Multilevel Approaches to Fine Tune Performance of Linear Algebra Libraries

Sanaz Gheibi, Tania Banerjee, Sanjay Ranka, and Sartaj Sahni
CISE, University of Florida, Gainesville, FL 32611
{sgheibi, tmishra, sranka, sahni}@ufl.edu

***Abstract*—We propose a multilevel methodology to improve the performance of parallel codes whose run time increases at a faster rate than the increase in workload. We have derived the conditions under which the proposed methodology improves performance for a simple parallel computing model. Formulas to predict the amount of performance improvement that is attainable are also derived for this simple computing model. The effectiveness of the proposed strategy is demonstrated by applying it to the highly optimized BLAS (Basic Linear Algebra Subprograms) routines cblas_dgemm, cblas_dtrmm and cblas_dsymm from the Intel MKL (Math Kernel Library) on the Intel KNL (Knights Landing) platform. We are able to reduce the run time of MKL cblas_dgemm by 20%, cblas_dtrmm by 15%, and cblas_dsymm by 50% on double-precision matrices of size 16Kx16K. Further, our performance prediction formulas are demonstrated to be accurate on this platform.**

Keywords: Parallel speedup, large matrices, performance fine tuning

I. Introduction

For some parallel codes, as the problem size is increased, the run time increases at a faster rate than does the computational workload. This may happen, for example, because of increased communication time. In this paper, we propose a methodology to improve the performance of codes that exhibit this property and demonstrate its effectiveness on 3 example optimized commercial linear algebra codes. The methodology is used to solve a problem at multiple levels, and the target application must be decomposable into smaller instances. For a two-level implementation, the problem is decomposed into smaller pieces where each piece is solved using a one-level algorithm. The concept of a one-level algorithm is different from what is conventionally used. Here, by a one-level algorithm, we mean the original parallel code that exhibits a speedup reduction on large instances. The decomposed pieces may be solved - using the one level algorithm - sequentially, in parallel, or by a hybrid of the two strategies. Finally, the solutions to the subproblems are combined to solve the original problem. For a n level implementation, the problem is divided into smaller pieces, each piece is solved using a $n-1$ level implementation, and finally, the solutions are combined to solve the whole problem.

The described strategy has similarities to divide and conquer, block matrix multiplication, and Canon's submatrix movement scheme [1]. In typical divide and conquer algorithms [2], [3], [4], [5], [6] the decomposition and recombination follow a uniform strategy at all levels of the divide-and-conquer recursion while in our outlined strategy, the way an instance is decomposed and recombined may vary across levels of the decomposition and also within the same level. For example, Strassen's matrix multiplication algorithm [2] uses $O(log\ n)$ levels of dividing a $n \times n$ matrix multiplication problem; where at each level, the division and recombination strategy is the same. Block matrix multiplication (which can be viewed as a single-level divide-and-conquer algorithm) has also been used to improve measured performance on both serial and parallel computers [7], [8], [1]. Cannon's matrix multiplication algorithm, for instance, was developed for mesh connected parallel computers with wraparound connections. Each processor in the mesh begins with a distinct submatrix of A and B, where A and B are the two matrices to be multiplied. Each processor computes a submatrix of the result matrix through a series of rounds. In each round, each processor multiples the A and B submatrices it has, adds the result to the partially computed result submatrix it has. Next, the A submatrices are shifted one left circularly in each row of the mesh, and the B submatrices are shifted one up circularly in each column, and we move on to the next round.

Our strategy differs from a pure divide-and-conquer in that we use potentially different division and recombination schemes at each level. It is different from the block matrix multiply in that we may use more than one level of problem decomposition, and it differs from Cannon's scheme both in the number of division levels and the data movement scheme (we use a serpentine scheme versus Cannon's row and circular column shifts). However, independent of the similarity with techniques used in the past to speed matrix multiplication is the key observation that using our methodology, we can speed up existing (finely tuned) parallel codes whose run time increases at a faster rate than the workload. Thus, the value of our work lies in substantially reducing the run time of fine-tuned code by recursive subdivision and not necessarily in proposing a "new" algorithm design method that has been taught in classrooms for decades. Our methodology could potentially be applied to speed fine-tuned parallel codes for non-linear-algebra codes as the methodology views the parallel code to be sped as a black box.

We develop a framework based on a simple parallel computing model to evaluate the conditions under which the proposed methodology improves performance and also derive formulas to predict performance improvement attainable. We demonstrate our methodology on a matrix-matrix multiplication problem and show that the runtime of MKL cblas_dgemm is reduced by 20% on the KNL platform. Further, the performance

978-1-7281-5341-4/19/$31.00

prediction formulas are shown to be accurate on this platform. Using the same methodology, we also obtained considerable speedup on cblas_dtrmm and cblas_dsymm, though the details of these routines are omitted due to space constraints.

Although the Knights Landing is being discontinued by Intel, our work does not lose relevance because firstly, there are existing KNL clusters at many of the national laboratories which will continue to operate their KNL; and secondly, the work will still be useful for other architectures that implement a high-bandwidth memory. For example, exascale computers are expected to have multiple levels of memory with different bandwidths.

The rest of the paper is organized as follows. Section II presents related work. This is followed by Section III, which describes the generalized problem and an application on matrix-matrix multiplication for matrices of huge sizes. Section IV presents the performance improvements we obtained for matrix-matrix multiplications and shows that the improvement predicted using our framework was very close to the actual improvement obtained. Finally, we conclude in Section V.

II. Related Work on Performance Tuning

There has been wide-scale research on high-performance computing (HPC) applications on parallel systems. We present here some existing work in this category that focuses on application speedup.

Data layout transformation [9], [10], [11], [12] has been widely used to improve memory utilization, and cache hit rate in memory-bound applications. These methods work for multidimensional arrays where there is a canonical compiler-imposed mapping from the array index space to the virtual address space. Data layout transformation works by changing this mapping, which is also referred to as the layout function. In our proposed method, we do not use layout transformation explicitly. As we explain in Section III in more detail, our method repeatedly breaks the problem into smaller subproblems. At the leaf level of the solution tree, we solve the subproblems using a state-of-the-art algorithm. Using or not using data layout transformation depends on the specifications of that algorithm.

Tiling [13], [14], [15], [16] is another widely used method to improve data locality and cache performance. By breaking the problem into subproblems or tiles, the amount of data reuse in the fast levels of cache hierarchy increases. Increasing the cache hit rate improves the overall speed of the application. Our target BLAS routines implement tiling. Our methodology, when applied to BLAS operations, decomposes the matrices as bigger tiles, which are then processed by the target routines.

Synchronization places a big overhead on parallel programs. Therefore, asynchrony is sometimes used as a method to reduce communication costs and increase the speed [17], [18]. Although asynchrony can be very effective in reducing the communication overhead, it is only applicable to a certain class of problems where lack of synchronization does not affect the correctness of the solution. Other methods [19], [20] have used computation and communication overlapping to reduce the communication overhead. These methods, however, are algorithm-specific, and unlike our method, the details of the corresponding fast algorithms should be known.

Another group of methods [21], [22] uses automatic data placement to increase memory efficiency and speedup the memory-bound applications. These methods use compile-time/run-time analysis methods to analyze memory access patterns and find the optimal data placement scheme. We do not use automatic data placement in our method because the real data is not accessed until the very last level. Access patterns can not be determined at compile-time, and using runtime profiling methods introduces much overhead.

III. Proposed Approach

A. Generalized problem

In this paper, the term generalized problem refers to any decomposable problem whose runtime increases at a faster rate than the workload size. Algorithm 1 describes an n_level algorithm where $n > 2$. Here, we solve a problem A of size s. The algorithm divides the original problem A into smaller pieces of size bs. Then through the loop in lines $2-4$, we sequentially call a $(n-1)$-level function on each small piece $A[I]$ and combine the partial results to get the output C. The implementation of GEN_{n-1} is similar. Each $A[I]$ is further subdivided, and a $(n-2)$-level algorithm is called to solve the pieces of $A[I]$. This goes on until a two-level algorithm GEN_2 is reached. Algorithm 2 describes GEN_2, in which we perform the last level of problem size reduction and then solve the pieces using a GEN_1, or a 1-level algorithm. Any state-of-the-art algorithm can be used as GEN_1.

In a k-level algorithm, where $2 < k \leq n$, the subproblems are processed sequentially. Due to space constraints in fast memory, only our two-level algorithm runs the subproblems in parallel.

Algorithm 1 n_level algorithm

```
1: function GEN_n(A, C) ▷ Problem A of size s is divided
   into s/bs blocks of sizes bs.C is the output.
2:     for I = 1 to s/bs do
3:         Call GEN_{n-1} over A[I]
4:     end for
5:     do in parallel
6:         Combine the results of GEN_{n-1}
7:         to form the final solution C
8:     end do
9: end function
```

GEN_2 solves problem $\overline{A}$ of size $\overline{s}$ and produces output $\overline{C}$. In this algorithm, we also input an integer b, which determines the number of pieces of $\overline{A}$ that are to be processed in parallel. Here, we divide $\overline{A}$ into $\overline{s}/\overline{bs}$ blocks each of size $\overline{bs}$. The loop at lines 2 to 14 implements the main body of the algorithm. At each iteration, we copy b pieces from $\overline{A}$ in order to create an input for GEN_1 (lines 3 to 5). Then in lines 6 to 8, we call b parallel instances of GEN_1 over each of these pieces, and in lines 9 to 11, the partial results are summed up to $\overline{C}$.

Algorithm 2 two_level algorithm

1: **function** $GEN_2(\overline{A}, \overline{C}, b)$ ▷ Problem $\overline{A}$ of size $\overline{s}$ is divided into $\overline{s}/\overline{bs}$ blocks of size $\overline{bs}$. $\overline{C}$ is the output and b is the level of parallelism.
2: **for** $\overline{s}/\overline{bs}$ iterations involving I **do**
3: **do in parallel**
4: copy b blocks ($\overline{A}[I]$ to $\overline{A}[I+b-1]$)
5: **end do**
6: **do in parallel**
7: Call b instances of GEN_1 over the b pieces
8: **end do**
9: **do in parallel**
10: Sum up partial results of GEN_1 to $\overline{C}$
11: **end do**
12: $I \leftarrow I + b$
13: **end for**
14: **end function**

Algorithm 3 L_level matrix multiplication algorithm

1: **function** $MXM_L(A, B, C)$ ▷ Input matrices $A_{m\times p}$, $B_{p\times n}$ and output matrix $C_{m\times n}$ are divided into blocks of sizes $bs_1 \times bs_3$, $bs_3 \times bs_2$ and $bs_1 \times bs_2$ respectively.
2: **for** $I = 1$ to m/bs_1 **do**
3: **for** $J = 1$ to n/bs_2 **do**
4: **for** $K = 1$ to p/bs_3 **do**
5: Call MXM_{L-1} over $A[I, K]$ and
6: $B[K, J]$
7: **end for**
8: **do in parallel**
9: Sum up partial results of MXM_{L-1}
10: to $C[I, J]$
11: **end do**
12: **end for**
13: **end for**
14: **end function**

B. Matrix multiplication

In this section, we illustrate how our methodology may be employed to matrix multiplication. Speeding up matrix-matrix multiplication is a well-researched problem. There are many open source as well as proprietary codes that provide outstanding performance on a parallel platform. Some examples of packages providing an efficient implementation of matrix multiplication as part of the Basic Linear Algebra Subroutines (BLAS), are AMD Core Math Library (ACML), OpenBLAS, ATLAS, Intel Math Kernel Library (MKL), and the relevant routine for double-precision matrix-matrix multiplication is cblas_dgemm. Our parallel algorithm leverages cblas_dgemm as a very efficient algorithm to solve subproblems at the leaf level.

In the matrix multiplication problem, matrices A and B of sizes $m \times p$ and $p \times n$ respectively, are multiplied resulting in a matrix C of size $m \times n$. The same methodology is used as the one used for the generalized problem. Algorithm 3 describes the L_level algorithm (MXM_L) where $L > 2$. Here, we break A, B and C into blocks of sizes $bs_1 \times bs_3$, $bs_3 \times bs_2$ and $bs_1 \times bs_2$ respectively.Through the nested loop in lines $2-13$, we iteratively call a lower-level function on $A[I, K]$ and $B[K, J]$ and sum up the partial results to get $C[I, J]$.

MXM_2 is our two-level algorithm for matrix-matrix multiplication. To optimize the block copy operation of MXM_2 on a computer that has a large but slow memory as well as a smaller but fast memory (e.g., our benchmark KNL as well as future exascale computers), we use a serpentine pattern in selecting the b blocks from input matrices $M1$ and $M2$. This reduces the amount of data transferred between the larger slow and smaller fast memories. Figure 1 shows an example of this particular movement when the number of blocks along each direction is 3 and $b = 1$, which means that only 1 pair of blocks are multiplied at the same time. Here, to compute the first row of blocks in the output matrix $M3$, our algorithm fetches blocks from $M1$ by iterating back and forth on the

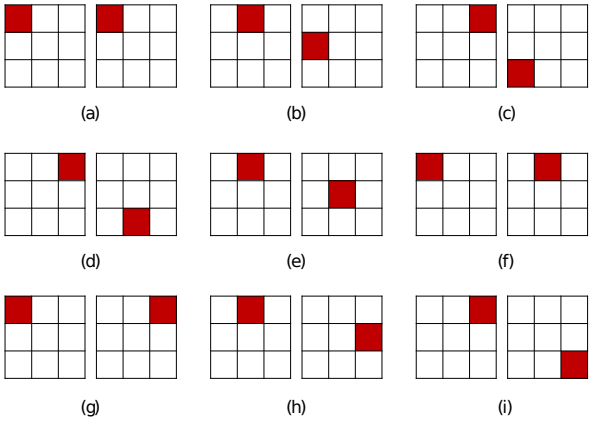

Fig. 1: Parts (a)-(i) show the block choice for computing row 1 in $M3$. In each part, the left matrix is $M1$, and the right matrix is $M2$. The resulting matrix $M3$ is not shown. (a)-(c) compute $M3[1,1]$, (d)-(f) compute $M3[1,2]$ and (g)-(i) compute $M3[1,3]$. Blocks chosen from $M2$ follow a pattern similar to the serpentine movement.

first row of blocks in the input matrix $M1$, while at the same time fetching the corresponding block from $M2$. Each pair of blocks colored red are copied and multiplied in one step. Note that in steps (c) and (d), the third $M1$ block is reused. Similarly, in steps (f) and (g), the first $M1$ block is reused. Note how the block copying in a row alternately iterates between left-to-right and right-to-left to reduce the number of required block copies. In a naive algorithm, the first row of $M1$ is always iterated left-to-right. Then each block on the first row of $M1$ is copied 3 times. Using our scheme, we reduce the number of required copies of $M1$ by 2 for this example. The benefit from reuse gets more significant as the block size is increased and with a larger b.

Algorithm 4 describes our 2-level matrix multiplication algorithm, MXM_2. We divide $M1_{\overline{m}\times\overline{p}}$ and $M2_{\overline{p}\times\overline{n}}$ into blocks of sizes $\overline{bs_1} \times \overline{bs_3}$ and $\overline{bs_3} \times \overline{bs_2}$ respectively. The resulting matrix $M3_{\overline{m}\times\overline{n}}$ will also be divided into blocks of

size $\overline{bs}_1 \times \overline{bs}_2$. We will have:

$$M3[I,J] = \sum_{K=1}^{\overline{p}/\overline{bs}_3} M1[I,K] \times M2[K,J]$$

where I, J and K are indexes for the matrix blocks. Lines 8 to 29 implement the main body of the algorithm where b blocks of $M1[I,K]$ and b blocks of $M2[K,J]$ are copied. Then in lines 22 to 25, b parallel multiplications are done using MKL cblas_dgemm over the corresponding block pairs and in line 26 to 29, the partial results are summed up to $M3[I,J]$. Variables $copy1, copy2, K\_direction$ and $J\_direction$ are used to implement the serpentine pattern. This pattern requires a periodic change of direction from "left-to-right" to "right-to-left" (when moving along the columns) and from "top-to-bottom" to "bottom-to-top" (when moving along the rows). That is why lines 6 and 7 do not implement a direct iteration. Determining the values of J and K and the movement directions are done in lines 30 to 42. Also, blocks from $M1$ and $M2$ only get copied when $copy1$ and $copy2$ are $true$ respectively. These values are only $false$ at the turning points of the serpentine movement when we want to preserve the previously visited block in cache or in a high bandwidth memory.

Thus, original matrices A and B are divided into sub-blocks, each sub-block may be further divided, this division continues based on memory constraints when we finally stop dividing and start multiplying the smallest sub-blocks and combine their results to solve the original matrix multiplication problem eventually.

C. Performance Analysis of Our Methodology

We developed a simple model for parallel computing to derive a formula for predicting the expected performance improvement when our methodology is used. The detail of this analysis is omitted here due to a lack of space but is available at [23].

IV. Experimental Results

We have evaluated the performance improvement of Intel MKL cblas_dgemm on the Intel KNL platform using our methodology. Different configurations for MCDRAM and different clustering modes were combined, and the performance of our algorithm tested. We used OpenMP for parallel processing, and the number of threads was set to 64. Our experiments show that having more threads causes a degradation in performance due to severe resource contention. We used Intel compiler icpc with the following compilation flags: "-O3 -xMIC-AVX512 -mkl -lmemkind -qopenmp". When MCDRAM is used in Flat mode, we allocate memory from MCDRAM using hbw_posix_memalign. We allocate memory from DDR4 using posix_memalign.

The reported run times for matrix sizes up to $16K \times 16K$ are an average over 10 runs. For matrices of sizes $32K \times 32K$ and $64K \times 64K$, the run times have been reported as an average over 5 runs since these runs are time-consuming and the run times are already stable enough for these large sizes.

Algorithm 4 two_level matrix multiplication algorithm

1: **function** $MXM_2(M1, M2, M3, b)$ ▷ In put matrices $M1_{\overline{m}\times\overline{p}}$ and $M2_{\overline{p}\times\overline{n}}$ and the output matrix $M3_{\overline{m}\times\overline{n}}$ are divided into blocks of sizes $\overline{bs}_1 \times \overline{bs}_3$, $\overline{bs}_3 \times \overline{bs}_2$ and $\overline{bs}_1 \times \overline{bs}_2$ respectively. b is the level of parallelism.
2: $copy1 \leftarrow true, \quad copy2 \leftarrow true$
3: $K \leftarrow 1, \quad J \leftarrow 1$
4: $K\_direction \leftarrow 1, \quad J\_direction \leftarrow 1$
5: **for** $I = 1$ to $\overline{m}/\overline{bs}_1$ **do**
6: **for** $\overline{n}/\overline{bs}_2$ iterations involving J **do**
7: **for** $\overline{p}/\overline{bs}_3$ iterations involving K **do**
8: **if** copy1 **then**
9: **do in parallel**
10: copy b blocks from $M1$ starting
11: from $M1[I,K]$ along K_direction.
12: **end do**
13: **end if**
14: **if** copy2 **then**
15: **do in parallel**
16: copy b blocks from $M2$ starting
17: from $M2[K,J]$ along K_direction.
18: **end do**
19: **end if**
20: **do in parallel**
21: Call b instances of $cblas\_dgemm$ over
22: blocks of copied from $M1$ and $M2$
23: **end do**
24: **do in parallel**
25: Sum up partial results of $cblas\_dgemm$
26: to $M3[I,J]$
27: **end do**
28: $K \leftarrow K + b \times Kdirection$
29: $copy1 \leftarrow true$
30: **if** $K = 1$ or $K = \overline{p}/\overline{bs}_3$ **then**
31: $K\_direction \leftarrow -K\_direction$
32: $copy1 \leftarrow false$
33: **end if**
34: **end for**
35: $J \leftarrow J + 1 \times J\_direction$
36: $copy2 \leftarrow true$
37: **if** $J = 1$ or $J = \overline{n}/\overline{bs}_2$ **then**
38: $J\_direction \leftarrow -J\_direction$
39: $copy2 \leftarrow false$
40: **end if**
41: **end for**
42: **end for**
43: **end function**

The reported numbers have an error bound of at most 4%. We used 9 different configurations on KNL, using different combinations of MCDRAM and clustering modes.

A. Results

Table I shows the run time values for the cblas_dgemm which is our 1-level algorithm. The values are measured on different sizes of double precision matrices and on different KNL memory-cluster configurations.

Table II shows the runtime values for our 2-level algorithm. For all the results shown here, we used 16 blocks arranged as a 4×4 grid. At each iteration, 4 parallel cblas_dgemm calls were initiated on the blocks $(b = 4)$. We have tried other blocking arrangements such as 2×2, as well as 2 and 8 parallel cblas_dgemm calls, but the configuration presented here gave us the best results.

Tables III and IV show the run time values for the 3-level and 4-level algorithms respectively. For our third and fourth level we divided the matrices using 2×2 grids and multiplied the resulting blocks serially.

The results in these tables are used to plot Figure 2. For each of the matrix sizes, we picked the best runtime among all average runtimes for the different KNL memory-cluster configurations. We also picked the best average runtime of the one-level code (cblas_dgemm) among all the configurations. Figure 2 shows the comparison results. We used 2-, 3- and 4-level algorithm on $16K \times 16K$, $32K \times 32K$ and $64K \times 64K$ matrices, respectively. We got an improvement of 20.5% on the 2-level algorithm, 19.0% on the 3-level algorithm, and 16.2% on the 4-level algorithm relative to the 1-level cblas_dgemm code for the same problem.

Using the same methodology, we were also able to speed up other MKL BLAS routines such as cblas_dtrmm (triangular matrix multiplication) and cblas_dsymm (symmetric matrix multiplication). For the former, a speedup of 15% was observed, and for the latter, the speedup was 50%; in both cases, the matrix size was (double-precision) 16Kx16K.

V. Conclusions

We proposed a multilevel methodology to improve the performance of parallel codes whose run time increases at a faster rate than the actual workload. This methodology is applicable to problems where a large instance may be solved by decomposing into smaller instances. The methodology treats the original fine-tuned parallel code as a black box and makes no change to it. For the proposed methodology, we used a simple parallel computing model to derive the conditions under which our strategy improves performance and further derived formulas to predict the amount of performance improvement that is attainable using this methodology. Although, in this paper we demonstrated the effectiveness of our methodology by applying it to the highly optimized BLAS routines cblas_dgemm, cblas_dtrmm and cblas_dsymm on the Intel KNL platform using 64 processors, the methodology itself is quite general and could potentially be used to speed parallel codes for other applications and other platforms.

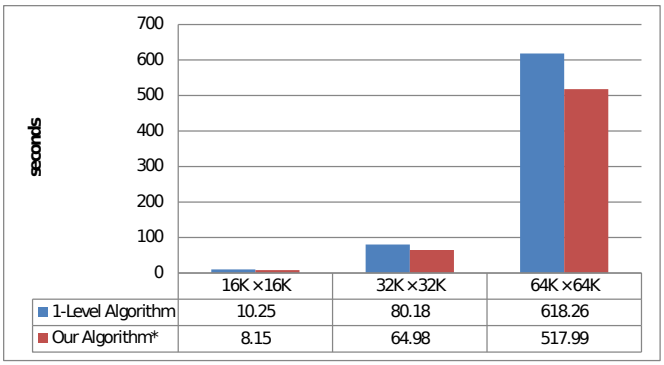

Fig. 2: Comparison of the best runtimes among all average runtimes taken for the different KNL memory-clustering configurations for each matrix size. In case of the results for "our algorithm", the runtime for $16K \times 16K$ matrices was obtained using a 2-level algorithm, for $32K \times 32K$ matrices using a 3-level algorithm and for $64K \times 64K$ matrices using a 4-level algorithm.

We could reduce the run time of MKL cblas_dgemm by 20%, cblas_dtrmm by 15% and cblas_dsymm by 50% on double-precision matrices of size 16Kx16K. We repeated each experiment several times, and the runtime in each case was within 4% of the average runtime. We tried 9 configurations of memory and clustering modes on the KNL platform. As shown in the complete version of the paper [23], the condition we derived for performance improvement using our theoretical models correctly predicted if the performance gets improved or not for 8 out of 9 configurations. Further, the prediction of the amount of performance increase or decrease was within 1.2% of the actual values for all the 8 configurations when the problem size was $64K \times 64K$ (Flat-SNC4 is the configuration where we failed in either case).

Acknowledgment

This work was funded, in part, by the National Science Foundation, under Contract No. 1748652.

References

[1] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," Ph.D. dissertation, Montana State University-Bozeman, College of Engineering, 1969.

[2] V. Strassen, "Gaussian elimination is not optimal," *Numerische mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[3] H. Prokop, "Cache-oblivious algorithms," Ph.D. dissertation, Massachusetts Institute of Technology, 1999.

[4] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall, "An analysis of dag-consistent distributed shared-memory algorithms," in *SPAA*, vol. 96, 1996, pp. 297–308.

[5] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-optimal parallel recursive rectangular matrix multiplication," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 261–272.

[6] B. Lipshitz, G. Ballard, J. Demmel, and O. Schwartz, "Communication-avoiding parallel strassen: Implementation and performance," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

[7] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU press, 2012, vol. 3.

TABLE I: Average run time (in seconds) of 1-level algorithm using cblas_dgemm

matrix sizes	flat-snc4	flat-snc2	flat-all2all	flat-quadrant	flat-hemisphere
$4K \times 4K$	0.65	0.39	0.30	0.33	0.30
$8K \times 8K$	3.41	1.99	1.64	1.49	1.39
$16K \times 16K$	23.21	15.42	12.74	11.92	10.25
$32K \times 32K$	119.81	121.47	102.53	101.58	102.63
$64K \times 64K$	1002.30	843.25	843.04	826.23	824.15
matrix sizes	cache-snc4	cache-snc2	cache-all2all (not supported in KNL)	cache-quadrant	cache-hemisphere
$4K \times 4K$	0.48	0.42	—	0.32	0.29
$8K \times 8K$	1.94	1.71	—	1.51	1.39
$16K \times 16K$	10.89	11.32	—	12.32	10.36
$32K \times 32K$	88.90	87.39	—	97.52	80.18
$64K \times 64K$	618.26	781.83	—	871.10	652.60

TABLE II: Average run time (in seconds) of 2-level algorithm

matrix sizes	flat-snc4	flat-snc2	flat-all2all	flat-quadrant	flat-hemisphere
$4K \times 4K$	0.36	0.32	0.30	0.30	0.28
$8K \times 8K$	2.42	1.52	1.27	1.21	1.25
$16K \times 16K$	113.51	14.82	10.12	8.87	8.85
matrix sizes	cache-snc4	cache-snc2	cache-all2all (not supported in KNL)	cache-quadrant	cache-hemisphere
$4K \times 4K$	0.31	0.29	—	0.30	0.32
$8K \times 8K$	1.62	1.36	—	1.17	1.21
$16K \times 16K$	11.71	11.54	—	8.15	8.61

TABLE III: Average run time (in seconds) of 3-level algorithm

matrix sizes	flat-snc4	flat-snc2	flat-all2all	flat-quadrant	flat-hemisphere
$32K \times 32K$	151.12	119.23	80.62	70.50	70.28
$64K \times 64K$	10540.52	884.25	768.97	720.37	653.97
matrix sizes	cache-snc4	cache-snc2	cache-all2all (not supported in KNL)	cache-quadrant	cache-hemisphere
$32K \times 32K$	93.53	94.26	—	64.98	68.35
$64K \times 64K$	729.14	805.44	—	686.36	651.94

TABLE IV: Average run time (in seconds) of 4-level algorithm

matrix size	flat-snc4	flat-snc2	flat-all2all	flat-quadrant	flat-hemisphere
$64K \times 64K$	1292.55	945.01	640.35	560.46	557.92
matrix size	cache-snc4	cache-snc2	cache-all2all (not supported in KNL)	cache-quadrant	cache-hemisphere
$64K \times 64K$	744.34	745.16	—	517.99	544.31

[8] R. A. Van De Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[9] J. Mellor-Crummey, D. Whalley, and K. Kennedy, "Improving memory hierarchy performance for irregular applications using data and computation reorderings," *International Journal of Parallel Programming*, vol. 29, no. 3, pp. 217–247, 2001.

[10] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi, "Nonlinear array layouts for hierarchical memory systems," in *Proceedings of the 13th international conference on Supercomputing*. ACM, 1999, pp. 444–453.

[11] E. Athanasaki and N. Koziris, "Fast indexing for blocked array layouts to improve multi-level cache locality," in *Interaction between Compilers and Computer Architectures, 2004. INTERACT-8 2004. Eighth Workshop on*. IEEE, 2004, pp. 107–119.

[12] C. Kulkarni, C. Ghez, M. Miranda, F. Catthoor, and H. De Man, "Cache conscious data layout organization for embedded multimedia applications," in *Proceedings of the conference on Design, automation and test in Europe*. IEEE Press, 2001, pp. 686–693.

[13] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. Pudov, V. O. Pirogov, and P. Dubey, "Parallel efficient sparse matrix-matrix multiplication on multicore platforms," in *International Conference on High Performance Computing*. Springer, 2015, pp. 48–57.

[14] D. I. Lyakh, "An efficient tensor transpose algorithm for multicore cpu, intel xeon phi, and nvidia tesla gpu," *Computer Physics Communications*, vol. 189, pp. 84–91, 2015.

[15] C. Yount and A. Duran, "Effective use of large high-bandwidth memory caches in hpc stencil computation via temporal wave-front tiling," in *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), International Workshop on*. IEEE, 2016, pp. 65–75.

[16] Q. Xiangzhen, "Cache performance and algorithm optimization," in *High Performance Computing on the Information Superhighway, 1997. HPC Asia'97*. IEEE, 1997, pp. 12–17.

[17] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in neural information processing systems*, 2011, pp. 693–701.

[18] W.-S. Chin, Y. Zhuang, Y.-C. Juan, and C.-J. Lin, "A fast parallel stochastic gradient method for matrix factorization in shared memory systems," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 6, no. 1, p. 2, 2015.

[19] P. D'alberto, M. Bodrato, and A. Nicolau, "Exploiting parallelism in matrix-computation kernels for symmetric multiprocessor systems: Matrix-multiplication and matrix-addition algorithm optimizations by software pipelining and threads allocation," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 2, 2011.

[20] J. Huang, T. M. Smith, G. M. Henry, and R. A. van de Geijn, "Strassen's algorithm reloaded," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 59.

[21] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu, "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 513–522.

[22] G. Chen, B. Wu, D. Li, and X. Shen, "Porple: An extensible optimizer for portable data placement on gpu," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 88–100.

[23] S. Gheibi, T. Banerjee, S. Ranka, and S. Sahni, "Multilevel Approaches to Fine Tune Performance of Linear Algebra Libraries," http://www.cise.ufl.edu/~sahni/papers/MXM_paper_IEEE.pdf, 2019.