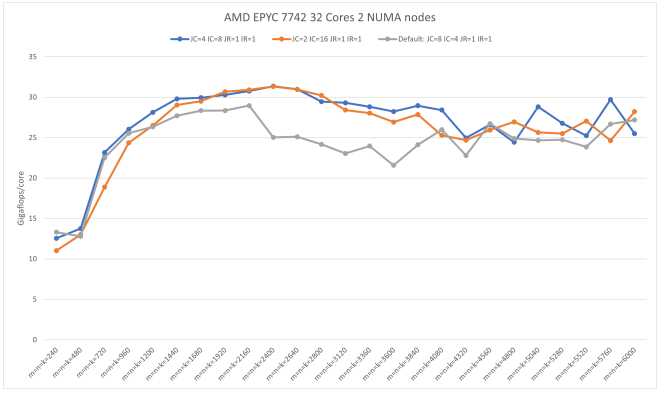

Fig. 11. 32 threads paralllesim using JC and IC loops on AMD EPYC 7742 on 2 NUMA nodes

We have 3 configurations on Figure 11 for 32 threads with 2 NUMA domains. We select configuration $JC = 4$, $IC = 8$, $JR = 1$, $IR = 1$ because we maintain priorities for the 32 threads mentioned on section 3. This configuration will spread B matrix into 4 B columns while each column has 8 threads to pack macro A panels to the individual L2 caches. This configuration has multiple B columns to each NUMA domain while maintaining the best performance throughout most problem sizes, peaking at 31.37 Gigaflops per core.

We include the configuration $JC = 2$, $IC = 16$, $JR = 1$, $IR = 1$ to show a performance tradeoff between the JC loop and the IC loop. This configuration shows a comparable performance where it peaks at 31.33 Gigaflops at the small

problem size range and maintains between 26-28 Gigaflops per core for the remaining problem sizes.

However, it splits B columns less than the configuration $JC = 4$, $IC = 8$, $JR = 1$, $IR = 1$ by 2 columns to parallelize more on the IC loop, which results in having a lower performance at the small problem range.

The default configuration parallelizes the JC loop more than IC loop on the 2 NUMA domains, and it performs the worst among 3 configurations.

D. AMD EPYC 7742 using 64 threads

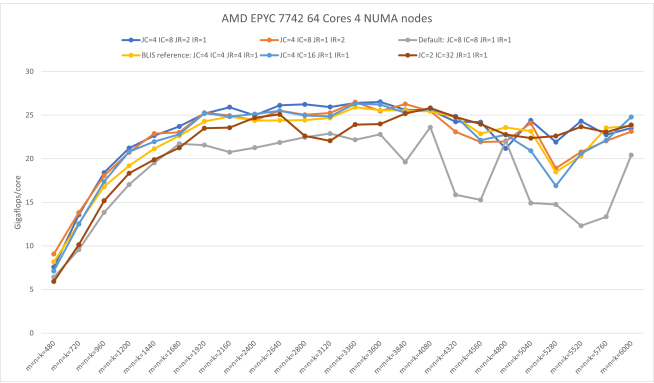

Fig. 12. 64 threads paralllesim using JC, IC, JR, and IR loops on AMD EPYC 7742 on 4 NUMA nodes

We have more factorization choices for 64 threads in Figure 12. Factorization options are 8, 4, and 2, 4, 4, and 4, 16 and 4, 8 and 8, 32 and 2, and 64 and 1. We do not select factorizations 1 and 64, 4, 4, and 4 because they do not follow our priorities approach. We first discuss configurations prioritizing the IC loop followed by the JC loop. Within this group, we will discuss several variations on further parallelizing other loops.

Configurations that follow our first two priorities all have closely similar performance throughout the problem size range, except the configuration $JC = 2$, $IC = 32$, $JR = 1$, $IR = 1$ whose performance is lower from the problem size of 480 to 3600 for two primary reasons.

First, it only partitions matrix B into columns less than other configurations in the same group, which reduces the degree of parallel packing A, given 64 threads available. Second, there is not enough work for each thread to perform because it takes a problem size of 3600 to have a single iteration, while the other configurations take smaller problem sizes to obtain the steady state.

These two factors contribute to this particular configuration requiring larger problem sizes to gain a comparable performance, which results in lower performance at the problem range 480 to 3600 by 2-3 Gigaflops per core than other configurations.

Also, when there are 32 threads on the IC loop, the first 32 threads have the first B column residing in the first NUMA domain memory. This forces the 16 threads on the second NUMA domain to get macro A panels on the first NUMA domain and parts of row B panels to the second NUMA domain for the problem size 4080 onwards. This configuration does not get performance degradation because despite having multiple NUMA domains, they all reside in the same socket.

For other configurations within the group, we will discuss the configuration $JC = 4$, $IC = 16$, $JR = 1$, $IR = 1$, $JC = 4$, $IC = 8$, $JR = 1$, $IR = 2$, and $JC = 4$, $IC = 8$, $JR = 2$, $IR = 1$.

Configuration $JC = 4$, $IC = 8$, $JR = 2$, $IR = 1$ takes 3 priorities from our approach into account, showing the best performance overall because this avoids constraints that previous configuration encounters. It partitions the matrix B into 4 columns, each with 8 threads, to pack different macro A panels onto individual L2 caches. Parallelizing the IC loop with 8 threads leaves each thread 7.81 iterations to pack A, which allows exercising for a steady state. This configuration also has 2 threads from each thread on the IC loop to partition the B row panel by $(N/4)/2$, reducing iterations to load the sliver B onto the L1 cache.

Configuration $JC = 4$, $IC = 8$, $JR = 1$, $IR = 2$ benefits from partitioning four B columns while parallelizing the IC loop by 8 threads in parallel. This allows for reaching a faster steady state than the previous configurations. BLIS on the AMD 7742 processor does not have double buffering, so parallelizing the IR loop can be beneficial. Therefore, it performs similarly to the configuration $JC = 4$, $IC = 8$, $JR = 2$, $IR = 1$ except for some large problem sizes that are affected by having more iterations on the row B panel.

Configuration $JC = 4$, $IC = 16$, $JR = 1$, $IR = 1$ encounters the same reason of requiring large problem sizes for each thread on the IC loop to reach a steady state but less degree than having 32 threads on the IC loop. Once a thread on the IC loop has more than 1 iteration to pack, this configuration performs similarly to the $JC = 4$, $IC = 8$, $JR = 1$, $IR = 2$, and $JC = 4$, $IC = 8$, $JR = 2$, $IR = 1$.

We refer to BLIS configuration $JC = 4$, $IC = 4$, $JR = 4$, $IR = 1$ from the BLIS official Github with the same processor for performance comparison purposes. This configuration equally parallelizes the JC, IC, JR loops. The difference between this configuration and our approach is that it parallelizes 4 threads on the IC loop while allocating the rest to the JR loop. This configuration takes more time to pack macro A panels to the L2 from the main memory, resulting in 1-2 lower Gigaflops than the configuration JC=4 IC=8 JR=2 IR=1. This shows that elapsed time for one iteration on the IC loop is higher than on the JR loop.

E. AMD EPYC 7742 using 128 threads

We incorporate our approach from section 3 and insights from the previous subsections to demonstrate performance improvement shown in Figure 13. We again have factorization options which are 4, 4, and 8 and 2, 2, 4, 8, and 2, 4, and 16, 2, 8 and 8, 2, and 64, and 4 and 32.

Our priorities approach suggests using 2, 4, and 16 and 4 and 32 factorization. We derive our configuration as $JC = 4$, $IC = 16$, $JR = 2$, $IR = 1$ as the most aligned with our priorities approach. We also show the performance of the configuration $JC = 4$, $IC = 32$, $JR = 1$, $IR = 1$ to demonstrate that some performance fluctuation can occur

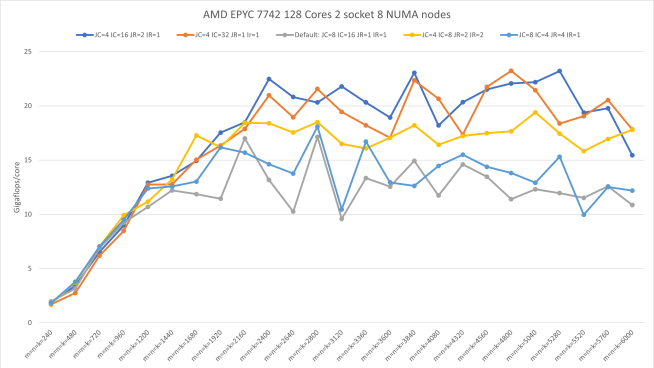

Fig. 13. 128 threads paralllesim using JC, IC, JR, and IR loops on AMD EPYC 7742 on 8 NUMA nodes

when not parallelizing the JR loop. Configuration $JC = 4$, $IC = 8$, $JR = 2$, $IR = 2$ is to demonstrate the effects of parallelizing the IR loop instead scaling up the threads on the IC loop. Other two configurations are the default configuration which is $JC = 8$, $IC = 16$, $JR = 1$, $IR = 1$ and the BLIS reference configuration $JC = 8$, $IC = 4$, $JR = 4$, $IR = 1$. Figure 13 shows that our configuration $JC = 4, IC = 16, JR = 2, IR = 1$ performs well throughout problem sizes with a peak of 23.3 Gigaflops per core, 193% better performance than the default configuration at the same problem size. Our configuration can encounter some stalling when each group of 16 threads attempts to access the same index of row B panel. Thus slight fluctuation is present. This configuration allocates threads to not communicate across sockets, so performance degradation due to this effect will not occur. We also show in subsection D that having threads cross to different NUMA domains on the same socket will not degrade performance.

The configuration $JC = 4$, $IC = 32$, $JR = 1$, $IR = 1$ performs similarly to our selected configuration but requires larger problem sizes for the threads on the IC loop to have a steady state. Also, having 32 threads to access the same index of the row B panel can cause stalling among threads which requires more time. Therefore, the fluctuation also occurs in Figure 13.

Configuration $JC = 4$, $IC = 8$, $JR = 2$, $IR = 2$ shows a lower peak performance than the previous configurations but provides a steady result, with a peak of 19.3 Gigaflops per core. This is because it parallelizes 8 threads on the IC loop instead of 16. Thus, the degree of parallel packing A is lower. However, because the memory contention from 8 threads accessing the row B panel is lower than the previous configurations, performance fluctuation is less, providing consistent performance throughout the problem sizes.

The default and BLIS reference configurations show the worst performance compared to the other 3 configurations. We see that parallelizing 8 threads on the JC loop give poor performance as we increase from 32 to 128 threads. This issue should be further studied in future work in detail.

VII. Conclusion

In this paper, we showed that our approach to prioritizing the for loops on BLIS yields relatively better performance than the default configuration. This approach helps determine which loop to parallelize first from a small to large number of threads. We discussed that the number of iterations on each loop is one of many factors to consider for parallelism. However, the hardware properties from steady state, memory latency, and data movement are also essential to obtain a good performance.

We showed that this approach can be transferable to different architectures where standard hardware, such as private caches and memory sizes, are available. There are opportunities to explore for future work, from tuning the BLIS parameters for parallelism, diagnosing system behaviors to reach closer to the peak, creating an automatic parallelism scheme generator based on this approach, or to further study in the NUMA domain. Overall, this is a progressive milestone that extends the existing work and can be carried on in the future.

Acknowledgment

We thank Dr. Tze Meng Low and his Ph.D students Elliot Binder, Nicholai Tukanov, Upasana Sridhar, and Maia Blanco for providing an expertise in BLIS framework and general supervision. We also thank CMKL University for providing the AMD processor on a DGX A100 cluster and the Intel Devcloud organization for the Intel processor to conduct experiments.

References

[1] R. van de Geijn and K. Goto, "Anatomy of high-performance matrix multiplication kazushige goto, robert a. van de geijn acm transactions on mathematical software (toms), 2008," *ACM Transactions on Mathematical Software*, vol. 34, p. Article 12, 05 2008.

[2] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Transactions on Mathematical Software*, vol. 41, no. 3, pp. 14:1–14:33, June 2015. [Online]. Available: https://doi.acm.org/10.1145/2764454

[3] T. M. Smith, R. A. van de Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee, "Anatomy of high-performance many-threaded matrix multiplication," in *28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014)*, 2014. [Online]. Available: https://doi.org/10.1109/IPDPS.2014.110

[4] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Ortí, "Analytical modeling is enough for high-performance BLIS," *ACM Transactions on Mathematical Software*, vol. 43, no. 2, pp. 12:1–12:18, August 2016. [Online]. Available: https://doi.acm.org/10.1145/2925987

[5] B. Kågström, P. Ling, and C. van Loan, "Gemm-based level 3 blas: High-performance model implementations and performance evaluation benchmark," *ACM Trans. Math. Softw.*, vol. 24, no. 3, p. 268–302, sep 1998. [Online]. Available: https://doi.org/10.1145/292395.292412

[6] "Amd rome processors," July 2022. [Online]. Available: https://www.nas.nasa.gov/hecc/support/kb/amd-rome-processors_658.html

[7] J. C. Sancho and D. J. Kerbyson, "Analysis of double buffering on two different multicore architectures: Quad-core opteron and the cell-be," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–12.

[8] "Intel xeon 6128 processor." [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/120482/intel-xeon-gold-6128-processor-19-25m-cache-3-40-ghz.html

[9] "Amd epyc 7742 processor." [Online]. Available: https://www.amd.com/en/products/cpu/amd-epyc-7742