

# Cache Optimization and Performance Modeling of Batched, Small, and Rectangular Matrix Multiplication on Intel, AMD, and Fujitsu Processors

SAMEER DESHMUKH, School of Computing, Tokyo Institute of Technology, AIST, Japan

RIO YOKOTA, School of Computing, Tokyo Institute of Technology, AIST, Japan

GEORGE BOSILCA, Innovative Computing Laboratory, University of Tennessee at Knoxville, USA

Factorization and multiplication of dense matrices and tensors are critical, yet extremely expensive pieces of the scientific toolbox. Careful use of low rank approximation can drastically reduce the computation and memory requirements of these operations. In addition to a lower arithmetic complexity, such methods can, by their structure, be designed to efficiently exploit modern hardware architectures. The majority of existing work relies on batched BLAS libraries to handle the computation of many small dense matrices. We show that through careful analysis of the cache utilization, register accumulation using SIMD registers and a redesign of the implementation, one can achieve significantly higher throughput for these types of batched low-rank matrices across a large range of block and batch sizes. We test our algorithm on 3 CPUs using diverse ISAs – the Fujitsu A64FX using ARM SVE, the Intel Xeon 6148 using AVX-512 and AMD EPYC 7502 using AVX-2, and show that our new batching methodology is able to obtain more than twice the throughput of vendor optimized libraries for all CPU architectures and problem sizes.

CCS Concepts: • **Theory of computation** → **Shared memory algorithms**; • **Computer systems organization** → **Multicore architectures**; • **Computing methodologies** → *Linear algebra algorithms*; **Shared memory algorithms**.

Additional Key Words and Phrases: Low-rank matrix multiplication, batched matrix multiplication, cache blocking, performance modeling

## ACM Reference Format:

Sameer Deshmukh, Rio Yokota, and George Bosilca. 2023. Cache Optimization and Performance Modeling of Batched, Small, and Rectangular Matrix Multiplication on Intel, AMD, and Fujitsu Processors. *J. ACM* 1, 1 (November 2023), 31 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

## 1 INTRODUCTION

Large dense matrices and tensors appear in applications such as Boundary integral methods, machine learning, computational finance and multivariate regression. While direct multiplication and factorization of such data sets is computationally expensive, careful use of low rank approximation techniques can drastically reduce the compute and memory cost of these methods with a controllable decrease in accuracy. Examples of such applications are the use of hierarchical matrices [26]

---

Authors' addresses: Sameer Deshmukh, [deshmukh.s.aa@m.titech.ac.jp](mailto:deshmukh.s.aa@m.titech.ac.jp), School of Computing, Tokyo Institute of Technology, AIST, Ishikawadai Bldg. 9, 2-12-1 Ookayama, Meguro-ku, Tokyo, Tokyo, Japan, 152-8550; Rio Yokota, [rioyokota@gsic.titech.ac.jp](mailto:rioyokota@gsic.titech.ac.jp), School of Computing, Tokyo Institute of Technology, AIST, Ishikawadai Bldg. 9, 2-12-1 Ookayama, Meguro-ku, Tokyo, Tokyo, Japan, 152-8550; George Bosilca, [bosilca@icl.utk.co.us](mailto:bosilca@icl.utk.co.us), Innovative Computing Laboratory, University of Tennessee at Knoxville, Claxton 308C, 1122 Volunteer Blvd, Knoxville, USA, 37996.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2023 Association for Computing Machinery.

0004-5411/2023/11-ART \$15.00

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

for factorization of large dense matrices, tensor decomposition for multilinear systems [9] and FMM in Deep Learning [44].

Maximizing the computational efficiency of such methods has its unique challenges, different from both dense matrices and sparse matrices. On one hand, it shares some of the challenges of sparse matrices, where the amount of arithmetic operations (Flops) per loaded data (Bytes) decreases with the rank of the low-rank blocks. On the other hand, it is different from sparse matrices in the sense that the low-rank blocks are not completely sparse but consist of small dense matrices, providing some opportunities for data reuse and prefetch. The resulting fine-grain regularity allows these methods to more efficiently utilize SIMD operations compared to the efficacy of sparse matrices. Optimizing the throughput of these structured low-rank matrices, however, is not an area that has been investigated sufficiently in the literature. We have attempted to address this gap by manually writing our own matrix multiplication kernels for low-rank matrices that greatly outperform vendor optimized libraries in respect to this particular issue.

This work focuses on optimizing the inner kernel for this new type of problem, where the matrix is neither dense nor sparse. As such, this work runs counter to other existing work that focus more on the parallel scalability of structured low-rank matrices. Although [10, 13, 47, 50, 62] report hierarchical matrix factorization on many hundreds of nodes, the computation shows sub-optimal resource utilization mainly attributed to library routines that are not efficiently tuned for handling the memory bound kernels of such factorization routines. We revisit this, and address the improvement of the efficiency of the low rank kernels that form a core of the computation.

In this paper we propose a new technique for optimizing a central component of structured low-rank matrices, the low rank matrix multiplication. Our technique performs batched computation of low rank matrices and shows, through a careful utilization of the different levels of cache, more efficiency than vendor optimized math libraries such as MKL, AMD-BLIS and SSL-2 (Scientific Software Library) for a variety of block and batch sizes. Thus, when compared to vendor libraries, our method shows stronger scaling for a shared memory execution. Specifically, we make two contributions in this paper:

- (1) An improved algorithm for batched computation of low rank matrix multiplication, that can achieve more than  $2x$  greater throughput than vendor optimized libraries for all the CPU architectures and problem sizes tested;
- (2) Techniques for optimization and performance validation of low level kernels with extensive use of the ECM [31] (Execution-Cache-Memory) performance model.

The rest of this paper is organized as follows. In Section 2 we review the existing literature on low rank matrices and concretely define the low rank multiplication operation optimized in this paper. In Section 3 we review the current state of the art in obtaining optimal chip performance, along with various performance modeling methodologies for guiding the development of high performance implementations. Section 4 describes the new algorithm and proposed cache blocking methodology. Section 5 reviews the applicability of the ECM performance model for our problem on the tested CPUs and demonstrates that use of the ECM model plays a pivotal role in achieving the best possible performance from a given CPU. Section 6 contains a detailed analysis of each kernel within our algorithm and presents the optimization on each of our target CPUs using the ECM performance model. We then provide the results of our method for our target CPUs compared to various vendor-optimized libraries in Section 7 and finally conclude the paper in Section 8.

## 2 LOW RANK MULTIPLICATION

The low rank approximation of a dense matrix allows capturing the most significant row and column bases of the dense matrix. If the singular values of the dense matrix (typically obtained

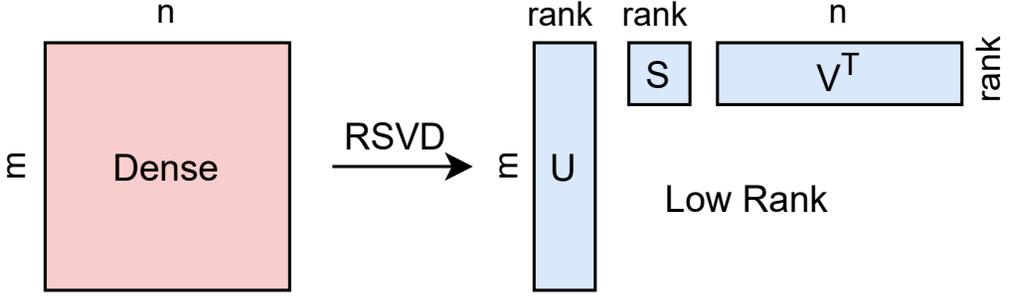


Fig. 1. Representation of a low rank matrix using an algebraic method such as the Randomized Singular Value Decomposition (RSVD).

---

**Algorithm 1:** Low Rank matrix multiplication.

---

**Input:** LowRank  $A(A_U, A_X, A_{V^T})$  of  $(m, k, rank_A)$ , LowRank  $B(B_U, B_X, B_V^T)$  of  $(k, n, rank_B)$

**Result:**  $G_{XY}$  of size  $rank \times rank$

- 1  $C_{temp} = A_{V^T} \cdot B_U$
  - 2  $E_{temp} = A_X \cdot C_{temp}$
  - 3  $G_{XY} = E_{temp} \cdot B_X$
- 

from the Singular Value Decomposition of the matrix) reduce very rapidly, we only need to retain the first few singular values and associated basis vectors, thus expressing the dense matrix with significant data compression. The number of significant bases retained is the *numerical rank* of the matrix.

Low rank matrices are generated from the corresponding dense matrix block using a decomposition like randomized SVD (Singular Value Decomposition) [28], Interpolative Decomposition [28], and Adaptive Cross Approximation (ACA) [49] (which are more efficient than SVD). The dimensions of the low rank approximation of a dense matrix of dimension  $m \times n$  using a rank of  $rank$  can be represented using a tuple of 3 elements  $(m, n, rank)$ . A dense matrix  $A_{m \times n}$  is represented as a product of three matrices as shown in Eq. 1, and represented in Fig.1.

$$A_{m \times n} \approx U_{m \times rank} \cdot S_{rank \times rank} \cdot V_{rank \times n} \quad (1)$$

Thus the total storage requirement of the matrix reduces to  $m \times rank + rank \times rank + rank \times n$ , a value significantly smaller than  $m \times n$  memory necessary for the dense matrix, if  $rank$  is much smaller than  $m$  and  $n$ .

An important component in the approximation and solution of hierarchical matrices as shown in Section 1 is the low rank multiplication algorithm shown in Algorithm 1. This algorithm involves multiplication between two ‘skinny’ matrices  $A_{V^T}$  and  $B_U$  and two ‘small’ matrices  $A_X$  and  $B_X$  of the form  $A_X \times A_{V^T} \times B_U \times B_X$ . Multiplication between such matrices is particularly challenging as a result of their small sizes, which makes the computation heavily memory bound. The technique that we develop in this paper optimizes this specific step by batching a large number of independent low rank matrices together for improved performance. The low rank multiplication forms the first step of the rounded addition [7] algorithm for addition of low rank matrices, and the low rank matrix-vector multiplication.

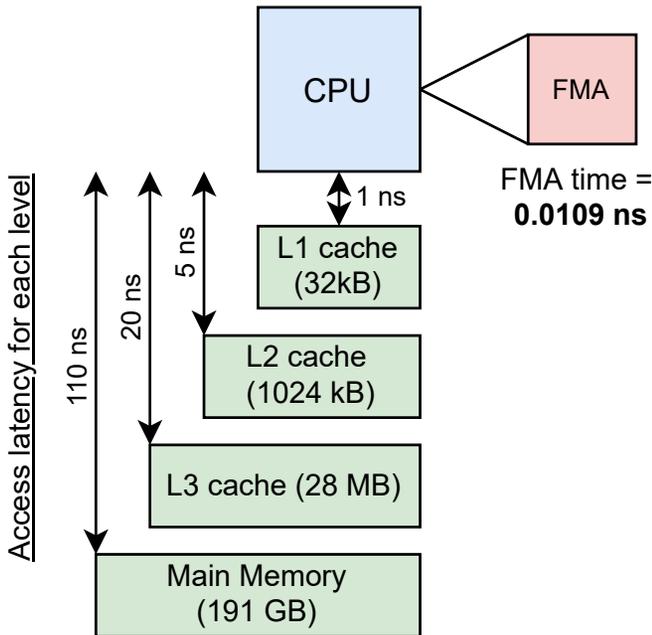


Fig. 2. Access latency along the memory hierarchy when fetching data to execute an FMA on an Intel Xeon 6148 CPU (Skylake-X micro-architecture).

### 3 PERFORMANCE OPTIMIZATION ON MULTI-PROCESSORS

#### 3.1 Software methodologies for optimal hardware utilization

Having reached the limits of Moore's law, CPU architecture designers have moved from simply improving CPU clock speed and die size to exposing multiple layers of parallelism in their designs. Innovations such as SIMD architectures, Simultaneous Multi-threading (SMT) and ccNUMA designs have led to potential parallelism on every level of the CPU. However, there is still a large difference in the time taken for arithmetic operations vs. time taken for fetching them from memory. For example, Fig. 2 shows the time to execute an FMA operation on a single core on the Intel Xeon Gold 6148 CPU vs. the time taken to fetch a single double-precision number from various layers of the cache hierarchy. Thus, the large ratio of memory vs. CPU speed (termed the 'machine balance' [43]), has led software writers to adopt innovative data locality optimizations in their designs in order to ensure that hardware spends most of its time on performing useful arithmetic calculations.

The dense linear algebra community has envisioned several approaches for optimizing dense linear algebra routines using analytically modeled blocked algorithms [6, 23, 40, 54, 61, 63], auto-tuning [56], systematic derivation of algorithms [24] and recursive cache oblivious algorithms [18, 19]. Memory bound sparse matrix algorithms have similarly seen various innovations [4, 35, 45, 55] where the implementation of register blocking and new sparse matrix formats have led to increased efficiency. The tiny matrices arising out of low rank multiplication can be potentially batched together for better SIMD and bandwidth utilization, as has been done by batched matrix multiplication routines from MAGMA [27], Intel Math Kernel Library, and KokkosKernels [38].

Various approaches have been proposed for batching on both CPUs and GPUs [1, 2, 14, 37, 41]. The LIBXSMM library [22, 29] even optimizes batched matrix operations using register blocking and a JIT compiler. Alternate data layouts that interleave data across batches have been shown to

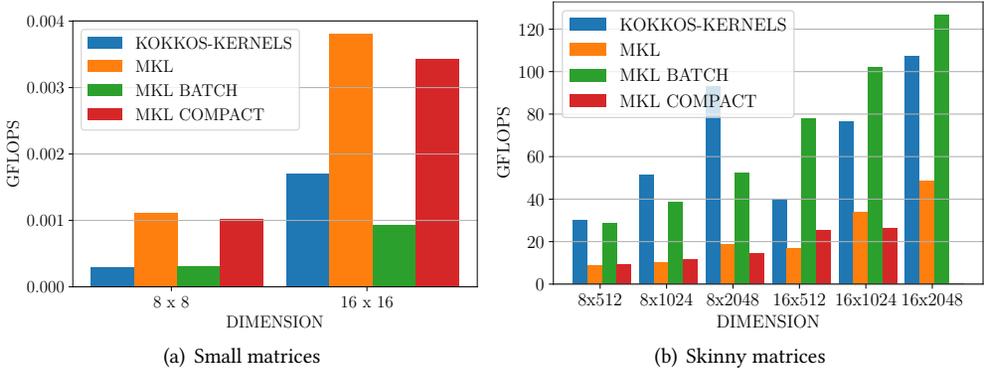


Fig. 3. Performance of batched small and skinny matrices for a batch size of 10000 using 20 physical cores using various batching techniques on a Intel Xeon Gold 6148 CPU. MKL uses non-batched MKL routines in a loop around the batch dimension. MKL BATCH performs batching with batched MKL routines without using memory interleaving across the batch dimension, and MKL COMPACT performs batching using memory interleaving as described by Dongarra et. al. [16]. Kokkos-kernels [38] is a library designed from the ground up for obtaining efficiency with heterogeneous architectures and irregular matrix sizes.

outperform simple batching in some cases [16, 38]. This approach differs from the other batching approaches in that it utilizes a different permutation of the data across the batch dimension to keep the SIMD units busy. While efficient for very small matrices, the packing and unpacking quickly becomes a bottleneck as the matrix sizes increase. LibShalom [60] does away with the packing overhead for specific matrix sizes and optimizes the instruction schedule to achieve high performance of small matrix multiplications on ARM v8 CPUs. BLASFEO [20, 21] also provides batched LAPACK routines apart from matrix multiplication and achieves better performance than vendor optimized libraries on various CPUs. TSM2 and TSM2X [15, 48] are specifically built for optimizing tall-and-skinny matrix multiplication and use code generation for tuning the usage of threads and the cache hierarchy on the GPU.

Fig. 3 shows the performance of various batching techniques (KOKKOS-KERNELS, MKL BATCH and MKL COMPACT) vs. not batched techniques (MKL) for independent small matrices in Fig. 3(a) and for independent skinny matrices in Fig. 3(b) for a constant batch size of 10,000. Performing the same computation using batched routines shows better performance in most cases. The batch size is fixed at 10,000 since it is found to be a sufficiently large batch size to show the benefits of batched vs. non-batched routines. Clearly, while efficient implementations for batched skinny matrices exist, small matrix operations are always under-performing. These operations are the least efficient part of many algorithms, and as such are the most promising candidates for enhancing the performance of low rank multiplication. Although Kokkos-kernels comes close to the performance of vendor-optimized batched routines, it does so for only some specific cases and we therefore use only vendor-optimized libraries in our experimental evaluation in Sec. 7.

### 3.2 Optimization with performance modeling

Performance modeling is useful for predicting the ideal or peak performance of an algorithm given architectural constraints. It can be used not only for analytically deriving the peak performance, but also for indicating which area most benefits from direct optimization.

Techniques such as the roofline model [58] are useful for gaining a measure of the memory-boundedness of an algorithm. However, these ‘top-level’ techniques do not dive deeper into the

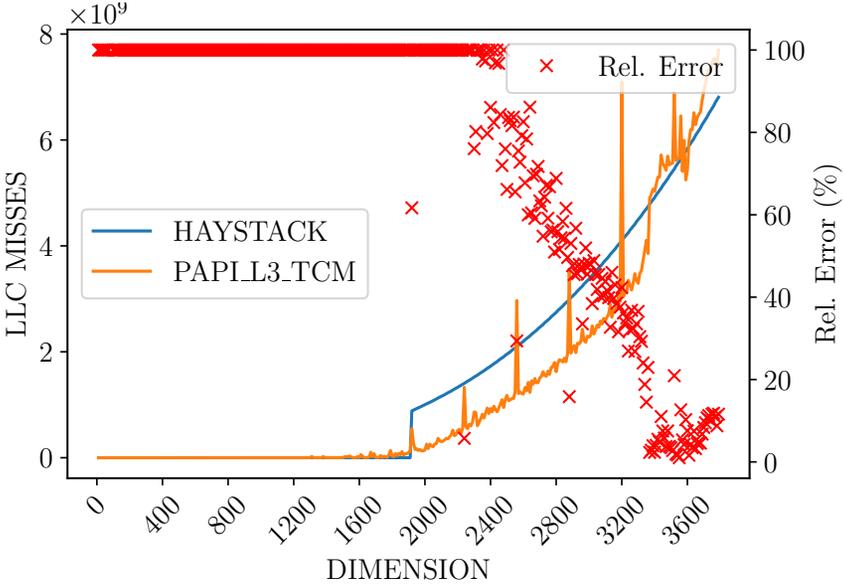


Fig. 4. LLC misses of small matrix multiplication as predicted by haystack. The Y axis on the right shows the relative error between the calculated LLC misses by haystack vs. PAPI\_L3\_TCM measurements for  $M = N = K$  as given in the X axis. Tests performed on a Intel Xeon Gold 6148 CPU (Skylake-X).

performance of the kernels in question. As a result, opportunities for optimization with the roofline model are difficult to identify.

### 3.3 Performance modeling of LLC misses

A simple approach for modeling the overall run time of memory bound applications is to measure the number of last-level-cache (LLC) misses [11, 25, 33, 34]. Several approaches ranging from analytical [34] and semi-empirical modeling [30] have been suggested for this purpose. More advanced approaches involving simulation have been suggested by [12, 42] using stack distances. This idea is extended by [25], whose tool ‘haystack’ allows predicting the LLC misses using a faster simulation methodology than previously available. However, as Fig. 4 shows, haystack is not able to predict the LLC misses of smaller matrices.

## 4 BATCHING METHODOLOGY

### 4.1 Looping order of Low rank multiplication

The usual low rank multiplication is shown in Algorithm 1. If done separately, this will lead to 3 nested loops for each multiplication. Since there exist dependencies between these loops, we end up writing the temporary blocks to memory after each step. With a rewrite of the loops, we can perform the batched matrix multiplication using 6 nested loops as shown in Algorithm 2. This way the result can be accumulated into a single variable  $G_{XY}$  and written back into memory only once, at the end of the computation. We also notice that all temporary blocks used in this new algorithm are tiny matrices (size  $rank \times rank$ ) and can fit within the vector registers when  $rank$  is sufficiently small. We term the  $A_X$  and  $B_X$  as ‘small matrices’ and  $A_{VT}$  and  $B_U$  as ‘skinny matrices’. Since the batch dimension is typically the largest dimension in a batched multiplication of small sized matrices, performing the multiplication as specified in Algorithm 2 allows for more optimal

---

**Algorithm 2:** Batched low rank multiplication expressed as 6 nested loops.

---

**Input:**  $A_{VT\_batch}$ ,  $B_{U\_batch}$ ,  $A_X\_batch$ ,  $B_X\_batch$

**Result:**  $G_{XY\_batch}$

```

1 for batch ← 0 to BATCH_SIZE do                               /* Loop 1 */
2    $A_{VT} = A_{VT\_batch}(batch)$ 
3    $B_U = B_{U\_batch}(batch)$ 
4    $A_X = A_X\_batch(batch)$ 
5    $B_X = B_X\_batch(batch)$ 
6    $G_{XY} = G_{XY\_batch}(batch)$ 
7   for m ← 0 to rank do                                       /* Loop 2 */
8     for n ← 0 to rank do                                       /* Loop 3 */
9        $C_{MN} = 0$ 
10      for k ← 0 to block_size do                               /* Loop 4 */
11         $C_{MN} += A_{VT}(m, k) \times B_U(k, n)$ 
12      end
13      for x ← 0 to rank do                                       /* Loop 5 */
14         $E_{XN} = A_X(x, m) \times C_{MN}$ 
15        for y ← 0 to rank do                                       /* Loop 6 */
16           $G_{XY}(x, y) += E_{XN} \times B_X(n, y)$ 
17        end
18      end
19    end
20  end
21 end

```

---

utilization of bandwidth and therefore reduces the amount of time spent in fetching data from main memory for a multi-threaded implementation. This has been experimentally proven to be true in Sec. 7. Therefore, a combination of improved bandwidth utilization and accumulation of intermediate results within SIMD registers allows our algorithm to achieve superior results than vendor optimized libraries.

#### 4.2 Locality optimization for low rank multiplication

BLISLAB [46] separates the implementation of the dense matrix multiplication into a portable macro kernel written in a high level language such as C, and an architecture-specific micro kernel typically written using intrinsics or assembly code. This allows libraries like BLIS to be portable across a diverse set of machines and exposes thread-level parallelism [51] in the macro kernel. This approach has been shown to attain near peak performance for a diverse set of CPU architectures as a result of enhanced data reuse.

We follow a similar approach for the low rank multiplication as shown in Algorithm 3. Assuming a three level cache hierarchy, loop 1A packs small matrices into the last level cache (L3) and loop 1B packs the skinny matrices into the L2 cache. Loop 1C then iterates over the batches of skinny matrices that are already packed in the cache. Thus loop 1 in Algorithm 2 is split into loop 1A, 1B and 1C in Algorithm 3. The data can then be streamed directly from the cache closest to the CPU (L1) when loading into registers. This can be changed to use only two cache levels in case of the A64FX CPU.

**Algorithm 3:** Batched low rank multiplication with a micro-kernel.**Input:**  $A_{VT\_batch}$ ,  $B_U\_batch$ ,  $A_X\_batch$ ,  $B_X\_batch$ **Result:**  $G_{XY\_batch}$ 

/\* Pack as many small matrices in as possible in L3 cache with thread-level parallelism. \*/

```

1 for batchsmall ← 0 to BATCH_SIZE step Bsmall do /* Loop 1A */
2   packed_AX ← pack_AX_matrices()
3   packed_BX ← pack_BX_matrices()
   /* Pack Bskinny skinny matrices into the L2 cache of each core. */
4   for batchskinny ← 0 to  $\frac{B_{small}}{B_{skinny}}$  do /* Loop 1B */
5     offset ← bsmall × Bsmall + bskinny × Bskinny
6     packed_BU ← pack_BU()
7     packed_AVT ← pack_AVT()
8     for batch ← offset to offset + Bskinny do /* Loop 1C */
9       GXY ← GXY_batch(batch)
10      for mc ← 0 to m step MPACK do /* Macro kernel. Loop 2 */
11        for nc ← 0 to n step NPACK do /* Loop 3 */
12          CMN = micro_kernel_cmn(
13            mc, nc, packed_BU,
14            packed_AVT
15          )
16          for xc ← 0 to x step XPACK do /* Loop 5 */
17            EXN = micro_kernel_exn(
18              mc, xc, packed_AX, CMN
19            )
20            for yc ← 0 to y step YPACK do /* Loop 6 */
21              GXY = micro_kernel_gxy(
22                packed_BX, EXN,
23                nc, yc
24              )
25            end
26          end
27        end
28      end
29    end
30  end
31 end

```

Algorithm 2 maintains portability across CPU architectures with varying cache sizes by changing the parameters  $B_{small}$  and  $B_{skinny}$  that control the number of small and skinny matrices being packed into cache, respectively. The number of small matrices  $B_{small}$  being packed into the LLC is determined using Eq. 2 (assuming **double** as the type of our matrices). Eq. 2 is obtained by dividing the number of bytes that the L3 cache can hold by the total number of bytes required for holding two  $rank \times rank$  small matrices.

CPU	$rank \geq VL$	$X_{PACK}$	$Y_{PACK}$	$M_{PACK}$	$N_{PACK}$
Fujitsu A64FX	YES/NO	8	8	8	8
Intel Xeon Gold 6148	YES	4	16	8	16
	NO	8	8	8	8
AMD EPYC 7502	YES/NO	4	4	4	4

Table 1. Values of slicing variables according to architecture and  $rank$ . It is experimentally found that AMD and Fujitsu micro kernels do not need modification irrespective of the  $rank$  whereas the Intel micro kernels perform best when the slice widths are changed if the rank is greater than the vector length ( $VL$ ).

The L2 cache typically has enough capacity to hold multiple skinny matrices from each operand of the low rank multiplication for a variety of block and rank sizes. Fig. 5 shows the effect of changing the number of skinny matrices from each low rank operand packed into the L2 cache. Experimentally, we find that packing only a single skinny matrix from each low rank operand leads to the best performance when using a sufficiently large batch size of 20,000 using an entire CPU socket (20 cores). Therefore, we fix  $B_{skinny} = 1$  for all future experiments.

Algorithm 3 shows how the loops shown in Algorithm 2 can be expressed in terms of *macro kernel* loops shown by the corresponding loops 2,3,5 and 6, and three micro kernels, each for accumulating the  $C_{MN}$ ,  $E_{XN}$  and  $G_{XY}$  block. Each of these blocks is of size  $S_{VEC} \times S_{VEC}$  where  $S_{VEC}$  is the vector length of the CPU. Loop 4 from Algorithm 2 is absorbed into *micro\_kernel\_cmn()* and optimized using assembly code. The macro kernel loops choose the slices of the packed blocks that must be computed by the micro kernels. The variables  $M_{PACK}$ ,  $N_{PACK}$ ,  $X_{PACK}$  and  $Y_{PACK}$  control the sizes of the slices that the macro kernel loops iterate over. These values are changed according to the architecture and the rank of the problem.

For the case where  $rank = S_{VEC}$ , an entire matrix  $G_{XY}$  of dimension  $S_{VEC} \times S_{VEC}$  can be computed within the registers without having to perform expensive reads and writes of the temporary matrices  $C_{MN}$  and  $E_{XN}$ . In this case all the blocking variables in Algorithm 3, i.e.  $M_{PACK}$ ,  $N_{PACK}$ ,  $X_{PACK}$  and  $Y_{PACK}$  will be equal to  $rank$ . In these instances the computation can be performed directly within the vector registers without a single write to memory. When the  $rank$  is too large to fit into the SIMD registers, we perform blocking by using register blocks of different values so that the multiplication time of the skinny matrices is minimized and yields the best performance. Table 1 shows the values of these variables when using a rank equal to the vector length and greater than the vector length for each CPU.

$$B_{small} = \left\lfloor \frac{LLC_{bytes}}{2 \times rank \times rank \times sizeof(double)} \right\rfloor \quad (2)$$

### 4.3 Packing techniques for minimization of latency

We tried out an alternate packing technique than that suggested in Sec. 4.2, where we packed the skinny matrices  $B_U$  and  $A_{VT}$  matrices in the L3 cache and small matrices in the L2 cache. However, this showed lesser performance than packing the small matrices in the L3 cache. This is because the time for fetching small matrices into the L3 cache is minimized when the outermost loop (loop 1A in Algorithm 3) performs this fetching since that loop has the most parallelism. Therefore, the small matrices can be packed into the cache with maximum bandwidth utilization.

Another technique is to pack the data across batches (similar to [38]). If this is done, the 6 loop structure used in Algorithm 2 cannot be used since it relies on processing each low rank matrix one after another and not when the matrices are interleaved. Moreover, the interleaved batched layout is slow for skinny matrices as shown in Fig. 3.

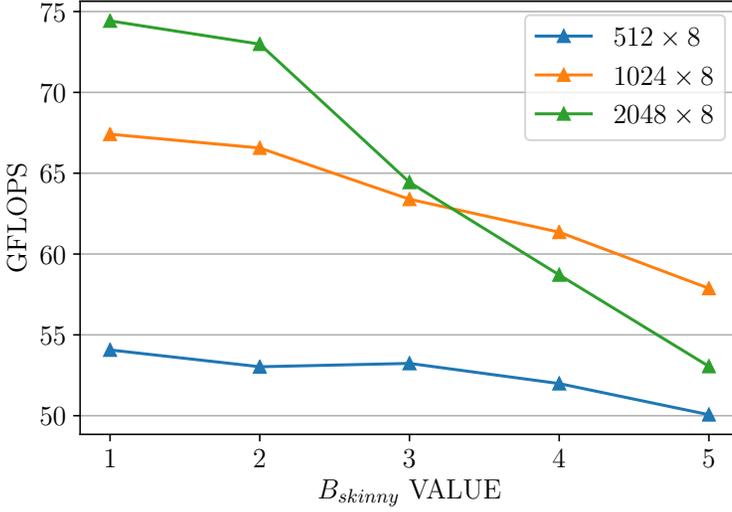


Fig. 5.  $B_{skinny}$  is the number of skinny matrices from each low rank matrix operand being packed into the L2 cache. This experiment shows the variation in performance as  $B_{skinny}$  is varied, keeping a sufficiently large batch size constant at 20000 and using 20 physical cores of an Intel Xeon Gold 6148 CPU. It can be seen that  $B_{skinny} = 1$  leads to the best performance, and this value is fixed in all future experiments.

## 5 THE ECM PERFORMANCE MODEL

The ECM (Execution-Cache-Memory) performance model [59] is an analytical technique for modeling the performance of steady-state loops. It models the ideal number of clock cycles necessary for execution of a single iteration of a loop subject to the constraints of the algorithm and machine. This level of detail allows individual modeling of the kernels of an algorithm and cycles through an optimization process until they match or are close to the predicted performance.

Alappat et. al. have previously used the ECM model for optimizing sparse matrix vector multiplication on the Fujitsu A64FX [5]. The ECM model has been used for optimizing conjugate gradient based iterative solvers [17, 31], modeling the proportion of bandwidth utilized by overlapping kernels [3], and performance tuning and optimization of CFD applications [57, 59]. Witmann et. al. [59] combine the ECM model with an energy consumption model to optimize both the performance and power consumption of a lattice-boltzmann CFD solver. They compare various performance modeling techniques and ultimately utilize insights gained from the ECM model for gaining the most significant speedups in their application.

The ideal number of clock cycles for executing a single iteration of a loop using a single thread on a single physical core is given by  $T_{ECM}$ . As shown in Eq. 3,  $T_{ECM}$  is composed of various terms, which can be described as follows:

- $T_c$  – Clock cycles for pure compute instructions such as FMA and addition.
- $T_{L1L}$  – Clock cycles for loads from L1 cache into registers.
- $T_{L1S}$  – Clock cycles for stores from registers to L1 cache.
- $T_l$  – Clock cycles for reads and writes between  $l - 1$  and level  $l$  cache.
- $T_{mem}$  – Clock cycles for reads and writes between main memory and the last level cache (LLC).

$$T_{ECM} = \max(T_c, f(T_{L1L}, T_{L1S}, T_2 \dots T_l, T_{mem})) \quad (3)$$

The main strength of the ECM model lies in the fact that it allows building an estimate of the overlap between levels of caches in a CPU. An important step in modeling the performance of any CPU using the ECM model involves first finding the function  $f$  in Eq. 3 in order to quantify whether the reads and writes between the caches are simultaneous or in serial. This step differs between various CPU designs and has a non-trivial impact on the predicted performance.

Since the ECM model takes into account the individual instructions that comprise a loop, various bottlenecks such as assembly code generation by the compiler, inconsistent cache usage, and lack of Out of Order execution can be quickly pointed out.

We use the methodology provided by Hofmann et. al. [31] for obtaining the ECM model for a given CPU architecture. We first build the ECM model for the STREAM TRIAD kernel for each CPU. This is done by first building a model of the machine as shown in Sec. 5.1. We then build an application model specifically for STREAM TRIAD as shown in Sec. 5.2. Finally, we can obtain the equation for  $T_{ECM}$  for each CPU as shown in Sec. 5.3. Since  $T_{ECM}$  for each CPU (Table 2) remains constant for all steady-state loops for that CPU, the same equation can be used for our specific application of low rank multiplication.

## 5.1 Building the machine model

	AMD EPYC 7502	Fujitsu A64FX	Intel Xeon Gold 6148
Vector Length (bits)	512	512	256
Instruction Set	AVX2	ARM SVE	AVX512
Microarchitecture	Zen2	ARM v8.2 SVE	Skylake-X
Cores	32	48	20
FMA (/core)	2	2	2
LOAD (/core)	2	2	2
STORE (/core)	1	1	1
Cache line size (bytes)	64	256	64
Cache write policy	write-allocate	write-allocate	write-back
Victim cache	Victim L3	-	Victim L3
L1 load (bytes/cycle)	32	64	64
L1 store (bytes/cycle)	32	64	64
L2 load (bytes/cycle)	32	64	64
L2 store (bytes/cycle)	32	32	64
L3 load (bytes/cycle)	16	-	14
L3 store (bytes/cycle)	16	-	14
L1 size	$32 \times 32$ KiB	$48 \times 64$ KiB	$20 \times 32$ KiB
L2 size	$32 \times 512$ KiB	$4 \times 8192$ KiB	$20 \times 1024$ KiB
L3 size	$8 \times 16$ MiB	-	$20 \times 1.375$ MiB
Clock freq. (Hz)	$2 \times 10^9$	$2 \times 10^9$	$2.2 \times 10^9$

Table 2. Various machine parameters used for building the ECM performance model for each CPU.

We first take into account various machine parameters as shown in Table 2 for our target architectures. The A64FX is configured to run on ‘normal’ mode, which means it runs at a frequency of  $2 \times 10^9$  Hz. For the Intel Xeon CPU we disable Turbo Boost and assume the frequency that is obtained by running a simple Fused Multiply Add loop using AVX-512 instructions. We chose these architectures since they use three diverse instruction sets, AVX2, ARM SVE and AVX-512. This allows us to compare the performance of the low rank matrix multiplication depending on

various capabilities provided by the Instruction Set Architecture (ISA), along with other machine parameters. The cache LOAD/STORE bandwidths from various levels of cache in Table 2 can be determined by formulating the STREAM usable bandwidth with empirical benchmarks and formulating and validating hypotheses about the bandwidth performance that fit the empirical measurements [31, Sec. 4].

## 5.2 Building the application model

The STREAM TRIAD [43] kernel is a simple kernel of the form  $A(i) = B(i) + \alpha \times C(i)$ . Assuming double precision, each execution of the kernel requires loading 16 bytes of memory ( $B(i)$  and  $C(i)$ ) and storing 8 bytes ( $A(i)$ ) for a total of 24 bytes data transfer per iteration, in addition to a single multiply and add operation that can be done as a single operation using the fused multiply-add instruction. Note that the actual amount of data transferred can vary according to the write policy of the caches. Every memory access is assumed to be a full cache line transfer [57].

We build an ECM application model for the STREAM TRIAD kernel on similar lines as has been done by [5, 31]. The instructions that make up the STREAM TRIAD kernel, along with their latency and throughput on various architectures can be seen in Table 3.

	AMD EPYC 7502		Fujitsu A64FX		Intel Xeon Gold 6148	
	Latency	Reci. TPut.	Latency	Reci. TPut.	Latency	Reci. TPut.
LOAD $A(i)$ , REG0	5	0.5	11	0.5	3	0.33
LOAD $B(i)$ , REG1	5	0.5	11	0.5	3	0.33
FMA REG0, alpha, REG1	5	0.5	9	0.5	4	0.75
STORE REG0, $C(i)$	4	0.75	9	1	3	0.66

Table 3. STREAM TRIAD kernel with respective latencies and throughputs.

Each instruction shown in Table 3 works in units of one Vector Length (VL) corresponding to the length shown in Table 2. The latency and throughput can be easily obtained by running identical instructions in succession with dependencies between successive instructions for latency and without dependencies for the throughput. Alternatively, the ibench [39] tool can be used.

## 5.3 Building the overlap hypothesis

As shown by Hofmann et. al. [31], building the overlap hypothesis is an important step in construction of the ECM model for a given CPU. Overlap hypotheses for the CPUs that we test have already been constructed for the Fujitsu A64FX [5], and also for AMD and Intel [31, 32] CPUs. In this section we use the techniques shown by the aforementioned authors to derive our own assumptions about performance using the ECM model. We build ECM models for each CPU as shown in Table 4. The performance assumptions from these models are validated in Fig. 6.

CPU	ECM Model
Fujitsu A64FX	$T_{ECM} = \max(T_c, \max(T_{L1L} + \max(T_{L1S}, T_{L2}), T_{mem}))$
Intel Xeon Gold 6148	$T_{ECM} = \max(T_c, T_{L1L} + T_{L1S} + T_{L2} + T_{L3} + T_{mem})$
AMD EPYC 7502	$T_{ECM} = \max(T_c, T_{L1L}, T_{L1S}, T_{L2}, T_{L3}, T_{mem})$

Table 4. Derived ECM model assumptions for each CPU in our tests.

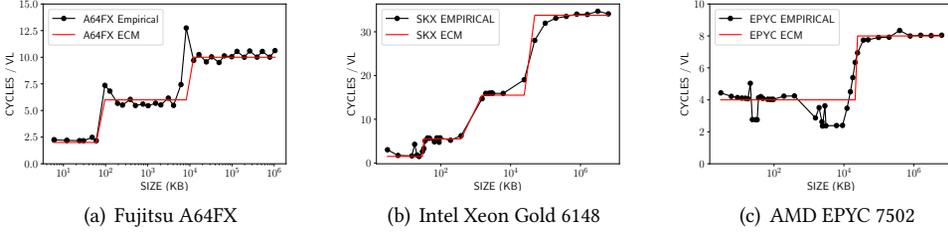


Fig. 6. Empirical vs. analytical clock cycles per vector length (VL) for each iteration of the STREAM TRIAD using a single core on each CPU. As the size of the data increases along the X axis, the number of cycles required for fetching one VL goes up. The ‘steps’ in each plot show how many cycles are needed when data fits into a particular level of cache. Each successive step shows that the data is being streamed from a progressively lower level of cache. The Fujitsu and Intel CPUs have a VL of 8 and the AMD CPU has a VL of 4.

Fig. 6(a) shows the cycles per VL for the STREAM TRIAD after disabling the compiler’s aggressive prefetching and pipelining for the Fujitsu A64FX. Each step corresponds to data being fetched from L1, L2 and main memory respectively. In practice, we observe that the compiler does a lot of aggressive prefetching into the L2 cache and exhibits behaviour as if the data were present in the L2 cache itself, which we use for constructing the ECM model. Fig. 6(c) shows that the AMD CPU goes from 4 cycles per VL to 8 after crossing the 16 MiB threshold as a result of the non-shared L3 cache present in the chiplet-based Zen2 architecture, in spite of having a total 128 MiB of L3 cache.

As per Table 11, even though the memory bandwidth of the AMD node is only 30% higher than the Intel node, comparing the clock cycles needed to fetch a single double precision digit from memory between Fig. 6(b) and Fig. 6(c) shows that the AMD CPU takes about half the clock cycles as the Intel CPU when the data size exceeds that of LLC as shown in Table 2. This can be attributed to the fact that the AMD CPU employs full memory overlap between all caches whereas the Intel CPU is completely non-overlapping, as can be seen in Table 4. Therefore, even though the Intel CPU employs the AVX-512 instruction set with 512-bit long SIMD, the advantage still lies with the AVX2-enabled 256-bit long AMD CPU as a result of fully overlapping communication between caches.

Thus, Fig. 6 shows that the Fujitsu A64FX takes the least number of clock cycles to fetch a single double precision number from main memory. This can be explained as a result of the better cache overlap design in the Fujitsu A64FX as shown in Table 4.

## 6 SINGLE THREADED OPTIMIZATION USING THE ECM PERFORMANCE MODEL

The computation of the low rank multiplication kernels can be broadly divided into kernel operations that involve packing the 4 matrices into caches, followed by the computation in the *micro\_kernel\_cmn()*, *micro\_kernel\_exn()* and *micro\_kernel\_gxy()* as shown in Algorithm 3. For brevity, we refer to *micro\_kernel\_cmn()*, *micro\_kernel\_exn()* and *micro\_kernel\_gxy()* from Algorithm 3 as the  $C_{MN}$ ,  $E_{XN}$  and  $G_{XY}$  kernels, respectively, after the intermediate products that they compute.

We use the ECM model for optimizing the performance of the  $C_{MN}$  kernel and the packing of  $A_{VT}$  and  $B_U$  since these are the most expensive parts. Using the ECM model we have identified and addressed several bottlenecks in the default C++ code, allowing us to reach very close to the maximum machine throughput. Fig. 7 shows the packing order of the operands into the caches. The small operands are packed into the shared L3 cache whereas the skinny matrices are packed

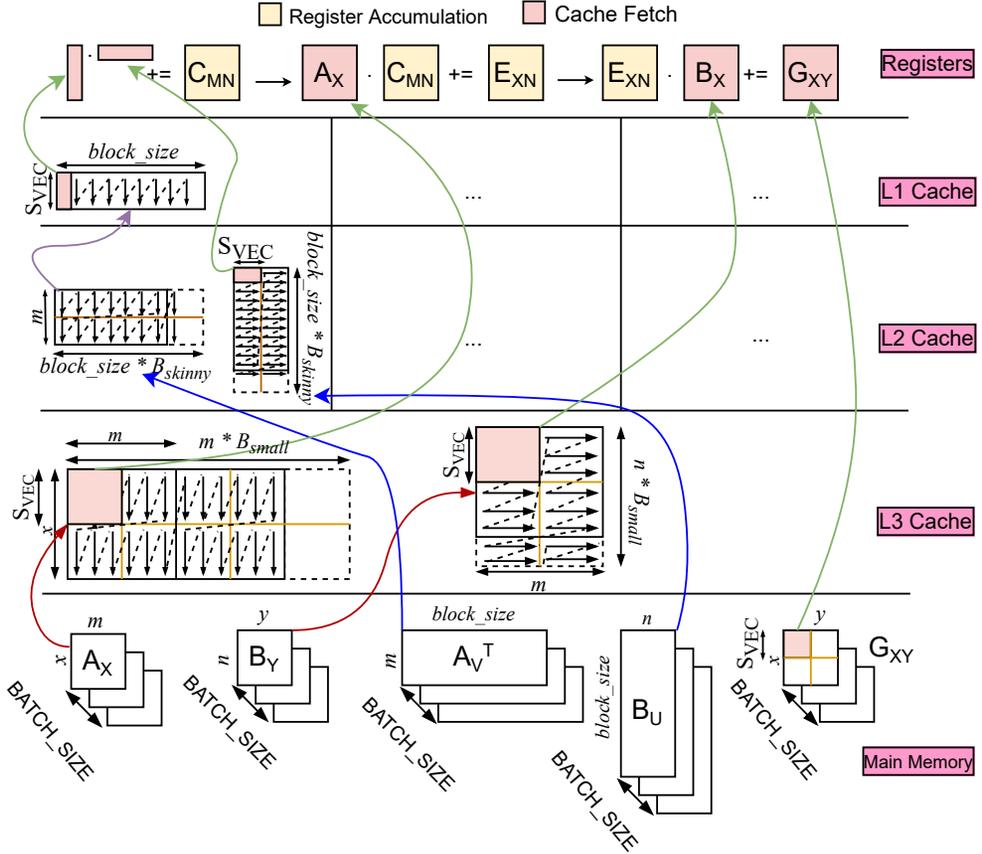


Fig. 7. Diagram of the proposed low rank multiplication batching method.

into the per-core L2 and L1 caches. The  $A_{VT}$  is packed in column major whereas the  $B_U$  is packed in row major for facilitating loading into SIMD registers [40].

We calculate the latency and throughput of each instruction used in the kernels, and report our findings in Table 5. Note that we only consider bench-marking the inner kernel execution and not the full computation, therefore we ignore all instructions outside of the specified kernel. However, the full computation is considered when reporting the final results in Sec. 7.

We demonstrate the use of ECM modeling for the packing kernels when using ranks that are multiples of the vector length of the machine. Indeed, as highlighted in [31, 3.4.2] the ECM prediction is a function of the total number of full cache lines transferred, so we need to count in full cache lines even if only a part of the cache line will be used. In case of strided cases with strides greater than  $rank$ , extra reads are considered as a result of reading more cache lines in order to factor in the increased stride.

When reporting ECM model predictions, we report them as  $T_{value} = (read) + (write)$  in order to differentiate between read and write contributions in case of caches where both values contribute to the outcome. In other cases the values show only read or only write contributions.

Instruction	Description	Latency	Reci. TPut.
<b>AMD EPYC 7502 (Zen2)</b>			
VMOVAPD(simple load)	Standard load with immediate addressing.	5	0.5
VMOVAPD(simple store)	Standard store with immediate addressing.	4	0.75
LEA	Load effective address.	2	0.5
VGATHERQPD(gather)	Gather with stride of 8/16/32/512/1024/2048/4096 .	-	6
VBROADCASTSD(simple)	Standard broadcast with a single double in memory.	4	0.75
VFMADD231PD(simple)	FMA between 3 256 bit AVX registers.	5	0.5
<b>Fujitsu A64FX (ARM v8.2 SVE)</b>			
LD1D(simple)	Standard load with immediate addressing.	9	0.5
LD1D(gather, stride 8)	Gather with stride of 8.	-	2
LD1D(gather, stride 16/32/512/1024/4096)	Gather with stride of 16/32/512/1024/4096.	-	4
LD1D(gather, stride 2048)	Gather with stride of 2048.	-	16
ST1D(simple)	Standard store with immediate addressing.	9	1
LD1RD(simple)	Standard broadcast with a single double in memory.	9	0.5
FMLA(simple)	FMA between 3 512 bit SVE registers.	11	0.5
<b>Intel Xeon Gold 6148 (Skylake-X)</b>			
VMOVAPD(simple load)	Standard load with immediate addressing.	3	0.33
VMOVAPD(simple store)	Standard store with immediate addressing.	4	0.66
VGATHERQPD(gather)	Gather with stride of 8/16/32/512/1024/2048/4096 .	-	3
VBROADCASTSD(simple)	Standard broadcast with a single double in memory.	1	0.33
VFMADD231PD(simple)	FMA between 3 512 bit AVX registers.	5	0.5

Table 5. Latency and throughput of instructions depending on their operators for each CPU tested. All operations except memory-specific operations are performed on double precision floating point numbers.

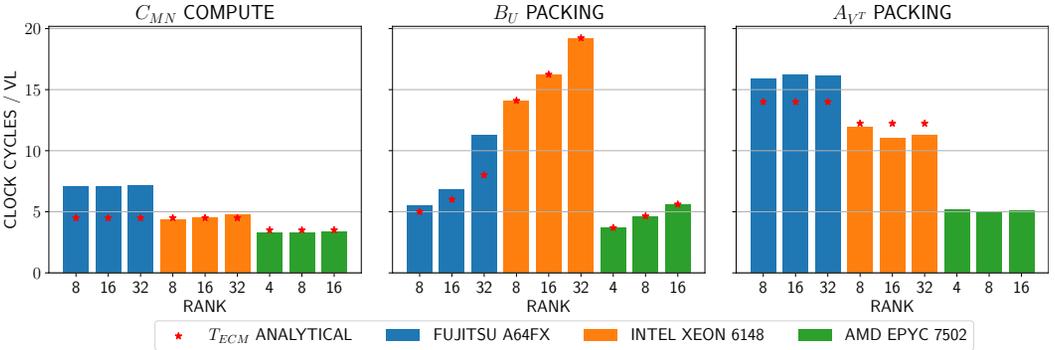


Fig. 8. Comparison of analytical and empirical number of clock cycles for packing  $A_{VT}$  and  $B_U$  and computing  $C_{MN}$ . All tests are performed for a block size of 1024 and batch size 10000 using a single thread of execution. The ranks are varied as shown in the X-axis. The Y-axis shows the number of clock cycles taken per vector length.

### 6.1 Overall comparison of analytical and empirical kernel performance

We first show analytical vs. empirical results for the computation  $C_{MN}$  and packing of  $B_U$  and  $A_{VT}$  for all CPUs. We address the challenges faced in reaching approximate peak ECM performance for each CPU in subsequent sections. We then elaborate on the strategy used for determining  $T_{ECM}$ .

Fig. 8 shows the empirical performance of the three kernels vs. the analytical peak performance determined using the ECM model for each. It can be seen that our code is able to reach theoretical peak performance as determined by the ECM model for almost all cases. In subsequent experiments, we term the number of matrices in a batch as the batch size and the longest dimension of the skinny

Variable	Stride 8	Stride 16	Stride 32
$T_{L1L}$	0.5	0.5	0.5
$T_{L1S}$	1	1	1
$T_{L2}$	$(\frac{64}{64}) + (\frac{64 \times 2}{64}) = 3$	$(\frac{64 \times 2}{64}) + (\frac{64 \times 2}{64}) = 4$	$(\frac{64 \times 4}{64}) + (\frac{64 \times 2}{64}) = 6$
$T_{mem}$	$(\frac{64}{64}) + (\frac{64}{32} + \frac{64}{32}) = 5$	$(\frac{64 \times 2}{64}) + (\frac{64}{32} + \frac{64}{32}) = 6$	$(\frac{64 \times 4}{64}) + (\frac{64}{32} + \frac{64}{32}) = 8$
$T_{ECM}$	5	6	8

Table 6. ECM performance breakdown for packing  $B_U$  for various strides for Fujitsu A64FX. All measurements are reported in number of clock cycles.

Block	Rank	$A_V$ analytical	$A_V$ empirical
2048	8	26	26.32
	16	26	26.22
	32	26	26.30

Table 7. Packing time per vector length for  $A_{VT}$  for Fujitsu A64FX for block size 2048. All measurements are shown in clock cycles per VL for a batch size of 10000.

matrices as the block size. We assume that both low rank operands have equal rank and equal block size.

We report data only for block size 1024 as we do not observe a significant deviation in the results for larger block sizes, except for  $A_{VT}$  packing using block size 2048 on the Fujitsu A64FX, which we elaborate on in Sec. 6.2.1.

## 6.2 Optimization on the Fujitsu A64FX with the ECM model

As shown in Table 5, the throughput of the LD1D instructions changes as the operands change. The reciprocal throughput of this instruction when performing a gather operation changes as the stride changes. We observe an anomaly when using a stride of 8 and 2048, where the reciprocal throughput changes to 2 and 16 respectively.

**6.2.1 Packing  $A_{VT}$  and  $B_U$  performance analysis.** We rely on manually inserting SVE gather and load intrinsics for reducing the packing time, since we found that the Fujitsu compiler does not make full use of vectorization. The  $B_U$  matrix is packed in row major order, corresponding to the format in which it is already stored, so we can utilize LD1D (load) instructions for this purpose. Table 6 shows the ECM calculations for a variety of possible strides for  $B_U$ .

The  $A_{VT}$  matrix is packed in column-major order, and similarly to the packing of  $B_U$  benefits from manually inserting SVE gather intrinsics. While Fig. 8 shows the empirical vs. analytical time taken when the block size is 1024. For most strides, the  $T_{ECM}$  can be calculated by setting  $T_{L1L} = 4$ ,  $T_{L1S} = 1$ ,  $T_{L2} = (\frac{64 \times 8}{64}) + (\frac{64}{64} + \frac{64}{64}) = 10$  and  $T_{mem} = (\frac{64 \times 8}{64}) + (\frac{64}{32} + \frac{64}{32}) = 12$ . We observe however a performance anomaly for a stride of 2048 corresponding to the increased latency of the LD1D instruction shown in Table 5. Table 7 shows the empirical vs. analytical performance when the block size is 2048.

**6.2.2  $C_{MN}$  kernel performance analysis.** The basic ARM SVE loop of the CMN kernel uses one LD1D instruction for loading 8 unique contiguous elements of the left operand, 8 FMA instructions for performing 8 SIMD multiplications and 8 LD1RD instructions for loading 8 elements of the right operand duplicated within each register for each rank-1 update that results in a matrix of size  $8 \times 8$ .

It can be seen that there are 8 LD1RD operations and 1 LD1D operation each taking 0.5 clock cycles, thus leading to a  $T_{LL}$  of 4.5. There are a total of 8 FMA operations, leading to a total  $T_c$  of 4 clock cycles. Thus  $T_{ECM} = \max(T_c, T_{LL}) = 4.5$  for a single rank-1 update.

We began by writing ARM ACLE intrinsics for the CMN kernel, but this turned out to be taking many more clock cycles than the ideal shown by the ECM model as a result of extra instructions generated in order to maintain portability between varying SVE lengths [53, 3.1]. We then enabled register length specific code generation and kept a fixed vector length of 512 using FCC compiler options, which led to more optimized code.

Table 5 shows that both the LD1RD and FMA instructions have a reciprocal throughput of 0.5 cycles each. Given that one rank-1 update generates an  $8 \times 8$  matrix, a single LD1RD and FMA will together generate one row of this matrix, taking 1 clock cycle. 8 of these pairs will use 8 clock cycles. The reason why these instructions are executed likewise is because we accumulate the intermediate products within the available SIMD registers, which places an upper limit on the size of the rank-1 update that can be performed in the CMN block. This, combined with the first LD1D will take about 8.5 clock cycles. As far as the LD1RD and FMA instructions are concerned, exactly 64 bytes are computed for 64 bytes loaded, which leads to a 1:1 ratio between the flops and bytes. For ideal FMA throughput, this ratio must be at least 2:1 so that two FMA instructions can execute independently on the two available FMA ports of the A64FX.

In order to overcome this limitation, we utilize 8 extra registers and perform two separate rank-1 updates using alternate slices of the skinny matrices to improve the performance further. We then add these slices at the end of the of the computation in order to obtain the block in registers z0-7. This leads to the time dropping to about 7 cycles per rank-1 update as a result of improved port pressure. The technique can be described as in Fig. 9.

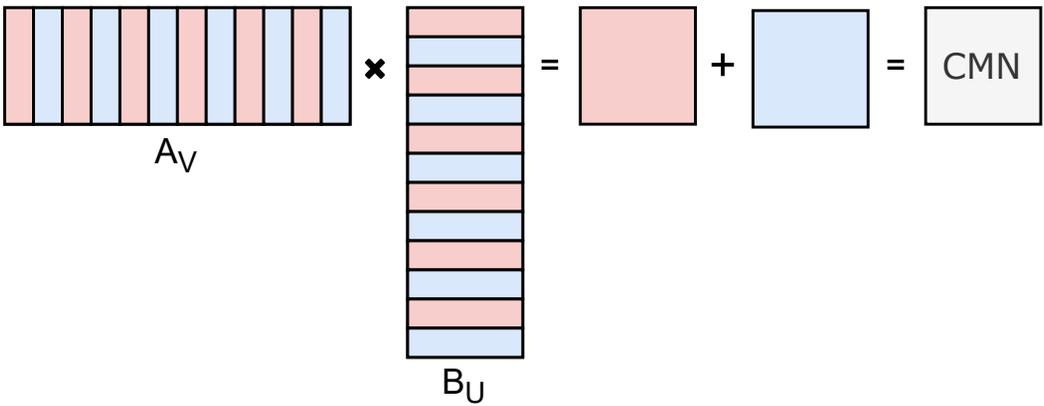


Fig. 9. Rank-1 update for  $8 \times 8$  matrix block using alternate rows and columns of the skinny matrices and then adding the blocks at the end.

The ECM model assumes that instructions run at peak throughput, however that can never be true in this case since the upper limit on the number of FMA instructions that can be executed given the constraints on blocking has been reached. Therefore, about 7 cycles per rank-1 update is the best that can be achieved given the constraints on the number of usable registers.

Table 8 shows the comparison of the analytical  $T_{ECM}$  compared to the performance obtained by successive optimizations. The usefulness of the ECM model can be seen in support of optimization.

Block	Rank	$T_{ECM}$	SVE ACLE	SVE intrinsics	SVE multi-register intrinsics
1024	8	4.5	30.11	8.59	7.12
	16	4.5	29.81	8.58	7.10
	32	4.5	29.77	8.78	7.15

Table 8. Comparison of successive optimizations to the CMN kernel compared to  $T_{ECM}$  for batch size of 10000 for the Fujitsu A64FX. All measurements are in terms of clock cycles per rank-1 update.

Variable	Stride 8	Stride 16	Stride 32
$T_{L1L}$	0.33	0.33	0.33
$T_{L1S}$	0.66	0.66	0.66
$T_{L2}$	$(\frac{64}{64}) + (\frac{64}{64}) = 2$	$(2 \times \frac{64}{64}) + (\frac{64}{64}) = 3$	$(4 \times \frac{64}{64}) + (\frac{64}{64}) = 5$
$T_{L3}$	$(\frac{64}{64}) + (\frac{64}{64}) = 2$	$(2 \times \frac{64}{64}) + (\frac{64}{64}) = 3$	$(4 \times \frac{64}{64}) + (\frac{64}{64}) = 5$
$T_{mem}$	$(\frac{64}{14}) + (\frac{64}{14}) = 9$	$(\frac{64}{14}) + (\frac{64}{14}) = 9$	$(\frac{64}{14}) + (\frac{64}{14}) = 9$
$T_{ECM}$	14	16	20

Table 9. ECM performance breakdown for packing  $B_U$  for various strides for Intel Xeon Gold 6148 (Skylake-X).

### 6.3 Optimization on Intel Xeon Gold 6148 with the ECM model

**6.3.1 Packing  $A_{VT}$  and  $B_U$  performance analysis.** The ECM models for packing  $B_U$  can be constructed by first noting the fact that packing one VL (i.e. 8 doubles) requires one load (VMOVAPD(load)) and one store (VMOVAPD(store)) operation. These operations have a throughput of 0.33 and 0.66, respectively. The breakdown of the analytical ECM modeling is shown in Table 9.

When packing  $A_{VT}$ , we make use of gather instructions, which means there is one VGATHERQPD and one store (VMOVAPD(store)) being used per VL. It is hard to exactly model the behaviour of the gather instruction since it seems to be fetching several cache lines together without incurring the overhead for fetching each cache line individually. Our explanation is that as a result of page-based fetching for the L3 cache, we can ignore the fact that multiple cache lines are being fetched and model that as a single cache line fetch. With this assumption, we can state that  $T_{L1L} = 3$ ,  $T_{L2S} = 0.66$ ,  $T_{L2} = \frac{64}{64}$ ,  $T_{L3} = 2 \times \frac{64}{64} = 2$  and  $T_{mem} = \frac{64}{14} = 4.57$ .

**6.3.2  $C_{MN}$  kernel performance analysis.** Each rank-1 update on the  $C_{MN}$  kernel requires 1 VMOVAPD, 8 VBROADCASTSD, and 8 VFMAADD231PD instructions. Given that the data is always streamed from the L1 cache, the only cost is  $T_{L1L} = T_{ECM} = 4.66$ .

### 6.4 Optimization on AMD EPYC 7502 with the ECM model

In order to keep the comparison between A64FX, AVX-512 and AVX2 fair, we use data sizes that correspond to the vector length and its multiples for building the ECM models. Therefore, in case of AVX2, the ranks used are 4, 8 and 16 which correspond to the VL, twice the VL and four times the VL for each instruction. These factors of the VL are proportional to the factors taken for A64FX and AVX512, for which the ranks are 8, 16 and 32 as shown in Sec. 6.3 and Sec. 6.2 respectively. As shown in Sec. 7.4, the target application of the low rank multiplication is the matrix vector multiplication routine for a block low rank matrix. The accuracy of the multiplication can be changed using the admissibility condition, and we do not need to change the rank of the low rank matrices in order to modify the accuracy. Therefore, we test only for multiples of the SIMD register length for each CPU.

Variable	Stride 4	Stride 8	Stride 16
$T_{L1L}$	0.5	0.5	0.5
$T_{L1S}$	0.75	0.75	0.75
$T_{L2}$	$\frac{32}{32} + 2 \times \frac{32}{32} = 3$	$2 \times \frac{32}{32} + 2 \times \frac{32}{32} = 4$	$4 \times \frac{32}{32} + 2 \times \frac{32}{32} = 6$
$T_{L3}$	$\frac{32}{32} + 2 \times \frac{32}{32} = 3$	$2 \times \frac{32}{32} + 2 \times \frac{32}{32} = 4$	$4 \times \frac{32}{32} + 2 \times \frac{32}{32} = 6$
$T_{mem}$	$\frac{32}{16} + \frac{32}{16} = 4$	$\frac{32}{16} + \frac{32}{16} = 4$	$\frac{32}{16} + \frac{32}{16} = 4$
$T_{ECM}$	4	4	6

Table 10. ECM performance breakdown for packing  $B_U$  for various strides for a single thread on the AMD EPYC 7502. All measurements are reported in number of clock cycles.

**6.4.1 Packing  $A_{VT}$  and  $B_U$  performance analysis.** We can derive the ECM model by using the throughput values from Table 5 and the machine model from Table 2.  $B_U$  packing takes one LOAD and one STORE operation. Since the clock cycles depend on the stride, the theoretical performance for various values of stride are shown in Table 10.

**6.4.2  $C_{MN}$  kernel performance analysis.** The  $C_{MN}$  kernel in this case is very similar to that in Intel. The difference being that the VL is limited to 4 due to the AVX2 instruction set. Therefore, the rank-1 update in this case is for a 4x4 matrix block, unlike the 8x8 matrix block for the Intel. For each rank-1 update, we use 4 VBROADCASTSD, 4 FMADD231PD and 1 VMOVAPD instructions. This amounts to  $T_{L1L} = 3.5$ . Since the data is directly streamed from the L1 cache, all other terms in the ECM equation are 0 and therefore  $T_{ECM} = 3.5$ .

## 7 EXPERIMENTAL EVALUATION

We perform experiments using the nodes and compiler flags listed in Table 11. The hardware specification of the CPU within each node can be found in Table 2. Our method works for any kind of data. However, for these experiments we use randomly generated entries following a normal distribution in order to accurately evaluate all of our test matrices. Since we are only working with low rank multiplication, the data within the low rank blocks does not affect our results. All tests are performed using double precision floating point numbers.

$$GFLOPS = \frac{batch\_size \times (4 \times rank^3 + 2 \times rank^2 \times block\_size)}{time(s)} \times 10^{-9} \quad (4)$$

The GFLOPS, where shown, are calculated as presented in Eq. 4. Bandwidth calculation depends on the cache overlapping displayed by the particular CPU and will be specified where necessary. We compare the bandwidth of each problem size with the STREAM TRIAD bandwidth for a given number of physical cores in order to demonstrate the ‘ideal’ usable bandwidth vs. what is actually realized.

Experiments are performed using the full node available. For all cases except the A64FX we use the OpenMP configuration as `OMP_PLACES=cores` and `OMP_PROC_BIND=close`. We run all tests using `numactl` using the `-membind=all` configuration and set `-physcpubind` to bind distinct physical cores. The Fujitsu runtime in FUGAKU imposes a slightly different binding process than other machines. Spawning a single process per CMG (i.e. one process per 12 cores) is the recommended configuration for using all NUMA nodes with strictly local access. Thus, we run 4 MPI processes per node for the A64FX tests.

	AMD	Fujitsu	Intel
CPU	2 x AMD EPYC 7502	1 x A64FX	2 x Intel Skylake-X 6148
Memory	2 x 256 GiB	1 x 32 GiB (HBM)	2 x 192 GiB
NUMA configuration	4/CPU	4/CPU	1 NUMA node/CPU
Compiler	g++ 7.5.0	FCC 4.5.0 tcsds-1.2.31	g++ 7.4.0
Compile options	-Wall -fopenmp -O3 -Ofast -mavx2 -funroll-loops -masm=intel	-O3 -Nfjompilib -fopenmp -Kfast,zfill -Kopenmp -Ksimd_reg_size=512	-Wall -fopenmp -O3 -Ofast -march=skylake-avx512 -masm=intel
Math library	AMD BLIS 3.0.0	Fujitsu SSL-2	Intel MKL 2020.4
Math library linking options	libblis-mt.a -lgomp -lpthread -lm -ldl	-Kopenmp -Nfjompilib -lfjlapacksve	-lmkl_intel_ilp64 -lmkl_gnu_thread -lmkl_core -lgomp -lpthread -lm -ldl
TRIAD peak (Gb/s)	195	840	150
DGEMM peak (GFLOPS)	2184	2828	2621

Table 11. Machine architecture and the corresponding configuration used in our experiments. The Intel node is a single node of the ABCI supercomputer and the Fujitsu node is a single node of the FUGAKU supercomputer. The AMD node is a stand-alone SMP machine.

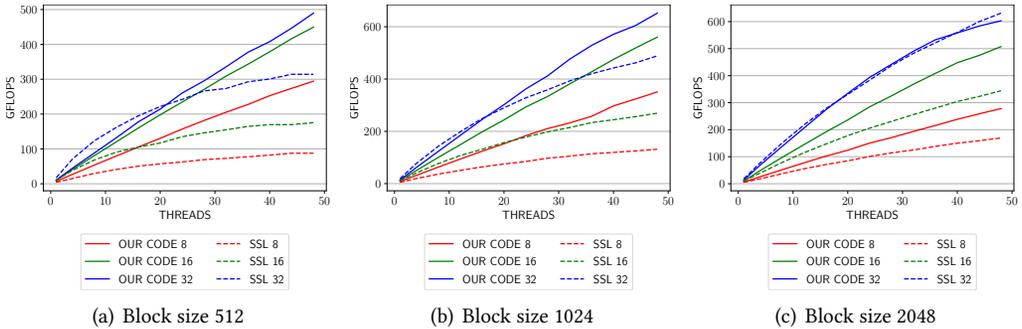


Fig. 10. Comparison of performance in GFLOPS vs. non-batched SSL-2 routines for Fujitsu A64FX. The batch size is kept constant at 20,000 for all the tests. The legend indicates the rank for each plot.

## 7.1 Evaluation on the Fujitsu node

The utilization on the Fujitsu A64FX is shown in Fig. 10. It can be seen that our code outperforms Fujitsu’s SSL-2 library by a wide margin except for one case using rank 32 and block size 2048 when not using all the cores in the CPU. The bandwidth utilization plots in Fig. 11 show that, although the GFLOPS utilization for rank 32 is consistently higher than other ranks, the bandwidth utilization is lower. This result indicates that the rank 32 case is in fact compute bound and not memory bound when using smaller ranks.

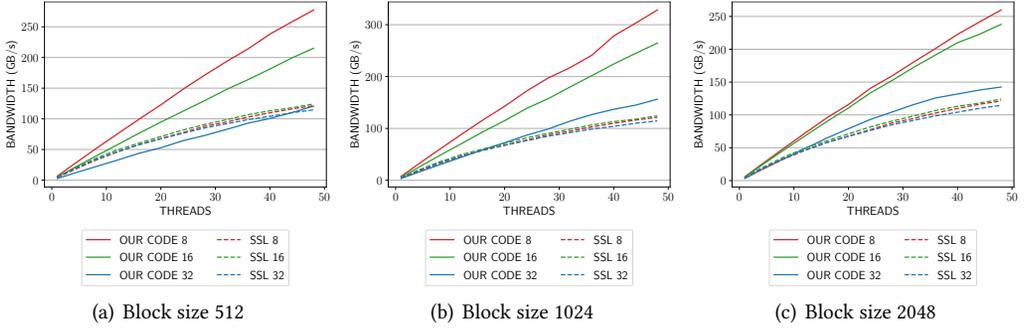


Fig. 11. Bandwidth utilization for varying ranks and block sizes on Fujitsu A64FX. The legend shows the rank for each plot. The bandwidth is calculated using Eq. 5. The peak STREAM TRIAD bandwidth for the A64FX is about 840 GB/s.

$$BW(\text{GiB/s}) = \frac{\text{batch\_size} \times (2 \times \text{rank}^2 + 2 \times \text{rank} \times \text{block\_size}) \times \text{sizeof}(\text{double}) \times 2^{-30}}{\text{time}(s)} \quad (5)$$

Lack of linear strong scaling can be seen when using rank 8 and block size 512 in Fig. 10, which gradually improves as the block size is increased. This effect is not observed for other ranks that show almost uniform linear scaling. The reason for this can be seen from the bandwidth utilization plot in Fig. 11, where the usage of the bandwidth is proportional to the GFLOPS utilization. The bandwidth is calculated as shown in Eq. 5. The bandwidth utilization is lesser than for rank 16 for the same block sizes since packing smaller skinny matrices individually into the L1 cache does not lead to optimal bandwidth utilization. Increasing the number of skinny matrices that are packed into the L1 cache during a single iteration of Loop 1 in Algorithm 2 might be a way to solve this problem.

When using rank 32 and block size 2048 in Fig. 11(c), it can be seen that the bandwidth utilization of SSL-2 and our code is almost the same unless all 4 NUMA nodes within the CPU are active, which happens after 36 threads are active. Fig. 12 shows the variation of the performance in GFLOPS as the number of threads is kept constant at 48 and batch size is varied. Our code consistently outperforms SSL-2, and there is minimal variation in the GFLOPS as the batch size is changed. Some results for the Fujitsu node could not be reported due to the limited 32 GiB HBM. However, it can be seen that each plot plateaus before we run out of memory, and therefore we can assume that performance will not degrade if there were more memory.

Fig. 13 shows the performance breakdown for each kernel of the computation on the A64FX when using 48 physical cores.

Table 12 shows that as the rank increases beyond 96, we can observe SSL showing better performance than our code. This can be attributed to the fact that the algorithm becomes more compute bound. The performance actually drops below that of rank 32, which can be attributed to the fact that better bandwidth utilization as shown in Fig. 11 becomes harder as a lesser number of small matrices can be packed into the L1 cache. This is one of the drawbacks of relying on packing intermediate products into the SIMD registers.

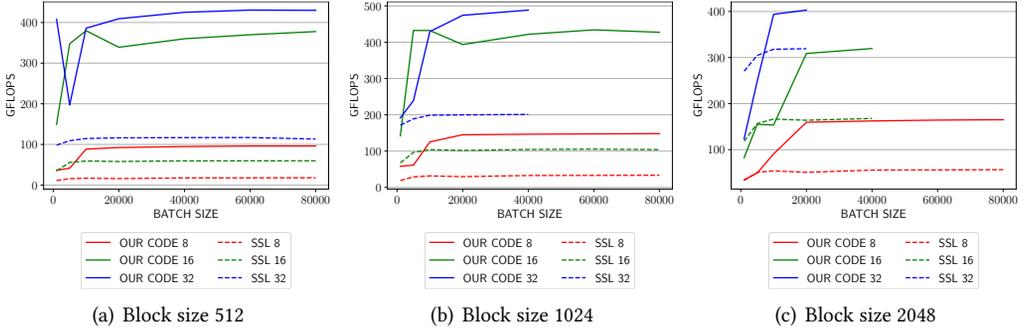


Fig. 12. Performance for various block sizes and ranks when the number of threads is constant at 48 for varying batch sizes for Fujitsu A64FX.

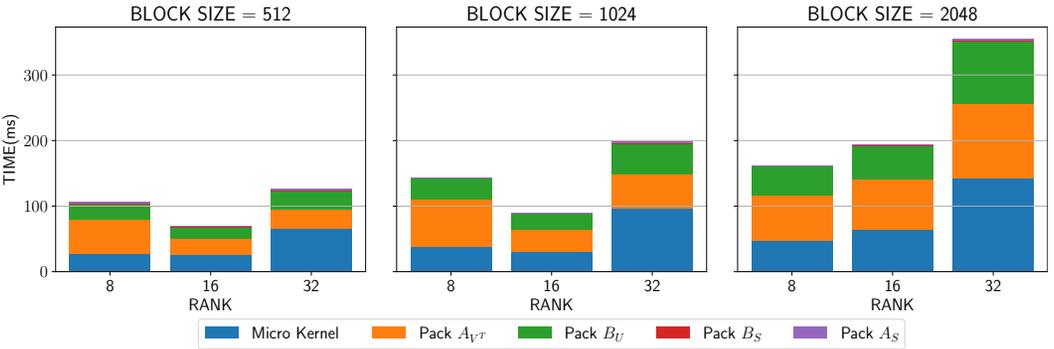


Fig. 13. Performance breakdown for various parts of the computation for the Fujitsu node using 48 threads and a constant batch size of 20000.

Batch size	Block size	Rank	SSL	OUR CODE
20000	512	96	370.62	207.55
20000	1024	96	598.93	343.912
20000	2048	96	851.74	521.93

Table 12. Performance of our code vs. SSL for larger rank on the Fujitsu node using 48 physical cores reported in GFLOPS. SSL shows better performance as the computation becomes more compute bound.

## 7.2 Evaluation on the Intel node

Fig. 14 shows the utilization in GFLOPS when the batch size is kept constant at 20,000 for a varying number of threads, block sizes, and ranks. It can be seen that our approach shows almost perfect scaling with respect to the number of threads whereas batched MKL routines stop scaling after approximately 10 physical cores have been utilized for all problem cases. We do not report findings for non-batched MKL routines since they show the least competitive performance for all problem cases.

The improved strong scaling can be attributed to the fact that our approach is able to saturate the maximum available bandwidth much better than batched MKL, as can be seen in Fig. 15, which

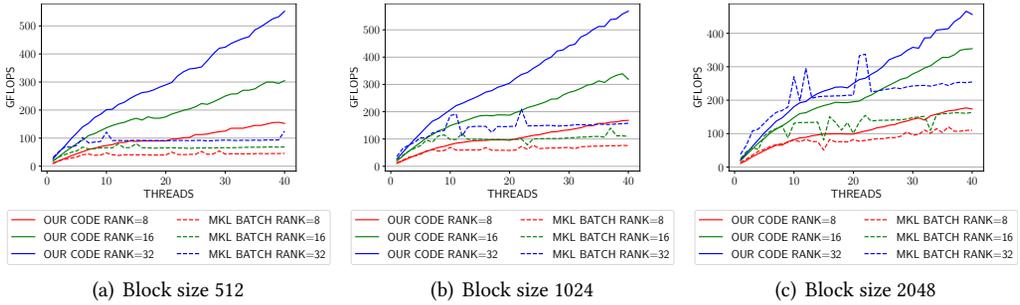


Fig. 14. Comparison of performance in GFLOPS vs. batched MKL routines for Intel Skylake-X. The batch size is kept constant at 20,000 for all the tests.

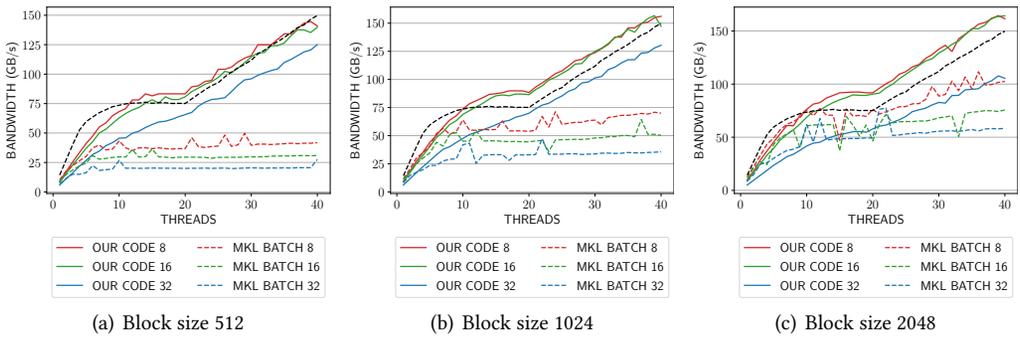


Fig. 15. Comparison of bandwidth utilization for varying ranks, threads and block sizes on Intel Xeon Gold 6148 (Skylake-X) for a constant batch size of 20,000. The black dashed line denotes the STREAM TRIAD bandwidth for the given number of threads.

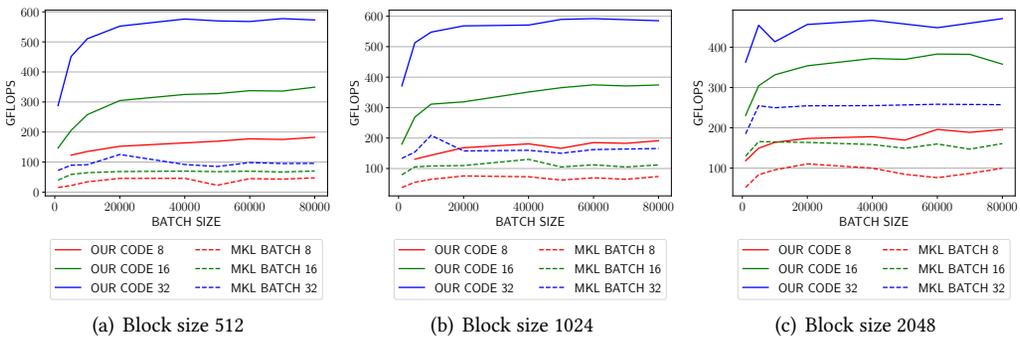


Fig. 16. Performance for various block sizes and ranks when the number of threads is constant at 40 for the Intel Xeon Gold 6148 (Skylake-X).

shows the bandwidth utilization in GiB/second with a constant batch size of 20,000 and varying number of threads, block sizes and ranks. This is as a result of our unique packing strategy.

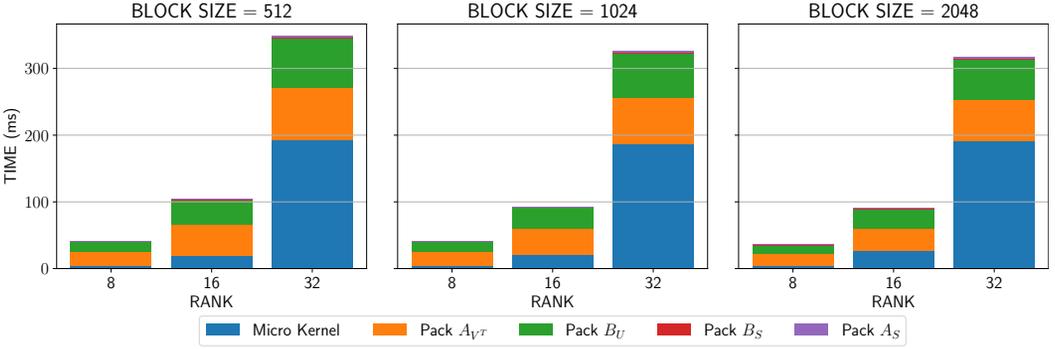


Fig. 17. Performance breakdown for various parts of the computation for the Intel node using 40 threads and a batch size of 20000.

Batch size	Block size	Rank	MKL BATCH	OUR CODE
20000	512	128	262.281	339.849
20000	1024	128	400.602	361.89
20000	2048	128	554.721	319.161

Table 13. Performance of our code vs. MKL batched for larger rank on the Intel node using 40 physical cores in GFLOPS. Batched MKL shows better performance as the computation becomes more compute bound.

The bandwidth numbers shown in Fig. 15 are calculated as shown in Eq. 6. We use the term  $3 \times rank^2$  in order to account for the write of the result matrix and reads of two small matrices. As the ECM model for Intel Skylake-X proves in Sec. 5.3, the reads and writes for this CPU are completely non-overlapping and the write term must be added into the bandwidth calculation.

$$BW(GiB/s) = \frac{batch\_size \times (3 \times rank^2 + 2 \times rank \times block\_size) \times sizeof(double) \times 2^{-30}}{time(s)} \quad (6)$$

Fig. 16 shows the performance when changing the batch size, block size and rank and keeping the number of threads constant at 40. It can be seen that both our implementation, and batched MKL show almost constant performance irrespective of the batch size. This, combined with Fig. 14 shows that the scaling of the method is primarily limited by the available bandwidth, rank and block size. Additionally, increasing the batch size does not have any effect on the performance.

Fig. 17 shows the time spent in the micro kernel (i.e. performing actual computation) vs. the time spent in packing data into the caches. This graph is a snapshot of the experiment with a batch size of 20,000 from Fig. 16.

As the rank increases beyond 128, we can observe MKL batched showing better performance than our code. Table 13 shows the difference in performance as the computation becomes more compute bound than memory bound.

### 7.3 Evaluation on the AMD node

As shown in Table 2, the vector length of the AMD CPU is 4, whereas that of the Intel and Fujitsu CPUs is 8. Therefore, we show benchmarks for rank 4 in case of the AMD chip as well as for rank 8, 16 and 32 as shown for the others. Fig. 18 shows the utilization as calculated using Eq. 4, which shows that our implementation is able to outperform the vendor optimized AMD BLIS 3.0.0

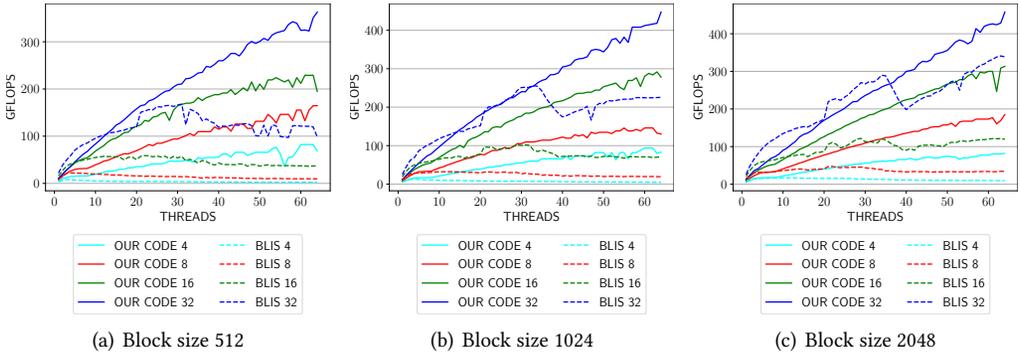


Fig. 18. Comparison of performance in GFLOPS vs. non-batched AMD BLIS routines for AMD EPYC 7502. The batch size is kept constant at 20,000 for all the tests. The legend indicates the rank for each plot.

BLAS implementation by a wide margin when a sufficient number of physical cores are active. The performance improves as we increase the rank, as expected, since memory bandwidth becomes less of a bottleneck as more data becomes available to keep the FMA units of the CPU busy.

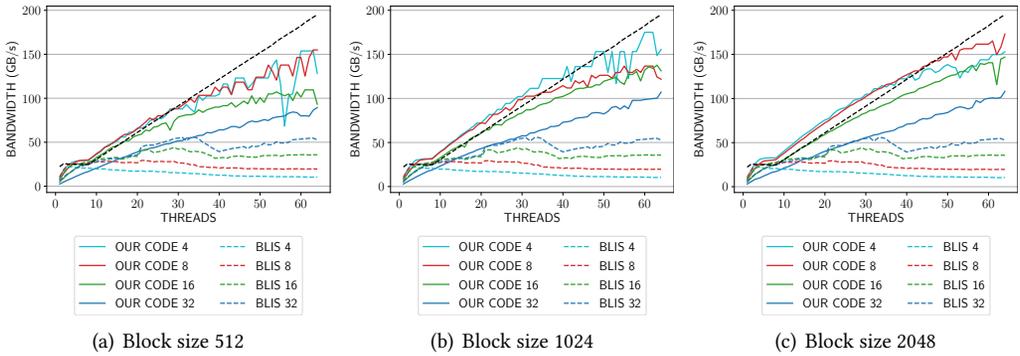


Fig. 19. Bandwidth utilization for varying ranks and block sizes on AMD EPYC 7502. The legend shows the rank for each plot. The black dashed line shows the STREAM TRIAD bandwidth for the given number of threads.

Fig. 19 shows the bandwidth utilization of our code vs. AMD BLIS for a variety of problem sizes. The bandwidth is calculated as shown in Eq. 5. For ranks 4 and 8, the bandwidth increases in proportion to the performance for all block sizes, as shown in Fig. 18. However, for rank 16 and 32, we observe that the total utilized bandwidth does not saturate the available bandwidth of the system, even though the corresponding GFLOPS utilization in 18 shows that the utilization for ranks 16 and 32 is much higher than that for 4 and 8. This phenomena can be explained by the fact that the AMD EPYC CPU implements fully overlapping caches as pointed out in Sec. 5.3. Therefore, the computation for rank 16 and 32 is more compute bound than memory bound when a sufficient number of physical cores are active. Compared to the bandwidth behaviour of the Intel node in Sec. 7.2, we can see that even though the AMD chip has similar bandwidth, it is able to use the available memory bandwidth much more efficiently as a result of the overlapping design of the caches.

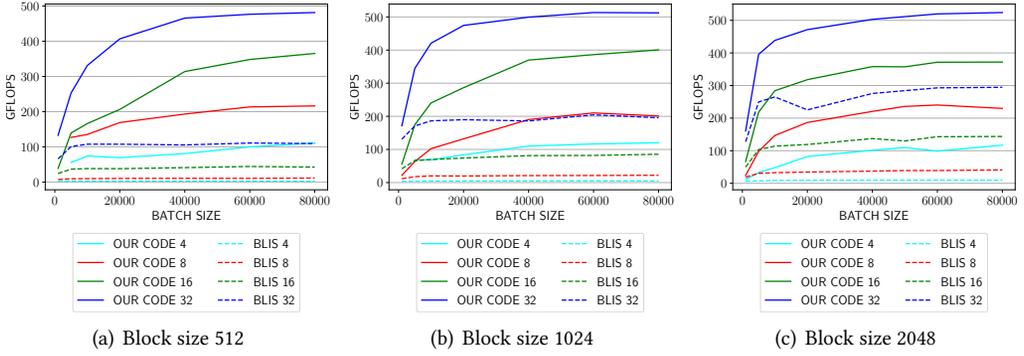


Fig. 20. Performance for various block sizes and ranks when the number of threads is constant at 64 for varying batch sizes for AMD EPYC 7502.

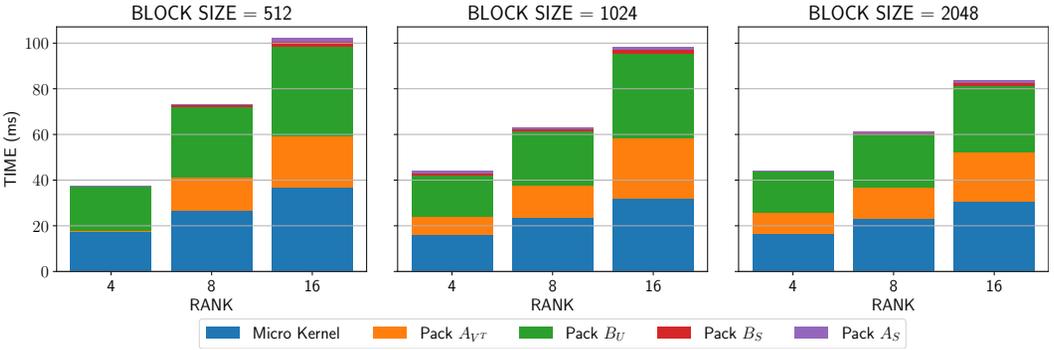


Fig. 21. Performance breakdown for various parts of the computation for the AMD node using 64 threads and a batch size of 20000.

Batch size	Block size	Rank	BLIS	OUR CODE
20000	512	96	197.056	237.441
20000	1024	96	373.555	283.175
20000	2048	96	559.075	348.566

Table 14. Performance of our code vs. BLIS for larger rank on the AMD node using 64 physical cores reported in GFLOPS. Our code can consistently beat AMD-BLIS until rank 96 is reached.

Fig. 20 shows the performance when keeping the number of threads constant at 64 (i.e. using the full node) and changing the batch sizes for variety of block sizes and ranks. While the AMD results differ from the Intel results in the fact that the batch size has a non-trivial effect on the utilization, our implementation still outperforms the AMD BLIS by a wide margin for every problem size. Fig. 21 shows the performance breakdown for each part of the execution for the AMD node.

Table 14 shows the performance of our code vs. BLIS as the problem gets more compute bound than memory bound. As shown for the Intel and Fujitsu nodes, the higher performance of AMD-BLIS can be attributed to the fact that larger sizes of small matrices leads to less optimal bandwidth utilization for our method.

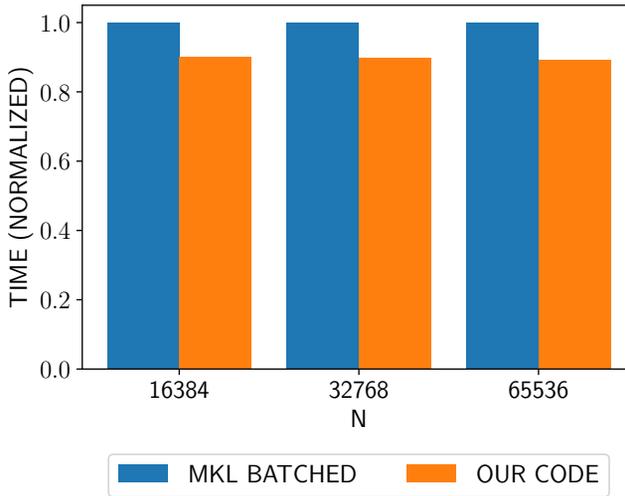


Fig. 22. Multiplying multiple right hand sides with a weakly admissible block low rank matrix of various sizes using batched MKL vs. our code for the batched multiplication routine. The rank and number of right hand sides are kept constant at 8 for all tests.

#### 7.4 Evaluation of Block Low Rank matrix vector multiplication

Fig. 22 shows the comparison of run time for multiplying multiple right hand sides using batched MKL vs. our implementation of batched low rank multiplication on the Intel node. A gain of about 15% can be observed. While the multiplication of just the low rank blocks and vectors shows a performance gain of about 50%, adding the intermediate vectors and multiplication of the dense blocks takes up more time, which reduces the performance gain to about 15%.

Since we show in the previous sections that our batching methodology is about twice as fast as vendor optimized libraries, we expect similar performance gains for other nodes too.

### 8 CONCLUSION AND FUTURE WORK

In this paper we have shown that performance of batched low rank multiplication can be improved with an alternative batching methodology based on improved data reuse and bandwidth utilization. Our results indicate better CPU utilization than vendor optimized libraries for a variety of thread counts and batch sizes on 3 major CPUs architectures, and for problem sizes critical to low rank operations. Specifically, we are able to achieve more than twice the performance of vendor optimized matrix multiplication routines when using the entire node for most problem cases. While most cases are memory bound, a larger rank of 32 actually results in a compute bound process. Thus our batching technique is able to keep the SIMD units busy enough that bandwidth is no longer the bottleneck, which is not true for vendor optimized libraries even for larger ranks. We run the same algorithm on all the CPUs, thus the cache misses generated by the data accesses would be proportional for the different libraries, thus the performance improvement comes from our optimization.

It is important to note that the constraints placed by the limited number of registers in SIMD architectures places limitations on the scalability of our method. However, hierarchical matrix factorization and vector multiplication typically involves low rank multiplication with block size up to 2048 and batch sizes not exceeding 20,000 – ranges where our approach is highly competitive with state of the art implementations, yielding significantly better results for this specific problem.

In the future we plan to use our technique for building fast factorization routines that run on distributed supercomputers. Distributed hierarchical factorization is challenging due to irregularity of communication patterns and computation. Cao et. al. [10] report better load balance by using an alternate process distribution and prioritization of the critical path using the PaRSEC [8] runtime system. A more analytical approach [36, 52] leads to better understanding of the trade-off between replication and communication of data for determining the data distribution on multiple nodes, which can possibly lead to more efficient process distribution and replication methodologies for minimization of communication overhead.

## ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Number JP18H03248, JP20K20624, JP21H03447. This work is conducted as a research activity of AIST - Tokyo Tech Real World Big-Data Computation Open Innovation Laboratory (RWBC-OIL). This work is supported by “Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures” in Japan (Project ID: jh210024-NAHI).

Many thanks to colleagues from Tokyo Institute of Technology, AIST, RIKEN-CCS and University of Tennessee at Knoxville who provided their valuable feedback for the improvement of this manuscript. Special thanks to Mohamed Wahib, Jens Domke and Chen Peng for their valuable feedback on performance models and optimization of Fujitsu A64FX benchmarks.

## REFERENCES

- [1] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, Tz. Kolev, I. Masliah, and S. Tomov. 2016. High-Performance Tensor Contractions for GPUs. *Procedia Computer Science* 80 (Jan. 2016), 108–118. <https://doi.org/10.1016/j.procs.2016.05.302>
- [2] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2016. On the Development of Variable Size Batched Computation for Heterogeneous Parallel Architectures. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, Chicago, IL, USA, 1249–1258. <https://doi.org/10.1109/IPDPSW.2016.190>
- [3] Ayesha Afzal, Georg Hager, and Gerhard Wellein. 2020. An Analytic Performance Model for Overlapping Execution of Memory-Bound Loop Kernels on Multicore CPUs. *arXiv:2011.00243 [cs]* (Oct. 2020). [arXiv:2011.00243 \[cs\]](https://arxiv.org/abs/2011.00243)
- [4] Christie Alappat, Nils Meyer, Jan Laukemann, Thomas Gruber, Georg Hager, Gerhard Wellein, and Tilo Wettig. 2021. ECM modeling and performance tuning of SpMV and Lattice QCD on A64FX. *arXiv:2103.03013 [hep-lat]* (March 2021). <http://arxiv.org/abs/2103.03013> [arXiv: 2103.03013](https://arxiv.org/abs/2103.03013).
- [5] Christie L. Alappat, Jan Laukemann, Thomas Gruber, Georg Hager, Gerhard Wellein, Nils Meyer, and Tilo Wettig. 2020. Performance Modeling of Streaming Kernels and Sparse Matrix-Vector Multiplication on A64FX. *arXiv:2009.13903 [cs]* (Sept. 2020). [arXiv:2009.13903 \[cs\]](https://arxiv.org/abs/2009.13903)
- [6] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. 1990. LAPACK: A Portable Linear Algebra Library for High-Performance Computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing (Supercomputing '90)*. IEEE Computer Society Press, New York, New York, USA, 2–11.
- [7] M. Bebendorf and W. Hackbusch. 2007. Stabilized Rounded Addition of Hierarchical Matrices. *Numerical Linear Algebra with Applications* 14, 5 (June 2007), 407–423. <https://doi.org/10.1002/nla.525>
- [8] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hault, and Jack J. Dongarra. 2013. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45. <https://doi.org/10.1109/MCSE.2013.98>
- [9] Michael Brazell, Na Li, Carmeliza Navasca, and Christino Tamon. 2011. Tensor and Matrix Inversions with Applications. *arXiv:1109.3830 [math]* (Sept. 2011). [arXiv:1109.3830 \[math\]](https://arxiv.org/abs/1109.3830)
- [10] Quinglei Cao, Yu Pei, Thomas Hault, Kadir Akbudak, Aleksandr Mikhalev, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. 2019. Performance Analysis of Tile Low-Rank Cholesky Factorization Using PaRSEC Instrumentation Tools. In *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. 25–32. <https://doi.org/10.1109/ProTools49597.2019.00009>
- [11] Marc Casas and Greg Bronevetsky. 2014. Active Measurement of Memory Resource Consumption. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, Phoenix, AZ, USA, 995–1004. <https://doi.org/10.1109/IPDPS.2014.105>

- [12] Calin Cascaval and David A. Padua. 2003. Estimating Cache Misses and Locality Using Stack Distances. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS '03)*. Association for Computing Machinery, San Francisco, CA, USA, 150–159. <https://doi.org/10.1145/782814.782836>
- [13] Ali Charara, David Keyes, and Hatem Ltaief. 2018. Batched Tile Low-Rank GEMM on GPUs. (2018), 12.
- [14] Ali Charara, David Keyes, and Hatem Ltaief. 2019. Batched Triangular Dense Linear Algebra Kernels for Very Small Matrix Sizes on GPUs. *ACM Trans. Math. Software* 45, 2 (June 2019), 1–28. <https://doi.org/10.1145/3267101>
- [15] 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. (2019), 11.
- [16] Jack J. Dongarra, Sven Hammarling, Nicholas J. Higham, Samuel D. Relton, and Mawussi Zounon. 2017. Optimized Batched Linear Algebra for Modern Architectures. In *Euro-Par*. [https://doi.org/10.1007/978-3-319-64203-1\\_37](https://doi.org/10.1007/978-3-319-64203-1_37)
- [17] Nils-Arne Dreier and Christian Engwer. 2019. Strategies for the Vectorized Block Conjugate Gradients Method. *arXiv:1912.11930 [cs, math]* (Dec. 2019). arXiv:1912.11930 [cs, math]
- [18] Toshio Endo. 2020. Integrating Cache Oblivious Approach with Modern Processor Architecture: The Case of Floyd-Warshall Algorithm. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia2020)*. Association for Computing Machinery, Fukuoka, Japan, 123–130. <https://doi.org/10.1145/3368474.3368477>
- [19] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 2012. Cache-Oblivious Algorithms. *ACM Transactions on Algorithms* 8, 1 (Jan. 2012), 4:1–4:22. <https://doi.org/10.1145/2071379.2071383>
- [20] Gianluca Frison, Dimitris Kouzoupis, Tommaso Sartor, Andrea Zanelli, and Moritz Diehl. 2018. BLASFEO: Basic Linear Algebra Subroutines for Embedded Optimization. *ACM Trans. Math. Software* 44, 4 (Aug. 2018), 1–30. <https://doi.org/10.1145/3210754>
- [21] Gianluca Frison, Tommaso Sartor, Andrea Zanelli, and Moritz Diehl. 2020. The BLAS API of BLASFEO: Optimizing Performance for Small Matrices. *ACM Trans. Math. Software* 46, 2 (June 2020), 1–36. <https://doi.org/10.1145/3378671> arXiv:1902.08115
- [22] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy Of High-Performance Deep Learning Convolutions On SIMD Architectures. *arXiv:1808.05567 [cs]* (Aug. 2018). arXiv:1808.05567 [cs]
- [23] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Software* 34, 3 (May 2008), 1–25. <https://doi.org/10.1145/1356052.1356053>
- [24] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. 2001. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Software* 27, 4 (Dec. 2001), 422–455. <https://doi.org/10.1145/504210.504213>
- [25] Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefer. 2019. A Fast Analytical Model of Fully Associative Caches. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Phoenix, AZ, USA, 816–829. <https://doi.org/10.1145/3314221.3314606>
- [26] Wolfgang Hackbusch. 2015. *Hierarchical Matrices: Algorithms and Analysis 1st Edition*. Springer Publishing Company, Incorporated ©2015.
- [27] Azzam Haidar, Tingxing Dong, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. 2015. Towards Batched Linear Solvers on Accelerated Hardware Platforms. *ACM SIGPLAN Notices* 50, 8 (Jan. 2015), 261–262. <https://doi.org/10.1145/2858788.2688534>
- [28] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. 2009. Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. 53, 2 (Sept. 2009), 217–288. <https://doi.org/10.1137/090771806> arXiv:0909.4061
- [29] 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 '16)*. IEEE Press, Salt Lake City, Utah, 1–11.
- [30] Torsten Hoefer, William Gropp, William Kramer, and Marc Snir. 2011. Performance Modeling for Systematic Performance Tuning. In *State of the Practice Reports (SC '11)*. Association for Computing Machinery, Seattle, Washington, 1–12. <https://doi.org/10.1145/2063348.2063356>
- [31] Johannes Hofmann, Christie L. Alappat, Georg Hager, Dietmar Fey, and Gerhard Wellein. 2020. Bridging the Architecture Gap: Abstracting Performance-Relevant Properties of Modern Server Processors. *Supercomputing Frontiers and Innovations* 7, 2 (June 2020). <https://doi.org/10.14529/jsfi200204> arXiv:1907.00048
- [32] Johannes Hofmann and Dietmar Fey. 2016. An ECM-based energy-efficiency optimization approach for bandwidth-limited streaming kernels on recent Intel Xeon processors. *arXiv:1609.03347 [cs]* (Sept. 2016). <http://arxiv.org/abs/1609.03347> arXiv: 1609.03347.
- [33] Jianyu Huang, Leslie Rice, Devin A. Matthews, and Robert A. van de Geijn. 2016. Generating Families of Practical Fast Matrix Multiplication Algorithms. *arXiv:1611.01120 [cs]* (Nov. 2016). arXiv:1611.01120 [cs]

- [34] Jianyu Huang, Tyler M. Smith, Greg M. Henry, and Robert A. Van De Geijn. 2016. Strassen's Algorithm Reloaded. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 690–701. <https://doi.org/10.1109/SC.2016.58>
- [35] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. 2004. Sparsity: Optimization Framework for Sparse Matrix Kernels. *The International Journal of High Performance Computing Applications* 18, 1 (Feb. 2004), 135–158. <https://doi.org/10.1177/1094342004041296>
- [36] Dror Irony, Sivan Toledo, and Alexander Tiskin. 2004. Communication Lower Bounds for Distributed-Memory Matrix Multiplication. *J. Parallel and Distrib. Comput.* 64, 9 (Sept. 2004), 1017–1026. <https://doi.org/10.1016/j.jpdc.2004.03.021>
- [37] Lijuan Jiang, Chao Yang, and Wenjing Ma. 2020. Enabling Highly Efficient Batched Matrix Multiplications on SW26010 Many-Core Processor. *ACM Transactions on Architecture and Code Optimization* 17, 1 (March 2020), 1–23. <https://doi.org/10.1145/3378176>
- [38] Kyungjoo Kim, Timothy B. Costa, Mehmet Deveci, Andrew M. Bradley, Simon D. Hammond, Murat E. Guney, Sarah Knepper, Shane Story, and Sivasankaran Rajamanickam. 2017. Designing Vector-Friendly Compact BLAS and LAPACK Kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. Association for Computing Machinery, Denver, Colorado, 1–12. <https://doi.org/10.1145/3126908.3126941>
- [39] Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager, and Gerhard Wellein. 2018. Automated Instruction Stream Throughput Prediction for Intel and AMD Microarchitectures. *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)* (Nov. 2018), 121–131. <https://doi.org/10.1109/PMBS.2018.8641578> arXiv: 1809.00912.
- [40] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. 2016. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Trans. Math. Software* 43, 2 (Aug. 2016), 12:1–12:18. <https://doi.org/10.1145/2925987>
- [41] Ian Masliah, Ahmad Abdelfattah, A. Haidar, S. Tomov, Marc Baboulin, J. Falcou, and J. Dongarra. 2016. High-Performance Matrix-Matrix Multiplications of Very Small Matrices. In *Euro-Par 2016: Parallel Processing*, Pierre-François Dutoit and Denis Trystram (Eds.). Vol. 9833. Springer International Publishing, Cham, 659–671. [https://doi.org/10.1007/978-3-319-43659-3\\_48](https://doi.org/10.1007/978-3-319-43659-3_48)
- [42] R.L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. 1970. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal* 9, 2 (1970), 78–117. <https://doi.org/10.1147/sj.92.0078>
- [43] John D. McCalpin. 1995. Sustainable Memory Bandwidth in Current High Performance Computers.
- [44] Tan M. Nguyen, Vai Suliafu, Stanley J. Osher, Long Chen, and Bao Wang. 2021. FMMformer: Efficient and Flexible Transformer via Decomposed Near-field and Far-field Attention. *arXiv:2108.02347 [cs, math]* (Aug. 2021). <http://arxiv.org/abs/2108.02347> arXiv: 2108.02347.
- [45] Rajesh Nishtala, Richard W. Vuduc, James W. Demmel, and Katherine A. Yelick. 2007. When Cache Blocking of Sparse Matrix Vector Multiply Works and Why. *Applicable Algebra in Engineering, Communication and Computing* 18, 3 (May 2007), 297–311. <https://doi.org/10.1007/s00200-007-0038-9>
- [46] Flame Working Note and Robert A Van De Geijn. 2016. BLISlab : A Sandbox for Optimizing GEMM Lec 6. (2016), 1–16. arXiv:1609.00076v1
- [47] Yu Pei. 2019. Evaluation of Programming Models to Address Load Imbalance on Distributed Multi-Core CPUs: A Case Study with Block Low-Rank Factorization. (Nov. 2019), 12.
- [48] Cody Rivera, Jieyang Chen, Nan Xiong, Shuaiwen Leon Song, and Dingwen Tao. 2021. TSM2X: High-Performance Tall-and-Skinny Matrix-Matrix Multiplication on GPUs. *J. Parallel and Distrib. Comput.* 151 (May 2021), 70–85. <https://doi.org/10.1016/j.jpdc.2021.02.013> arXiv:2002.03258
- [49] S Rjasanow. 2002. Adaptive Cross Approximation of Dense Matrices. In *International Association for Boundary Element Methods*. UT Austin, TX, USA.
- [50] François-Henry Rouet, Xiaoye S. Li, Pieter Ghysels, and Artem Napov. 2015. A distributed-memory package for dense Hierarchically Semi-Separable matrix computations using randomization. 42, 4 (2015). <https://doi.org/10.1145/2930660> arXiv: 1503.05464.
- [51] Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. 2014. Anatomy of High-Performance Many-Threaded Matrix Multiplication. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS '14)*. IEEE Computer Society, USA, 1049–1059. <https://doi.org/10.1109/IPDPS.2014.110>
- [52] Edgar Solomonik and James Demmel. 2011. Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms. In *Euro-Par 2011 Parallel Processing (Lecture Notes in Computer Science)*, Emmanuel Jeannot, Raymond Namyst, and Jean Roman (Eds.). Springer, Berlin, Heidelberg, 90–109. [https://doi.org/10.1007/978-3-642-23397-5\\_10](https://doi.org/10.1007/978-3-642-23397-5_10)
- [53] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2017. The ARM Scalable Vector Extension. *IEEE Micro* 37, 2 (March 2017), 26–39. <https://doi.org/10.1109/MM.2017.35> arXiv:1803.06185

- [54] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Software* 41, 3 (June 2015), 14:1–14:33. <https://doi.org/10.1145/2764454>
- [55] R. Vuduc, J.W. Demmel, K.A. Yelick, S. Kamil, R. Nishtala, and B. Lee. 2002. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. 26–26. <https://doi.org/10.1109/SC.2002.10025>
- [56] R.C. Whaley and J.J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *Proceedings of the IEEE/ACM SC98 Conference*. IEEE, Orlando, FL, USA, 38–38. <https://doi.org/10.1109/SC.1998.10004>
- [57] Karl-Robert Wichmann, Martin Kronbichler, Rainald Löhner, and Wolfgang A Wall. 2019. Practical Applicability of Optimizations and Performance Models to Complex Stencil-Based Loop Kernels in CFD. *The International Journal of High Performance Computing Applications* 33, 4 (July 2019), 602–618. <https://doi.org/10.1177/1094342018774126>
- [58] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures.
- [59] Markus Wittmann, Georg Hager, Thomas Zeiser, Jan Treibig, and Gerhard Wellein. 2016. Chip-Level and Multi-Node Analysis of Energy-Optimized Lattice-Boltzmann CFD Simulations. *Concurrency and Computation: Practice and Experience* 28, 7 (May 2016), 2295–2315. <https://doi.org/10.1002/cpe.3489> arXiv:1304.7664
- [60] Weiling Yang, Jianbin Fang, Dezun Dong, Xing Su, and Zheng Wang. [n.d.]. LibShalom: Optimizing Small and Irregular-Shaped Matrix Multiplications on ARMv8 Multi-Cores. ([n. d.]), 13.
- [61] K. Yotov, Xiaoming Li, Gang Ren, M.J.S. Garzaran, D. Padua, K. Pingali, and P. Stodghill. 2005. Is Search Really Necessary to Generate High-Performance BLAS? *Proc. IEEE* 93, 2 (Feb. 2005), 358–386. <https://doi.org/10.1109/JPROC.2004.840444> Conference Name: Proceedings of the IEEE.
- [62] Chenhan D. Yu, Severin Reiz, and George Biros. 2019. Distributed O(N) Linear Solver for Dense Symmetric Hierarchical Semi-Separable Matrices. In *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc)*. IEEE, Singapore, Singapore, 1–8. <https://doi.org/10.1109/MCSoc.2019.00008>
- [63] Field G. Van Zee, Tyler M. Smith, Bryan Marker, Tze Meng Low, Robert A. Van De Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John A. Gunnels, and Lee Killough. 2016. The BLIS Framework: Experiments in Portability. *ACM Trans. Math. Software* 42, 2 (June 2016), 12:1–12:19. <https://doi.org/10.1145/2755561>