Performance Issues of SYRK implementations in Shared Memory Environments for Edge Cases

Md Mosharaf Hossain
Computer Science
Tennessee Technological University, TN
mhossain44@students.tntech.edu

Thomas M. Hines
Computer Science
Tennessee Technological University, TN
tmhines42@students.tntech.edu

Sheikh K. Ghafoor
Computer Science
Tennessee Technological University, TN
sghafoor@tntech.edu

Ryan J. Marshall
Computer Science
Tennessee Technological University, TN
rmarshall42@students.tntech.edu

Muzakhir S. Amanzholov
Computer Science
Tennessee Technological University, TN
msamanzhol42@students.tntech.edu

Ramakrishnan Kannan
Oak Ridge National Laboratory, TN
kannanr@ornl.gov

***Abstract*—The symmetric rank-k update (SYRK) is a level-3 BLAS routine commonly used by many Data Mining/Machine Learning(DM/ML) algorithms such as regression, dimensionality reduction algorithms like PCA, matrix factorization and k-mean clustering. This paper presents a comprehensive analysis of the SYRK routine under popular dense linear algebra libraries such as OpenBLAS, Intel MKL, and BLIS particularly focusing on edge cases of dense matrices (thin or fat shapes) that are common in DM/ML applications. Our work identifies some performance issues of the SYRK routine in multi-threaded shared memory environments for edge cases and discuss matrix dependent modifications for performance improvement.**

***Index Terms*—Performance Issues of SYRK, BLAS, Multicore, Scalability of AA^T**

I. Introduction

The two prominent challenges in DM/ML domain are improving (a) the quality of the results and (b) ability to handle big datasets. In the latter case, researchers look at novel ways to develop approximation algorithms that attempt to look at the subset of the samples or the features, at a compromise of some quality of the solution. Since most of the DM/ML algorithms are based on stochastic convergence, establishing the expected convergence of these approximation algorithms is always challenging. Also, these algorithms come with heavy assumptions about the distribution of the underlying data and the number of hyper-parameters. It requires a deep DM/ML expertise to reproduce the algorithmic results on a different dataset. This is a consistent bottleneck toward the applicability of algorithms in the real world.

To overcome this problem, HPC researchers are investing in parallel and distributed algorithms on shared and distributed memory, extreme threaded and throughput devices like GPUs to build computationally expensive DM/ML algorithms with less hyper-parameters that still handle huge volumes of scientific [1] and internet data. There are some recent attempts on various hyper-parameter free DM/ML algorithms in the HPC community such as NMF[2], [3], [4], SVM [5], Graph Algorithms [6] and Convolutional Neural Networks [7] using dense Basic Linear Algebra Subprograms(BLAS) for matrix operations. In general, the BLAS operations are fine-tuned for the specific architecture for common matrices that are assumed to be square where the number of rows and columns are nearly same. However, these fine tunings fail for the following edge cases (where an $m \times k$ matrix is thin or fat) degrading the performance:

- Case $m \gg k$ (*thin*): These are classical cases of machine learning, where input data contains billions of samples having hundreds of features (e.g., the profile information of customers). Typically an enterprise will have millions of customers and collect numerous features such as name, age, sex, address, phone number, etc.
- Case $m \ll k$ (*fat*): There are scientific experimental use cases where it is very expensive to conduct repeated trials, particularly when a single trial generates terabytes of data. Neutron scattering and molecular dynamics experiments often belong to this category.

In this paper, we primarily look at the Symmetric Rank k Update (SYRK) algorithm. Given a matrix C of size $m \times m$, update C with a symmetric rank k matrix ($C = \alpha AA^T + \beta C$). Many ML algorithms such as Regression, SVM, PCA, and K-Means use a SYRK kernel. A detailed list of these types of algorithms can be found at [8].

The main contributions of this paper are the benchmarking and investigation of the performance and scalability of the Double precision SYRK (DSYRK) algorithm in Level-3 BLAS library implementations for thin and fat dense matrices (called edge cases) in multicore shared memory environments. We consider only the cases where the matrix fits in the main

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

memory. We additionally give a detailed analysis of the reasons the matrix block pack algorithm gives poor performance on fat matrices as well as potential modifications to mitigate the issue.

II. Related Work

Researchers have been studying the performance of matrix operations using Level-3 BLAS implementations in parallel environments. Studies such as [9], [10] focus on square matrices and/or square blocking factors that could be optimal for a large subset of input data, but not necessarily for edge cases where the matrices are thin or fat. However, their approaches to hardware-based tuning and performance observations can be useful in the development of our experiments.

The particular strategy used in [10] involved leveraging the Block Packed Format, which resulted in a dramatic performance increase due to increased flexibility in the register blocking factor of the default underlying Level-2 BLAS routines. The introduction of replacements for factorization routines based on the Level-2 BLAS routines POTF2 and PPTF2 for cases of symmetric positive-definite matrix-matrix operations improved the dependent Level-3 BLAS routines of a Cholesky factorization. Tarca [11] showed that the performance in Cholesky and related algorithms varied greatly when different pivoting strategies were used and demonstrated under certain pivoting strategies, the more indefinite a matrix, the more pivots are required. He claims an ideal strategy would be to allow the algorithm to change pivoting strategies based on matrix characteristics and hardware architecture, and flexibility in choosing block size, based on architecture and matrix shapes.

Early work in the area of dense linear algebra (DLA) optimization has shown that low-level storage strategies like multilevel blocking in LU and Gram-Schmidt algorithms can increase efficiency, and is transferable both to other problems and other architectures [12]. Other works have shown that optimizing a particular Level-3 BLAS routine like GEMM can transfer well to related routines like SYRK and TRSM. In [13], the authors optimized a GEMM routine and introduced two parameters called r and c to adjust the blocking dimensions, depending on the cache size and layout. By adding these parameters, the optimized routine was portable across different architectures, and its performance was comparable to hand-tuned versions. Additionally, this method was shown to be transferable to other GEMM-based routines like DTRSM.

Goto and Geijn [14] described how the application of techniques used to develop a fast implementation of GEMM can be carried over to improve the performance of related Level-3 BLAS routines such as SYMM and SYRK. The authors call for the abstraction of "building blocks" to reduce the redundant packing operations. Since a number of constant values used by Level-3 BLAS are set at build time, determining the task granularity at runtime depends on increased parameterization of hybrid algorithms for DLA solvers, specifically the parameterization of the underlying Level-2 BLAS routines to change or override the constants. Coupled with performance metrics of the specific algorithm (either from a database of benchmarks or from runtime analysis from a previous run), a dynamic scheduler can offload tasks to appropriate or optimal compute devices. Because modern GPUs support double precision (slower than single precision), tasks can be offloaded to the single precision unit, while using an iterative refinement technique on the double precision unit to test for accuracy. Using such a strategy for Cholesky and LU to overlap communication and computation, a speedup of 100 was achieved by adding 4 GPUs to a 4-core CPU [15]. A redesign of the xSYRK routine for Cholesky factorization to introduce data persistence, dynamic scheduling and limit the scheduling to the lower diagonal via block index reordering resulted in a speedup of 54 for single precision and 27 for double precision using 4 GPUs [16].

Double precision Level-3 BLAS operations such as DGEMM, DSYMM, DHEMM, DSYRK, DTRMM and DTRSM [17] are commonly used in scientific computing applications. The objective of our research is to investigate the scalability characteristics of the DSYRK algorithm in popular BLAS libraries on shared memory multicore for edge cases. We have designed and performed sets of experiments which is described in the following sections.

III. Experimental Design and Results

The purpose of the experiments was to identify the performance of SYRK for a range of matrix sizes, shapes, and thread distributions and their impact on performance using common DLA libraries.

A. Design

For our experiments, we used three dense linear algebra libraries: OpenBLAS 0.3.0 [18], [19], Intel MKL 2017 [20], and BLIS 0.2.2 [21] in a shared memory multicore environment. We conducted two sets of experiments. The first set of experiments performed AA^T operations using the DSYRK algorithm on a range of matrices – fat, square and thin. We wanted the different shapes to be comparable, so the matrices were chosen such that the number of floating point operations were nearly identical across different cases. We considered a square matrix $m \times k$ as 8192×8192 and reshaped as fat matrices by gradually decreasing m and a corresponding increase of k. Similarly, for thin matrices, we decreased k, the column dimension. This set of runs was repeated five times, with core count 1, 10, and 20, using each of the three libraries. The core counts were chosen due to the compute node having two sockets with 10 cores each. The objective of these experiments was to investigate the impact of matrix shape on performance.

The second set of experiments picked three representative matrix shapes (for fat, square, and thin) and varied the core count from 1 to 20. The three matrices were: 8192×8192 (square), 65536×128 (thin), and 256×8388608 (fat). With this set of experiments, we wanted to investigate the scalability characteristics of different matrix shapes.

We conducted all experiments on a compute node with two Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80 GHz processors (10 cores/processor, 20 cores total and 128 GB RAM). L1, L2, and L3 cache sizes are 32 KB, 256 KB and 25600 KB respectively.

B. Results

Figure 1 shows the performance of DSYRK on one core for the range of computationally equivalent multiplications. If the matrix multiplication is compute bound, as is generally assumed, the performance should remain steady as the matrix shape changes. However, there is a degradation in performance as the matrix shape deviates from the square. This degradation is more severe for the fat matrices. The 128 wide matrix achieves 90% the GFLOPS of the square, while the 128 tall matrix only achieves 60%.

Figure 2 shows the same run on 10 cores. The same pattern of performance degradation occurs here, but the degradation is more pronounced. The 128 wide matrix has dropped to 80% of the square, and the 128 tall matrix has dropped to 30%. This increases even further when twenty cores are used in Figure 3. OpenBLAS and BLIS perform roughly the same. Intel MKL in general performs worse than the other two with the major exception of very fat matrices on ten and twenty cores. It appears Intel MKL makes some change when $m < 512$ as there is a distinct jump in performance below that threshold.

Figure 4 shows the GFLOPS of DSYRK on the three matrix shapes as the number of cores increases from one to twenty. The square and thin matrices show a linear performance increase with increasing cores, with the thin matrix increasing at a lower rate. The fat matrix with Intel MKL is also linear, albeit at an even lower rate than the thin or square matrices. However, the fat matrix on OpenBLAS and BLIS has a markedly sub-linear performance curve. Beyond ten cores there is little to no performance change.

Three broad trends can be identified from both of these experiments:

- Performance of non-square matrices is worse than square matrices.
- Fat matrices perform worse than thin matrices.
- The difference between square and non-square performance increases with a higher number of cores.

It is also important to note the magnitude of the performance degradation. A very fat matrix can result in performance under 20% of the square matrix.

IV. Analysis

For further investigation of the scalability issues we profiled SYRK using the Intel VTune profiling tool. We then analyzed the BLIS implementation of SYRK.

A. Profiling DSYRK using Intel VTune

We used Intel VTune Amplifier 2017 [22] to investigate the delay related to memory shifting between cache and DRAM. Our primary interest was in VTune's Memory Bound and LLC Miss (Last-Level Cache miss) parameters. Intel defines memory bound as the fraction of time where pipelines could be stalled due to the high demand of load or store instructions [22]. LLC is the last and longest-latency level in the memory hierarchy before main memory - in our case, L3 cache. Any memory requests that are not in the LLC must be serviced by local or remote DRAM, with significant latency.

In Figure 5a, we observed that while running on a single core, a fat matrix (500 x 3,996,000) analysis reports it to be 27.2% memory bound, which is much higher than the almost square matrix (9,000 x 12,357). As the number of cores increases, so does the percentage gap. The results in Figure 5b show that SYRK the fat matrix needs to fetch data from DRAM instead of L3 cache an order of magnitude more times.

These figures show that the gap in performance between fat and square matrices can be at least partially attributed to issues with memory access. BLAS libraries use the block pack algorithm in order to reduce memory access times and improve scalability so this is a logical next place to explore.

B. Block Pack Algorithm Analysis

SYRK in BLIS (and OpenBLAS) is a specialization of the GEMM (GEneral Matrix Multiplication) algorithm where some multiplications are skipped as the output matrix is known to be symmetric. For clarity, the GEMM algorithm is described in this paper. GEMM is implemented by block pack matrix multiplication [23], described in Algorithm 1 and shown in Figure 6. It starts with an A matrix to be multiplied by a B matrix. In order to make use of cache and vector instructions, blocks of A and B are repackaged so that the values in the block are beside each other in memory.

Algorithm 1 GEMM Kernel

```
procedure GEMM
  for $j_c = 0, .., n-1$ with inc $n_c$ do                ▷ Loop 1
    for $p_c = 0, .., k-1$ with inc $k_c$ do              ▷ Loop 2
      Pack block of B into $\hat{B}$
      for $i_c = 0, .., m-1$ with inc $m_c$ do            ▷ Loop 3
        Pack block of A into $\hat{A}$
        for $j_r = 0, .., n_c-1$ with inc $n_r$ do        ▷ Loop 4
          for $i_r = 0, .., m_c-1$ with inc $m_r$ do      ▷ Loop 5
            Multiply the slice of $\hat{A}$ by the
            slice of $\hat{B}$
```

Loops 1 and 2 in GEMM iterate over the grid of blocks in B and create a packed block, $\hat{B}$, of size $k_c \times n_c$. Loop 3 iterates over the column in A that corresponds to the current $\hat{B}$ and creates a repacked block from A, $\hat{A}$, of size $m_c \times k_c$. $\hat{A}$ and $\hat{B}$ are created at an appropriate size to fit into the L2 and L3 caches, respectively. Slices of $\hat{A}$ (of size $m_r \times k_c$) and slices of $\hat{B}$ (of size $k_c \times n_r$) are taken in loops 4 and 5; these are sized to fit into the L1 cache. The slices are then multiplied together in a vectorized assembly function known as the kernel or micro-kernel and added to the corresponding section (of size $m_r \times n_r$) in the output matrix. The idea behind the sizing of the blocks is that for the entirety of loops 4 and 5, no data needs to be transferred from DRAM. $\hat{B}$ should remain

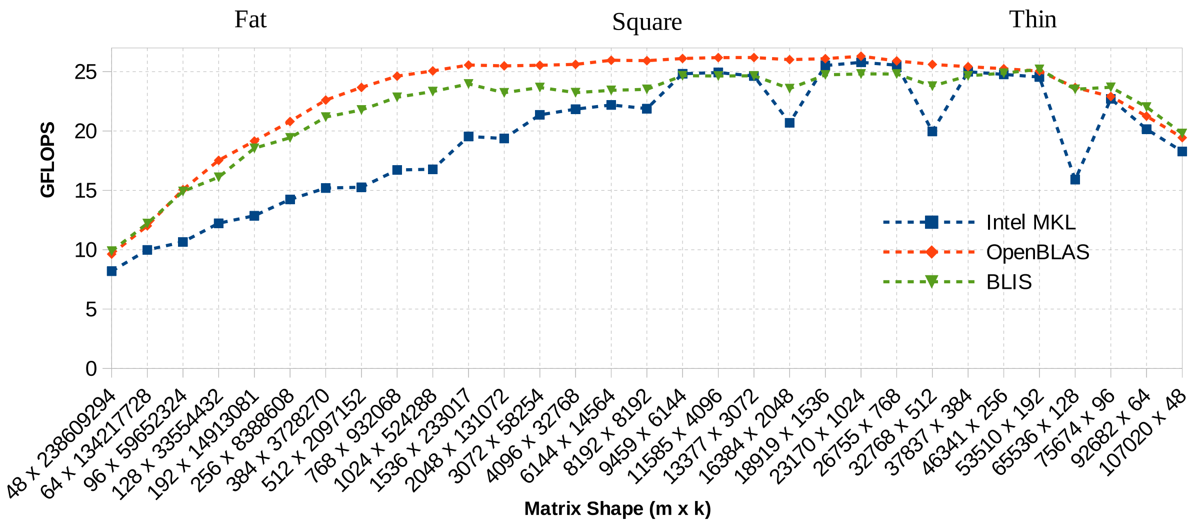


Fig. 1: *The GFLOPS of DSYRK using OpenBLAS, BLIS, and MKL on a range of matrices of shape m by k using one core.*

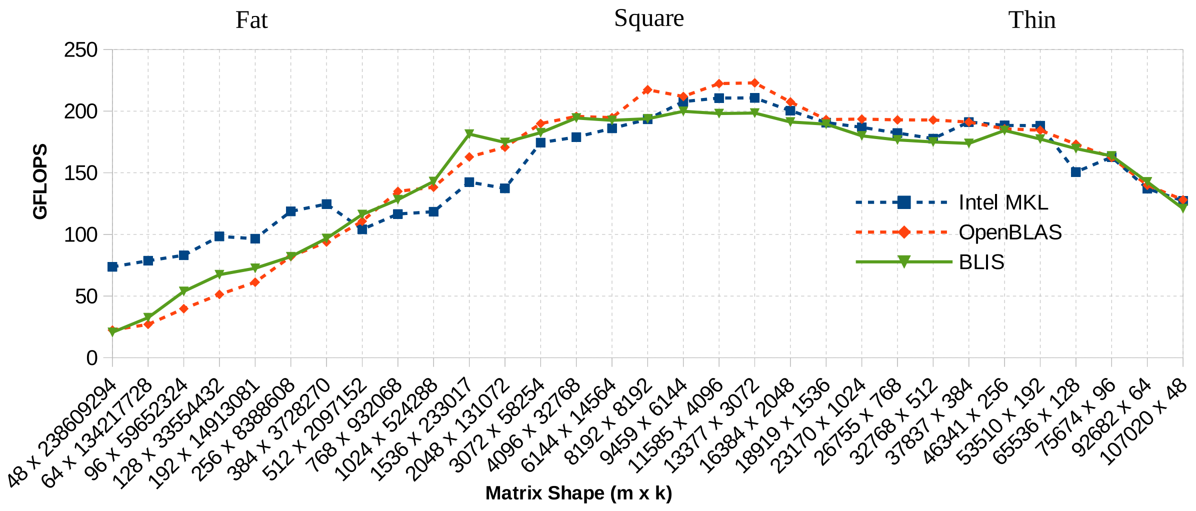


Fig. 2: *The GFLOPS of DSYRK using OpenBLAS, BLIS, and MKL on a range of matrices of shape m by k using ten cores.*

in L3 cache while slices are taken of it, and likewise for $\hat{A}$ and L2 cache. All but one of the loops can be multi-threaded. If the second loop were to be parallelized then there would be multiple $\hat{B}$s in the same column being multiplied at the same time. This would necessitate thread blocking as multiple threads try to write to the same section of the output matrix.

Most dense linear algebra (DLA) libraries do not allow the block sizes or the threading scheme to be changed. BLIS, however, does allow these parameters to be modified. Thus, it was used to discover how changing the parameters affects performance.

The most significant difference between SYRK with a square matrix and with a fat matrix occurs when creating the $\hat{B}$ block. The default size of $\hat{B}$ is 256 by 4096 in BLIS for our architecture (Ivy Bridge). However, as the transposed fat matrix, B, has far fewer than 4096 columns, this results in a much smaller $\hat{B}$. As the size of the block was chosen so that it fills the L3 cache, this results in poor L3 utilization for the fat matrix case. Presumably then, increasing the number of rows of $\hat{B}$, k_c, could mitigate this low L3 cache usage. The downside to this is increasing k_c increases the size of the slices sent to the micro-kernel. This makes the slices too large to fit into the L1 cache as they were intended to do. There is another way to increase L3 cache utilization - parallelize the first loop. This leads to multiple $\hat{B}$ blocks at the same time (instead of serially). However, it is important note that this does not increase the size of the (now multiple) $\hat{B}$s' memory size in L3. The first loop splits B into columns with one for each thread. The $\hat{B}$s each now have an even smaller number of columns, and the multiple $\hat{B}$s add up to the same size as the previous single $\hat{B}$. However, each $\hat{B}$ has its own $\hat{A}$ (or more depending on the parallelization of the third loop) and

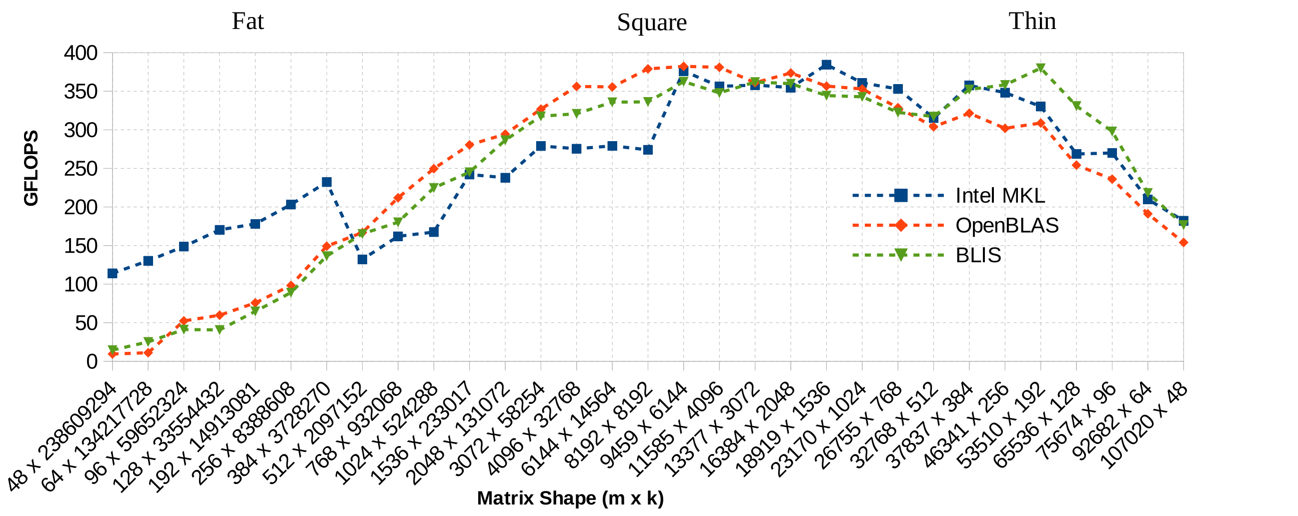


Fig. 3: *The GFLOPS of DSYRK using OpenBLAS, BLIS, and MKL on a range of matrices of shape m by k using twenty cores.*

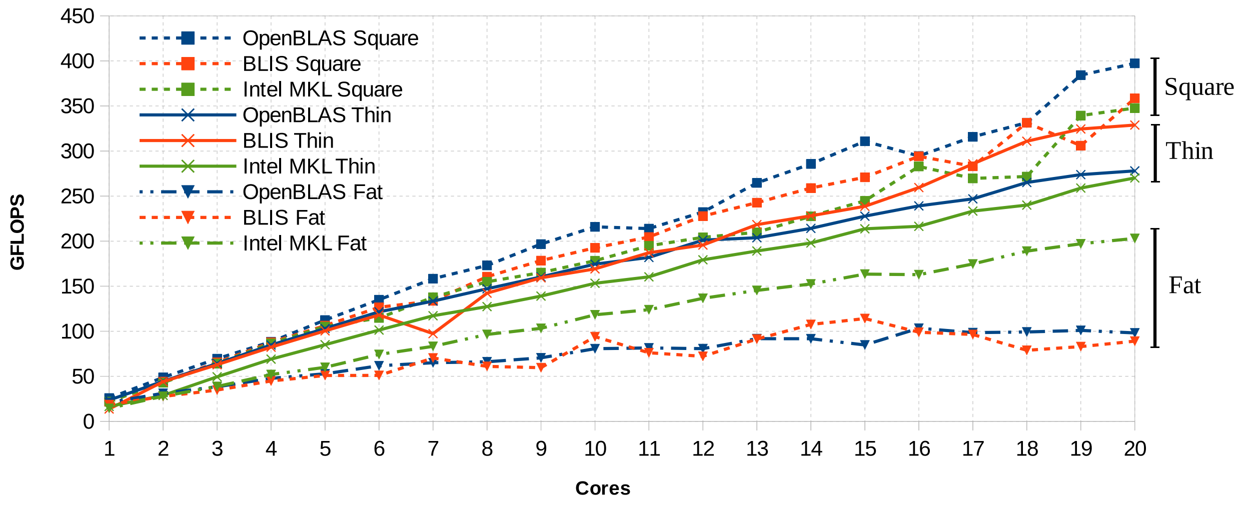


Fig. 4: *The GFLOPS of DSYRK using OpenBLAS, BLIS, and MKL for three matrix shapes on one through twenty cores.*

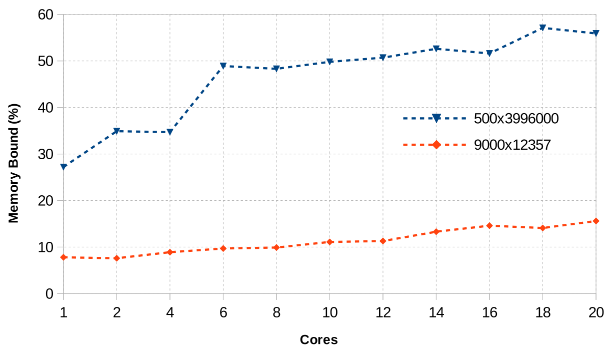
this is where the extra utilization actually comes from. As a side note, even though there are multiple $\hat{B}$s as in a multi-threaded second loop, they are in separate columns and so do not contend for the same section of the output matrix.

Figure 7 shows the effects of varying both $\hat{B}$ row size, k_c, and the threading for the loops. The numbers for each threading scheme represent the parallelization of loops 1, 3, 4, and 5 in that order. The default threading scheme in BLIS is $\{1\ 10\ 2\ 1\}$. Changing the row size of $\hat{B}$ gives a substantial improvement, especially for the default threading. Increasing the parallelization gives an even better improvement. Notice that as the outer loop threading helps fill up the L3 cache, larger $\hat{B}$ row size does not improve computation time nearly as much.

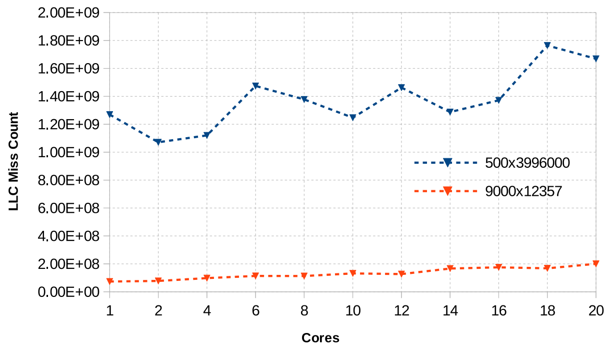
It is difficult to determine how much of the L3 cache is being used on a processor. Instead, we can estimate it based on the size and number of $\hat{A}$s and $\hat{B}$s that are in use at any one time. The first loop parallelization is the number of $\hat{B}$s; the third loop parallelization is the number of $\hat{A}$s per $\hat{B}$. The size of the blocks is determined by m_c, k_c, and n_c in addition to the limitations imposed by m and k. Table I shows the relationship between this estimated percent L3 cache utilization and the GFLOPS achieved. The *Default* rows show the parameters BLIS uses. The *Optimized* rows show parameters that gave

TABLE I: *DSYRK L3 cache utilization with different block-sizes and threading.*

	m **x** k	**Threading**	m_c	k_c	**GFLOPS**	**L3%**
Default	256 x 8388608	1 10 2 1	96	256	137	4%
Optimized	256 x 8388608	4 5 1 1	48	512	176	19%
Default	65536 x 128	1 10 2 1	96	256	353	20%
Optimized	65536 x 128	4 5 1 1	288	1280	399	87%



(a) Memory bound



(b) LLC miss count

Fig. 5: *VTune Analysis of the two representative cases. In the left, Figure 5a shows the percentage of time $DSYRK$ was memory bound on Intel MKL 2017 for a fat and a square matrix. The right side Figure 5b shows the LLC miss count*

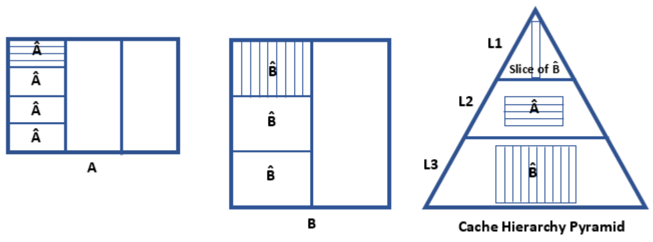


Fig. 6: *The GEMM block matrix partitioning showing the blocking pattern and the storage of blocks in cache*

the shortest time from a sweep of different block-sizes and threadings. In both the fat and thin cases, this optimization results in increased L3 cache usage.

These changes are tweaks to an algorithm that was not designed for fat matrices and do not solve the underlying problem. Increasing the $\hat{B}$ row size quickly leads to massive slices that no longer fit into L1 cache. Over threading the outer loop also leads to reduced performance as Figure 7 shows for 20 threads in the outer loop. One major problem that cannot be fixed without reworking the algorithm is that $\hat{B}$ is supposed to be large - much larger than $\hat{A}$, and fill a significant portion of the L3 cache. This is simply cannot be done for fat matrices as $\hat{A}$ and $\hat{B}$ are very close in size. This means either the L3 cache is underused or the higher caches are overfilled.

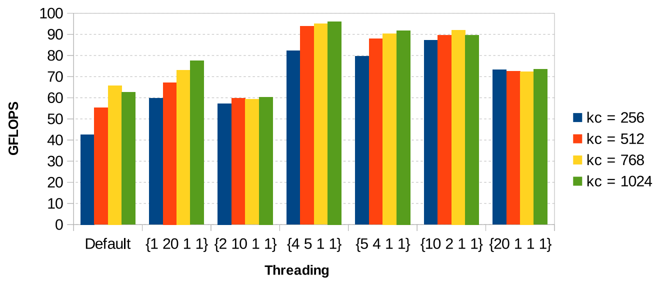


Fig. 7: *Comparison of GFLOPS achieved by DSYRK for a* $128 \times 100M$ *matrix in BLIS 0.2.2 using* 20 *threads by varying the* $\hat{B}$ *row size(*k_c*) and the threading strategy. The default size of* $\hat{A}$ *and* $\hat{B}$ *are* 96×256 *and* 256×4096 *respectively. The default threading used is* $\{1\ 10\ 2\ 1\}$.

V. Acknowledgements

This manuscript has been co-authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. This project was partially funded by the Laboratory Director's Research and Development fund. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the USDOE.

VI. Conclusion and Future Work

We first benchmarked the SYRK algorithm on matrices of different shapes using common linear algebra libraries. As illustrated in Figure 1, Figure 2, and Figure 3, performance issues exist when using the SYRK routine on fat and thin matrices, and the issue is persistent across multiple DLA libraries. Additionally, we showed in Figure 4 that the performance issues with fat and thin matrices increase as the number of cores is increased. The results shown in Figures 5a and 5b identified the problem to be related to cache use. Modifying the parameters of SYRK to reduce the cache usage problem resulted in performance improvements as shown in Figure 7 and Table I.

We conclude that the performance of SYRK in current shared memory libraries depends on the matrix shape. This is especially noticeable in the case of fat matrices. We see three major directions for our future work:

- Determine how to predict the performance of a block shape and threading. This would allow dense linear algebra libraries to automatically change the parameters to the optimum for a specific matrix.
- Expand the benchmarking to GPUs, other CPU architectures, and other BLAS routines.

- Redesign the block-pack algorithm in order to increase performance on edge cases beyond what is possible by modifying the block size and threading parameters.

REFERENCES

[1] R. Kannan, A. V. Ievlev, N. Laanait, M. A. Ziatdinov, R. K. Vasudevan, S. Jesse, and S. V. Kalinin, "Deep data analysis via physically constrained linear unmixing: universal framework, domain examples, and a community-wide platform," *Advanced Structural and Chemical Imaging*, vol. 4, no. 1, p. 6, Apr 2018. [Online]. Available: https://doi.org/10.1186/s40679-018-0055-8

[2] R. Kannan, G. Ballard, and H. Park, "A high-performance parallel algorithm for nonnegative matrix factorization," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: ACM, February 2016, pp. 9:1–9:11.

[3] ——, "Mpi-faun: An mpi-based framework for alternating-updating nonnegative matrix factorization," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 3, pp. 544–558, March 2018.

[4] G. Ballard, K. Hayashi, and R. Kannan, "Parallel nonnegative cp decomposition of dense tensors," in *25th IEEE International Conference on High Performance Computing(HiPC) 2018*, 2018, p. Accepted.

[5] K. Woodsend and J. Gondzio, "Hybrid mpi/openmp parallel linear support vector machine training," *Journal of Machine Learning Research*, vol. 10, no. Aug, pp. 1937–1953, 2009.

[6] T. Mattson, D. Bader, J. Berry, A. Buluc, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson *et al.*, "Standards for graph algorithm primitives," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. IEEE, 2013, pp. 1–2.

[7] M. D. Zeiler and R. Fergus, "Stochastic pooling for regularization of deep convolutional neural networks," *International Conference on Representation Learning*, pp. 1–9, 2013. [Online]. Available: http://arxiv.org/abs/1301.3557

[8] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," in *NIPS*, vol. 6. Vancouver, BC, 2006, pp. 281–288.

[9] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí, "Parallelizing dense and banded linear algebra libraries using SMPSs," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 18, pp. 2438–2456, 2009.

[10] F. G. Gustavson, J. Wasniewski, J. J. Dongarra, J. R. Herrero, and J. Langou, "Level-3 Cholesky Factorization Routines as Part of Many Cholesky Algorithms," *Transactions on Mathematical Software, in 2nd revision. also, LAWN*, no. 249, p. 23, 2011.

[11] S. Tarca, "Performance optimization of symmetric factorization algorithms," Ph.D. dissertation, Master Thesis, Department of Computer Science, Cornell University, 2010.

[12] K. Gallivan, W. Jalby, U. Meier, and A. H. Sameh, "Impact of Hierarchical Memory Systems On Linear Algebra Algorithm Design," *The International Journal of Supercomputing Applications*, vol. 2, no. 1, pp. 12–48, 1988.

[13] B. Kågström, P. Ling, and C. Van Loan, "High performance gemm-based level 3 blas: Sample routines for double precision real data," pp. 269–281, 1991.

[14] K. Goto and R. Van De Geijn, "High-performance implementation of the level-3 blas," *ACM Trans. Math. Softw.*, vol. 35, no. 1, pp. 4:1–4:14, jul 2008.

[15] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with gpu accelerators," *Proc. of the IEEE IPDPS'10*, pp. 1–8, 2010.

[16] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra, "A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators," in *High Performance Computing for Computational Science – VECPAR 2010: 9th International conference, Berkeley, CA, USA, June 22-25, 2010, Revised Selected Papers*, J. M. L. M. Palma, M. Daydé, O. Marques, and J. C. Lopes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 93–101.

[17] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software*, vol. 16, no. 1, pp. 1–17, mar 1990.

[18] Z. Xianyi, W. Qian, and Z. Yunquan, "Model-driven level 3 blas performance optimization on loongson 3a processor," in *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*. IEEE, 2012, pp. 684–691.

[19] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "Augem: automatically generate high performance dense linear algebra kernels on x86 cpus," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 25.

[20] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi*. Springer, 2014, pp. 167–188.

[21] F. G. Van Zee and R. A. Van De Geijn, "Blis: A framework for rapidly instantiating blas functionality," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 3, p. 14, 2015.

[22] R. K. Malladi, "Using intel® vtune performance analyzer events/ratios & optimizing applications," *http:/software. intel. com*, 2009.

[23] T. M. Smith, R. Van De Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee, "Anatomy of high-performance many-threaded matrix multiplication," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 1049–1059.