A Flexible-blocking Based Approach for Performance Tuning of Matrix Multiplication Routines for Large Matrices with Edge Cases

Md Mosharaf Hossain, Thomas M. Hines,
Sheikh K. Ghafoor, Sheikh Rabiul Islam
Computer Science, Tennessee Technological University, TN
{mhossain44, tmhines42}@students.tntech.edu,
sghafoor@tntech.edu, sislam42@students.tntech.edu

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

Sreenivas R. Sukumar
CRAY Inc
ssukumar@cray.com

***Abstract*—Efficient and scalable matrix operations are being highly demanding in the recent era of Machine Learning, Deep Learning, and Big Data Analytics. The two commonly used matrix-matrix operations in the Basic Linear Algebra Subprograms (BLAS) specification are General Matrix-Matrix multiplication (GEMM) and Symmetric Rank-k update (SYRK). The SYRK routine is a specialization of the GEMM routine, where half of the multiplications are skipped as the resultant matrix is known to be symmetric. Fortunately, several linear algebra libraries implement these BLAS routines quite efficiently. The libraries usually partition the input matrices into blocks and place them in processor caches, thus improving performance by leveraging the caches. However, the contemporary libraries are highly optimized for squarish matrices, but the performance degrades significantly for the matrices with edge case (strictly thin or strictly fat shapes) in the multicore machine. The primary reason is that the current state-of-the-art libraries make fixed block shapes based on a processor architecture, and do not consider the shape of the input matrices. In this paper, we propose a new blocking approach, we name it Flexible-blocking, to mitigate the scalability issues. In contrast to the contemporary libraries, our approach formulates the blocks of the input matrices based on the shapes of the matrices as well as the number of threads used in the implementation. Our proposed technique shows noticeable performance improvement on multicore shared-memory machines for the edge case matrices.**

***Index Terms*—BLAS, Multicore, Flexible-blocking, Big Data, Performance Tuning**

I. Introduction

Linear algebra operations, particularly vector-vector, matrix-vector, and matrix-matrix operations are the core building blocks in several areas of mathematics and applied domains. Throughout the past decades, researchers have continued to focus on efficient and scalable implementations of these operations on modern machines. However, to perform these common linear algebra operations a proposal was outlined by Hanson, Krogh, and Lawson in 1973 [1], where they prescribed how the routines should be used. The routines are commonly known as Linear Algebra Subprograms (BLAS). Later in 1977, A standard version of BLAS routines was developed in Fortran (described in [2]) and continued to be in use in several softwares and mathematical problem domains. At this point, the existing routines can only perform vector-vector operations but in efficient way. Later on in late 80s, an extended set of BLAS routines were proposed by Dongarra, Du Croz, Hammarling, and Hanson in [3], [4] to perform matrix-vector operations in various processor architectures efficiently. The proposal of Level 2 BLAS was made to build portable and efficient libraries across the common machine architectures that known as vector-processing machines [5].

At the time when Level 2 BLAS appeared, the efficiency of frequently used algorithms of numerical linear algebra were obtained by tailored implementations of Level 2 BLAS routines. Keeping vector lengths as long as possible was one of such implementations on the vector-processing machines. Moreover, the performance was increased by reusing the results of a vector register, and reducing memory transfer. This approach of building software is not performant or efficient on computers with a hierarchy of memory components and multicore processing units. Partitioning the input matrices into blocks and then computing on each of the blocks become more efficient and scalable. In addition to that the computation is performed in such a manner that provides full reuse of data block that resides in the local memory [5]. Also, the blocking strategies allow multiple cores to work with the different block at the same time or can work with the same block simultaneously. Later on, Level-3 BLAS routines were developed by Dongarra et. al [5] that leverage the efficient implementation of block partitioning which enables multiple cores to work efficiently.

The existing linear algebra libraries perform matrix operations very efficiently for square or nearly square matrices on multicore machines. However, these libraries show per-

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).

formance bottlenecks as matrices become more strictly fat or strictly thin. The primary reason we observe is that the cache utilization drops quite significantly when multiplying large matrices with edge cases (where an $m \times k$ matrix is strictly thin ($m \gg k$) or strictly fat ($m \ll k$)). In addition, these edge case matrices are common in Big data application sectors such as neutron scattering, molecular dynamics and enterprise systems. Moreover, the data analytics, machine learning, and deep learning tools use the BLAS routines extensively to get optimized and scalable performance when working with Big datasets [6]. Thus, a better blocking strategy can improve performance of the ML tools when working with large matrices with edge cases. However, we address the performance issue by proposing a model that finds appropriate block shapes to the matrices with edge cases on multicore machines. In our work, at first, we propose a thread assignment strategy. Based on the threading pattern, we formulate optimal block shapes which ensures better utilization of caches. We observe a noticeable performance improvement by our Flexible-blocking approach compared to the default selection of blocks by the linear algebra library like BLIS [7], which is architecturally similar to GotoBLAS [8], [9] and OpenBLAS [10], [11], and considered a state-of-the-art.

II. Related Work

In the last couple of years, several techniques have been proposed for tuning the BLAS routines. However, those attempts are mostly based on the squarish shape of input matrices. The Automatically Tuned Linear Algebra Software (ATLAS) library initially showed some way of generating optimized code automatically [12]. They have used a code generator which uses timings to determine the appropriate blocking, where the user may or may not supply many details about the processor (i.e cache size). If the user does not provide any information to the library, the generator produces the proper settings using timings. James et. al. presents SALSA package, which uses statistical data modeling as a tool to automatically choose an appropriate algorithm that ensures high performing kernel [13]. Qing Yi et. al. in [14] presents a new approach, where a general-purpose transformation engine automatically produces highly efficient library routines. They claim that their approach requires only an annotated kernel specification which generates the optimized implementations based on tuning parameters controlled by a search driver. Similar work is described in [15], where they use a transformation scripting language (POET) to get kernel-level optimization.

There are few other works where empirical tuning frameworks are leveraged that provide efficient kernel implementation for several scientific domains [16], [17]. Similar empirical tuning frameworks are used in [18]–[20], where the compilers perform the tuning based on the applications. A recent state-of-the-art linear algebra library is BLIS [7], [21]–[23], which is mostly an extension of GotoBLAS [8], [9]. Although BLIS makes hard-coded blocking parameters based on processor architecture, it provides a way to change the blocking parameters to the users.

In contrast to all the works, we primarily focus on the matrices of strictly fat or strictly thin case. In these particular situations, the contemporary approaches do not show impressive results as they still face poor cache utilization problem. However, we address the issues by formulating appropriate blocking parameters that ensures considerable utilization of processor caches.

III. Block Pack Algorithm

SYRK in BLIS is a specialization of the GEMM algorithm where half of the multiplications are skipped when updating the output matrix. For clarity, the GEMM algorithm is described in this paper. GEMM is implemented by block pack matrix multiplication [7], described in Algorithm 1 and shown in Figure 1. It starts with a matrix A 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

```
1: for $j_c = 0,..,n-1$ in steps of $n_c$ do //5th loop
2:     for $p_c = 0,..,k-1$ in steps of $k_c$ do //4th loop
3:         Pack block of B into $\hat{B}$
4:         for $i_c = 0,..,m-1$ in steps of $m_c$ do //3rd loop
5:             Pack block of A into $\hat{A}$
6:             for $j_r = 0,..,n_c-1$ in steps of $n_r$ do //2nd loop
7:                 for $i_r = 0,..,m_c-1$ in steps of $m_r$ do //1st loop
8:                     Multiply the slice of $\hat{A}$ by the slice of $\hat{B}$ //micro-kernel
9:                 end for
10:            end for
11:        end for
12:    end for
13: end for
```

Loops 5 and 4 around the micro-kernel 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 row of A 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 fill a considerable percentage of L2 and L3 caches, respectively. Slices of $\hat{A}$ (of size $m_r \times k_c$), we also call sliver, and slices of $\hat{B}$ (of size $k_c \times n_r$) are taken in loops 2 and 1; these are sized to fit into L1 cache. The slices are then multiplied together in a vectorized assembly function, known as 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 2 and 1, no data needs to be transferred from DRAM. $\hat{B}$ should remain 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 forth 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 locking as multiple threads try to write to the same section of the output matrix. Unfortunately, this incurs some additional time, and consequently, parallelizing this loop is ignored.

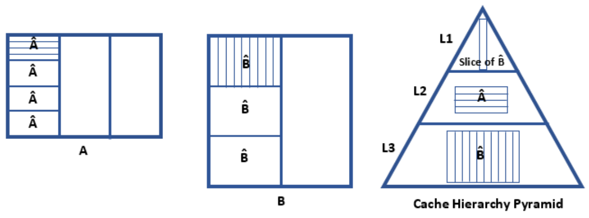


Fig. 1: *The GEMM block matrix partitioning showing the blocking pattern and the storage of blocks in cache*

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, we have used this utility 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, when 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.

In the next section, we discuss our approach to mitigate the poor cache utilization issue. We formulate proper sizes for m_c, k_c, and n_c such that a considerable portion of L2 remains filled-up all the time, and the technique also ensures good utilization of L3 cache compared to default approach in BLAS libraries.

IV. Flexible-blocking Strategy

In this section, we discuss a flexible block partitioning strategy, we call it Flexible-blocking, that improves cache utilization and enhance the performance of multiplication to the matrices of edge cases. At the very beginning, we discuss the theory and mechanism to show how we deduce sizes for $\hat{A}$, $\hat{B}$ and $\hat{C}$. In addition, placing threads in the loops of the five-loop GEMM algorithm is also important to get the best performance. Therefore, we start the discussion of the threading strategy used in our approach.

A. Threading-pattern

We have discussed the five-loop GEMM algorithm in the earlier section where threads can be assigned to the four of the five loops. For explanation, let's assume a threading pattern $\{t5, t3, t2, t1\}$ states that $t5$, $t3$, $t2$, and $t1$ number of threads are assigned in fifth, third, second and first loops around the micro-kernel, respectively. We find that the amount of parallelism is very limited for the first and second loop around the micro-kernel. Hence, we do not parallelize these two loops and reserve the threads to leverage in the upper loops. The resulting pattern now becomes $\{t5, t3, 1, 1\}$, where we have two variables $t5$ and $t3$ to work with. As we set $t5$ to the fifth loop and $t3$ to the third loop, thus the master process spawns $t5 \times t3$ number of threads according to the nested threading concept. Therefore, when we have t number of cores in the processor, we would like to use t number of threads, and deduce possible factor pairs of t. As a concrete example, if we use 20 threads in the GEMM routine call, we can factorize it in $(t5, t3)$ format, and choose any such pair from the set $\{(1, 20), (2, 10), (4, 5), (5, 4), (10, 2), (20, 1)\}$. Thus, our technique finds all possible factor pairs at the beginning. However, if no multi-threading is used, we have only one thread, and it returns the pattern $(1, 1)$.

B. Blocking Strategy

Linear algebra libraries are usually optimized for matrices with squarish shape and put hard-coded block size, we call them to default, to some specific processor architectures. Goto et al. [8] suggest that a considerable percentage of L2 cache need to be full to get the best performance. Best on our experiments, we observe that a percentage of more than 60% and less than 80% cache utilization shows the best result. In our Flexible-blocking approach, we formulate the block shape of $\hat{A}$ (of size $m_c \times k_c$) such that 75% of L2 cache remains filled-up. Below, we discuss the formulation of the blocks for the edge and squarish cases.

1) Case 1: $m \ll k$: In this case, a strictly fat matrix A with dimension $m \times k$ multiplies a strictly thin matrix B with size $k \times n$. In this situation, k is usually several thousands bigger than m, thus we have more parallelism capacity in the k dimension. Now, if a computing node has a large number of processors with private L2 cache, we can derive for the block of $\hat{A}$ such that k_c extends beyond the default value and the dimension $m_c \times k_c$ occupies a good percentage of the L2 cache in each core. Equation 1 shows how m_c is derived from the default m_c(expressed as dm_c), m, i_c, m_r, and number of threads assigned to third loop (i_c).

$$m_c = \lceil max(min(dm_c, \frac{m}{i_c}), m_r) \rceil \tag{1}$$

In the situation, if matrix A would be a large square or thin matrix, Equation 1 assigns dm_c to m_c, because dm_c turns smaller than $\frac{m}{i_c}$ for s moderately bigger value of i_c. On the contrary, when A is a strictly fat matrix, m becomes relatively small, and the Equation 1 assigns $\lceil \frac{m}{i_c} \rceil$ to m_c, which also can not be less than m_r. Now, one important thing is that to achieve optimal performance m_c should be evenly divisible by m_r, this ensures every thread to work with same size sliver. By Equation 2 we show the function call and the respective method is shown in Algorithm 2.

$$m_c = ceil\_to\_multiple(m_c, m_r) \tag{2}$$

The function shown in the equation 2 makes ceiling of m_c to the multiple of m_r. Algorithm 2 shows this simple concept of ceiling a number to the multiple of another number.

Algorithm 2 ceil_to_multiple(x, y)

```
1: $r \leftarrow x\%y$
2: if $r > 0$ then
3:     return $x + (y - r)$
4: else
5:     return $x$
6: end if
```

Now, let's assume that the variable $parcent\_L2$ means the percentage of memory we will allow to reside in L2. And, $size\_L2$ is the size of L2 cache in bytes for a particular processor. We can now derive k_c by the following formulations.

$$p * m_c * k_c + p * k_c * n_r = percent\_L2 * size\_L2 \tag{3}$$

In Equation 3, p is the byte size of the precision. The left term of the equation is the data that resides in L2, the terms $p * m_c * k_c$ and $p * k_c * n_r$ are the sizes of $\hat{A}$ and sliver of $\hat{B}$, respectively. The sliver of $\hat{B}$ is actually brought to L1 though bringing it into L2 first.

Therefore, we obtain k_c by the below equation.

$$k_c = \left\lceil \frac{percent\_L2 * size\_L2}{p * (m_c + n_r)} \right\rceil \tag{4}$$

Now, in case of k_c turns to greater than k, we practically consider Equation 5 instead of equation 4.

$$k_c = min(\left\lceil \frac{percent\_L2 * size\_L2}{p * (m_c + n_r)} \right\rceil, k) \tag{5}$$

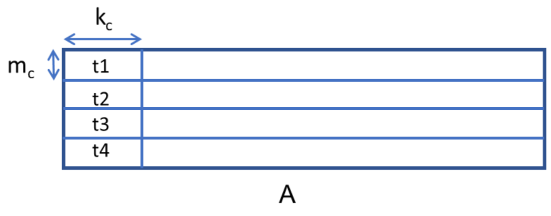
With the formulation of m_c and k_c, we leverage the L2 cache up to a considerable percentage (say 75%). In case of a strictly fat matrix, m is pretty low, whereas k is very large. When loop 3 is parallelized with large number of threads, and m is not big enough to assign default m_c to multiple threads, the linear algebra library assigns $\frac{m}{i_c}$ amount to the threads. However, the libraries do not touch the k_c dimension. This is why the utilization of L2 drops in this situation. In contrast, our formulation of m_c and k_c ensures good utilization of L2 cache. Figure 2 depicts the difference of our approach from the default library situation. In contrast to the libraries like BLIS, our approach also improves L3 cache utilization as we increase k_c, which apparently increases the size of $\hat{B}$.

Finally, the value of n_c can be derived with the number of threads assigned to the fifth loop (j_c) as follows. In the Equation 6, dn_c is assumed as the default value of n_c used in BLIS like library.

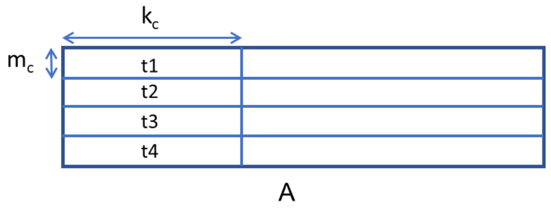
$$n_c = \left\lceil max(min(\frac{m}{j_c}, dn_c), n_r) \right\rceil \tag{6}$$

As discussed before, n_c is also should be multiple of n_r, hence we get Equation 7 shows below.

$$n_c = ceil\_to\_multiple(n_c, n_r) \tag{7}$$



(a) Default-blocking for $m \ll k$.



(b) Flexible-blocking for $m \ll k$.

Fig. 2: Figure (a) shows that BLIS like library breaks down m into m_c, but do not touch k_c. This gives poor utilization of L2 and L3 cache while performing multiplication of strictly fat and thin matrices. Figure (b) shows how our approach fixes the issue by expanding k_c and making it a bigger chunk.

2) Case 2: $m \approx k$: BLIS like libraries are optimized when two matrices have nearly the same dimension. When we multiply two considerably large square matrices, from the Equation 1, we can observe that our proposed idea assigns m_c to its default value, because default m_c more likely be smaller than $\frac{m}{i_c}$ in this situation. If we put this m_c value to the Equation 5, we observe pretty close to the default value, as we constraint a certain percentage of L2 to be full. Similarly, we can obtain n_c by Equation 6, which results in default or nearly default value when square matrices are large.

3) Case 3: $m \gg k$: In this particular case, k is very small, say only 100, whereas m can be very big, say some millions. Using default blocking of $\hat{A}$, utilization of L2 becomes very poor. Because, k_c becomes limited by the smaller k dimension, and BLAS libraries can not extend the default m_c while performing multi-threading. However, we propose the idea to increase m_c such that utilization of L2 reaches to a considerable percentage, say 75%. Equation 8 shows a simple formulation for k_c.

$$k_c = min(dk_c, k) \tag{8}$$

We represent dk_c as the default k_c, which is set to 256 in Sandy Bride architecture by BLIS. By the above equation, if k is smaller than dk_c, k_c is assigned with k. After setting k_c, we can now formulate for the value of m_c. Likewise, the previous equation 3, we can get as follows.

$$p * k_c(m_c + n_r) = percent\_L2 * size\_L2$$

$$m_c = \left\lceil \frac{percent\_L2 * size\_L2}{p * k_c} - n_r \right\rceil \tag{9}$$

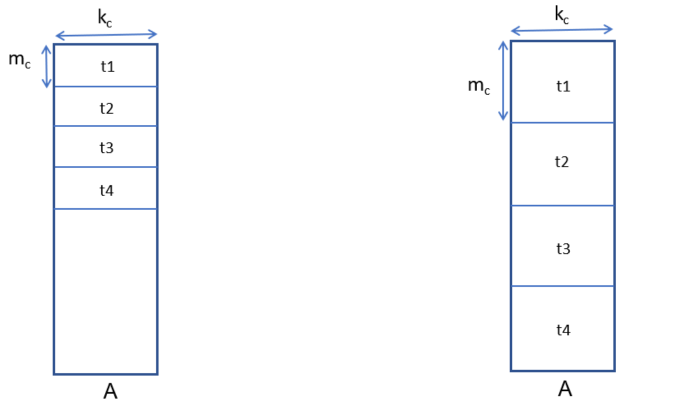
The Equation 9 generates a proper size of m_c such that

L2 is utilized to a considerable percentage. Unfortunately, m_c might be larger than $\lceil \frac{m}{i_c} \rceil$ whenever a large number of threads (i_c) are used in the third loop around the micro-kernel. To address the issue, we can limit m_c by Equation 10.

$$m_c = \left\lceil min(\frac{percent\_L2 * size\_L2}{p * k_c} - n_r, \frac{m}{i_c}) \right\rceil \tag{10}$$

Again, to maintain optimal performance we perform the ceil_to_multiple function such that m_c becomes evenly divisible by m_r, as we have shown in Equation 2. Like before, to get the value of n_c we apply same operation as shown in Equation 6 and 7.

Figure 3 shows how we address poor L2 cache utilization because of the strictly thin shape of matrix A. We find considerably large m_c toward m dimension such that a substantial percentage of L2 gets occupied.



(a) Default-blocking for $m \gg k$.

(b) Flexible-blocking for $m \gg k$.

Fig. 3: Figure (a) shows that BLIS like library applies default $m_c \times k_c$ even though k is very small and m is very big. This results poor utilization of L2 cache while performing multiplication of strictly thin with strictly fat matrices. In contrast, Figure (b) shows how our work addresses the issue by making flexible partitioning, where we expand m_c such that we can maintain a considerable percentage of L2 utilization.

C. Cache Utilization

After the formulation of m_c, k_c, and n_c we can now estimate the percentage of cache utilization for a particular thread pattern. The size of $\hat{A}$ and $\hat{B}$ calculated in Bytes, are shown below. In the case when k_c smaller than k, then k_c becomes k.

$$\hat{A}_{size} = p * m_c * min(k_c, k)$$
$$\hat{B}_{size} = p * min(k_c, k) * n_c$$

Where, p is the size in bytes for double precision we discussed before. Double and single precisions are commonly used for matrix operations. Now, we will calculate the size of a sliver of $\hat{A}$ (of dimension $m_r \times k_c$), and a silver of $\hat{B}$ (of dimension $k_c \times n_r$).

$$sliver\hat{A}_{size} = p * m_r * min(k_c, k)$$
$$sliver\hat{B}_{size} = p * min(k_c, k) * n_r$$

As discussed earlier, $\hat{A}$ will occupy a considerable portion of L2 cache, whereas $\hat{B}$ will occupy a good portion of L3 cache. $\hat{A}$ moves to L2 through L3 from the main memory. Therefore, L3 stores $\hat{A}$ and $\hat{B}$ both in its cache. Again, a sliver of $\hat{B}$ moves to L1 cache through L2, as a result L2 cache stores $\hat{A}$ and a sliver of $\hat{B}$. Below formulations estimates the memory occupied in L2 and L3 caches.

$$L2_{data} = \hat{A}_{size} + sliver\hat{B}_{size}$$
$$L3_{data} = \hat{B}_{size} * j_c + \hat{A}_{size} * (j_c * i_c)$$

As the nested parallelism is used in the five-loop GEMM algorithm, j_c threads in the fifth loop and i_c threads in the third loop gives total $j_c * i_c$ threads spawned. In this situation, each thread in the fifth loop becomes master of the i_c number of threads in the third loop.

Finally, the percentage utilization of L2 (U_{L2}) and utilization of L3 (U_{L3}) caches can be calculated simply as follows.

$$U_{L2} = \frac{L2_{data} * 100}{L2_{size}}$$
$$U_{L3} = \frac{L3_{data} * 100}{L3_{size}}$$

Where, $L2_{size}$ is the size in Bytes of L2 cache per core. And, $L3_{size}$ is the size in Bytes of L3 cache, which is most likely shared among the cores.

D. Finding Appropriate Parameters

At this point, we have already shown how we can generate factor pairs for any number of threads in Subsection IV-A. After that, we formulated the optimal value of m_c, k_c, and n_c for any given threading pattern in Subsection IV-B. Having done the threading and blocking strategy, we have calculated the estimated cache utilization in Subsection IV-C. Now, based on the L3 utilization, we can approximate the value of m_c, k_c, and n_c for all possible threading patterns of a given thread. However, we need to make several constraints while selecting the parameters. We do not allow the L3 utilization to be more than a cut-off percentage. Because, if L3 utilization exceeds a certain percentage, say 75%, then cache miss rate might starts to increase. In addition, we put equal or more threads on third loop than fifth loop for the fat case, which helps to choose bigger k. On the contrary, we put equal or more threads on the fifth loop than third for the thin case, which helps to choose a blocking pattern that ensures good L3 utilization. However, the algorithm basically turns a max value searching problem with constraints, which find max utilization of L3 within a given range.

V. Experimental Design

For all the experiments, we use BLIS 0.2.2 [21] in a shared memory multicore environment. Our results and analysis are based on machines from two separate clusters. We use only one node from each of the clusters, as we only focus on the shared memory system. In the first cluster setup, each node has two Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80 GHz processors, where each processor has 10 physical cores. The code name for this processor type is *Ivy Bridge*. Each core in the processor as private L1 and L2 caches. L3 cache is shared among the cores. Size of the L1, L2, and L3 caches are 32 KB, 256 KB, and 25600 KB respectively. The node has 128 GB RAM and the two processors are connected by Cache Coherent Non-Uniform Memory Access (cc-NUMA) memory architecture.

The second machine setup we have used is a 40 nodes cluster. we have requested only one node every time we have run the experiment. Each node has two Intel Xeon E5-2680v4 processors. The code name for this processor type is *Broadwell*. Each processor has 14 physical cores and 35 MB shared L3 cache. Each core has 32 KB L1 and 256 KB L2 caches. Like the first machine setup, each node has 128 GB RAM and the two processors are connected by the same cc-NUMA memory architecture.

VI. Experimental Results and Analysis

In the subsequent discussion, we have used performance in GFLOPS (Billion Floating Point Operation Per Second) as calculated by Equation 11 [24], [25] below, where t is computation time in seconds.

$$\frac{km(m+1)}{t} * \frac{1}{10^9} \tag{11}$$

A. Results on Intel Ivy Bridge

1) Case 1: $m \ll k$: Flexible-blocking performs very well for the strictly fat matrices compared to the default-blocking strategies used in BLAS libraries like BLIS. Figure 4 shows a comparison figure with four graphs where. We show performance along with the Y-axis, and the number of threads is shown in X-axis. The number of threads we used for this section are $2, 3, 6, 9, 12, 15, 18, 19, 20$. We have already described the mechanism of our Flexible-blocking for this fat case in sub section IV-B1.

In Figure 4a, the dimension of A matrix is $100 \times 83.7M$, which is an unquestionably very big fat matrix. We find that the performance is pretty close up to 3 threads for both the approaches. As we increase more threads, we observe literally high jump of GFLOPS for Flexible-blocking. At thread 9, performance improves 94% than the default BLIS. The performance improves by nearly 100% as we increase the number of threads beyond 12, except at thread 15. Fortunately at thread 15, BLIS generates threading pattern $\{5, 3, 1, 1\}$, which shows good utilization of the cache memories. However, after a certain number of threads, 18 in this case, Flexible-blocking slightly decreases, but the overall performance improvement is pretty much stable compared to Default-blocking.

In Figure 4b, we increase m and decrease k, thus making very fat matrix with dimension $128 \times 51.2M$, which is comparatively less fat than 4a case. We observe more than 50% performance gain than BLIS implementation as we use greater number of threads, usually some more than 9 threads. A similar situation happens at 15 threads in the default case as we described earlier.

We continue increasing m and decreasing k to observe the performance of our approach. Figure 4c shows pretty similar trend like the earlier two figures, but in this case, Flexible-blocking improves around 10% compared to the BLIS library while using more than 6 threads.

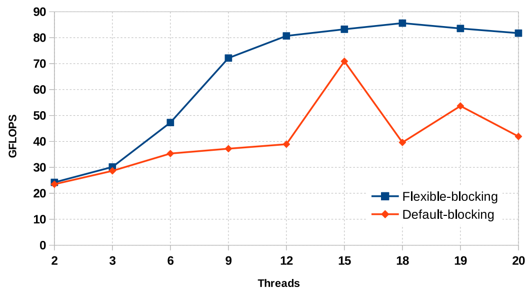
In Figure 4d, a comparatively less fat matrix is used than the previous three cases. In this particular situation, both approaches show very good performance and pretty close GFLOPS as we increase number of threads. As m is quite big, which means columns of the B matrix (transpose of A matrix) is big. This gives a comparatively bigger shape of $\hat{B}$, which occupies a considerably good portion of the L3 cache. Thus, our approach imitates performance very similar to the default approach.

Now, to show the threading and blocking pattern generated by our proposed approach, we present data in Table I. The table shows data for the fat matrix A having dimension $100 \times 83.7M$. The methodology of generating threading pattern, blocking pattern, and cache utilization are described in Subsections IV-A, IV-B, and IV-C, respectively. We observe that for all the threading situation, j_c remains 1, which means only master thread works at the fifth loop around the micro-kernel. However, all the threads are assigned to the third loop, hence i_c becomes t. Using this strategy, $\hat{A}$ and $\hat{B}$ practically follow the shape structure of the original matrices. For an example of thread 20, the dimension of block $\hat{A}$ becomes 8×2048, which is considerably fat. And, the dimension of $\hat{B}$ becomes 2048×100, which is very thin. Thus, our proposed approach finds suitable blocking based on L3 cache utilization while keeping L2 utilization to a considerable percentage.

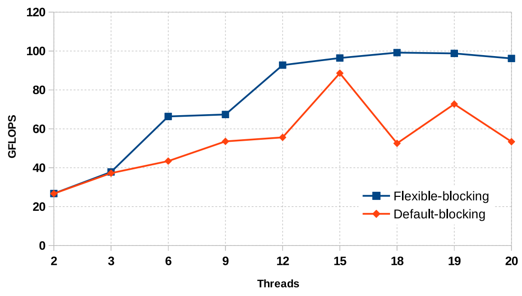
TABLE I: Threading pattern, blocking pattern and cache utilization by Flexible-blocking for the strictly fat matrix A with dimension $100 \times 83.7M$. t denotes number of threads used.

t	Threading-pattern		Blocking-pattern			Cache utilization (%)	
	j_c	i_c	m_c	k_c	n_c	$L3$	$L2$
2	1	2	56	410	100	2.65	75
3	1	3	40	559	100	3.75	75
6	1	6	24	878	100	6.54	75
9	1	9	16	1229	100	9.15	75
12	1	12	16	1229	100	10.95	75
15	1	15	8	2048	100	13.75	75
18	1	18	8	2048	100	15.25	75
19	1	19	8	2048	100	15.75	75
20	1	20	8	2048	100	16.25	75

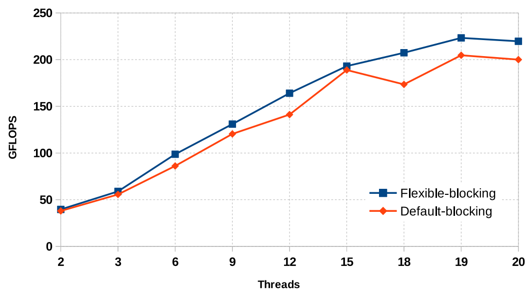
2) Case 2: $m \approx k$: We discussed in Section VI-A1 that as we deviate the A matrix from strictly fat to squarish, the performance issue in BLIS vanishes. Now, we show the situation for a perfect square matrix with dimension $20K \times 20K$ in Figure 5a. In both approaches, performance



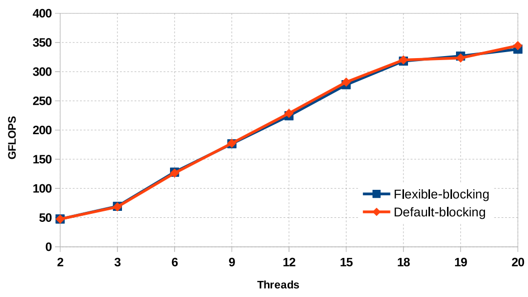
(a) $A = 100 \times 83.7M$



(b) $A = 128 \times 51.2M$



(c) $A = 500 \times 3.4M$



(d) $A = 3000 \times 94K$

Fig. 4: DSYRK experimental results are shown in GFLOPS in four graphs for the $m \ll k$ case. The dimension towards m increases and dimension towards k decreases in the subsequent graphs. Number of threads are used ranging from 2 to 20.

increases linearly starting from 3 threads and continues until 18 threads. After that performance slows down a bit. However, in this particular case, both Flexible-blocking and Default-blocking picks similar blocking of $\hat{A}$, $\hat{B}$, and $\hat{C}$ as m and k are large enough to partition into default block sizes.

Table II shows selected threading pattern for the square matrix with dimension $20K \times 20K$ by our proposed technique. As the dimension of m, n, and k are all equal, Flexible-blocking selects similar blocking pattern like BLIS. The methodology for selecting threading and blocking strategy for this particular case is described in Subsection IV-B2.

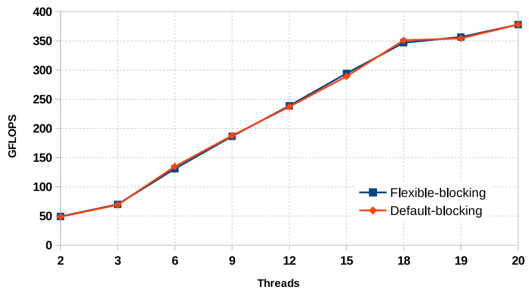
TABLE II: Threading pattern, blocking pattern and cache utilization by Flexible-blocking for the square matrix A with dimension $20K \times 20K$. t denotes number of threads used.

t	Threading-pattern		Blocking-pattern			Cache utilization (%)	
	j_c	i_c	m_c	k_c	n_c	$L3$	$L2$
2	2	1	96	246	4096	62.94	75
3	1	3	96	246	4096	32.91	75
6	2	3	96	246	4096	65.82	75
9	1	9	96	246	4096	37.24	75
12	1	12	96	246	4096	39.40	75
15	1	15	96	246	4096	41.56	75
18	1	18	96	246	4096	43.72	75
19	1	19	96	246	4096	44.44	75
20	1	20	96	246	4096	45.16	75

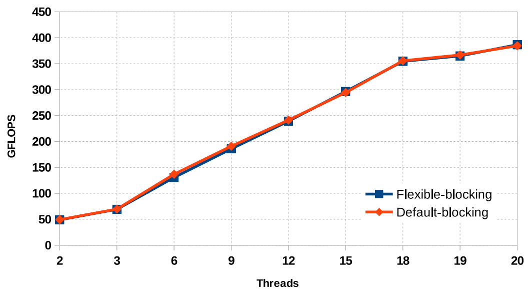
3) Case 3: $m \gg k$: In this situation, a big strictly thin matrix A gets multiplied with its transpose, which is a strictly fat matrix. Last three plots of Figure 5 depicts this skinny matrix case. We gradually increase m and decrease k to observe the performance drops in BLIS library. Fortunately, our approach also addresses the performance issue for this thin matrix case. The flexible-blocking for the skinny case is described in Subsection IV-B3.

In Figure 5b, we start matrix dimension from $50K \times 1000$, which is a considerably tall matrix. In this particular situation, k is much bigger than the default k_c. Additionally, the transpose matrix (A^T) has large n, that can be easily partitioned with n_c steps, and distributed among available threads. Thus, in the Flexible-blocking approach, the blocking turns to be very similar to the default approach. As a result, we observe the very similar trend of performance curve for both the approaches.

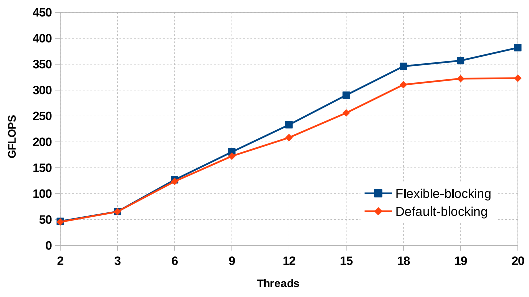
Figure 5c presents comparison graphs for a considerably large thin matrix. The matrix dimension is $120K \times 100$. Here, k is less than the default k_c. Hence, by the default approach, the utilization of L2 drops automatically as m_c remains constant regardless of the skinny shape of the A matrix. Our approach explained in Subsection IV-B3, addresses the poor cache utilization issue, which expands m_c such that a good portion of L2 gets filled-up by $\hat{A}$. In Figure 5c, up to 9 threads, both approaches show a similar trend. However, as we increase thread count, a significant gap appears between the two approaches. Our approach improves more than 10% compared to BLIS as much as 12 threads are used. Although a slight drop of GFLOPS happens after 18 threads for both the methods. The maximum difference is found at 20 threads, where the Flexible-blocking leads by 18%.



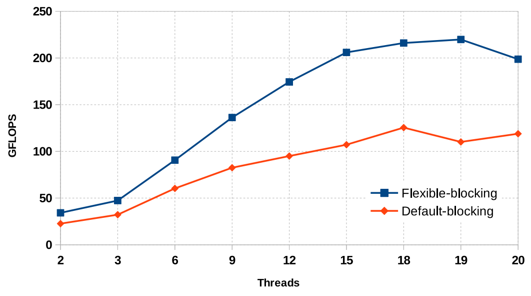
(a) $A = 20K \times 20K$



(b) $A = 50K \times 1000$



(c) $A = 120K \times 100$



(d) $A = 120K \times 25$

Fig. 5: DSYRK experimental results are shown in GFLOPS in four graphs. First graph is for squarish case and next three graphs represent case $m \gg k$ case. Number of threads are used ranging from 2 to 20.

At this point, we decrease k while keeping m constant. Thus, matrix A now becomes a significantly large skinny matrix. Similar kind of matrices are used in many machine learning applications, where features are reduced to a smaller set to reduce over-fitting. However, in Figure 5d, we present this situation of the skinny case. We find very promising results for Flexible-blocking approach by this figure. Even in 2 threads, our approach leads BLIS library by 51%. As we increase the number of threads, performance significantly improves by our approach compared to BLIS. We find a noticeable performance gain at 19 threads, where flexible-blocking leads BLIS by nearly 100%. However, at 20 threads, performance slows down a little for our approach, but it still leads BLIS approach by a bigger margin, nearly 67%.

Table III provides threading and blocking patterns for the thin matrix having dimension $120K \times 25$. As the matrix has very large in m dimension, the Flexible-blocking selects considerably large m_c such that a good portion of L2 gets filled-up. Whereas, the BLIS approach does not expand m_c, although it suffers poor L2 utilization.

B. Results on Intel Broadwell

At this point, we discuss performance improvement of $DSYRK$ by the Flexible-blocking on Intel Broadwell micro-architecture. In this architecture, BLIS picks dimensions 72×256 and 256×4080 for the blocking of $\hat{A}$ and $\hat{B}$ respectively. We show four graphs in Figure 6 that present a performance comparison of Flexible-blocking against Default-bocking of

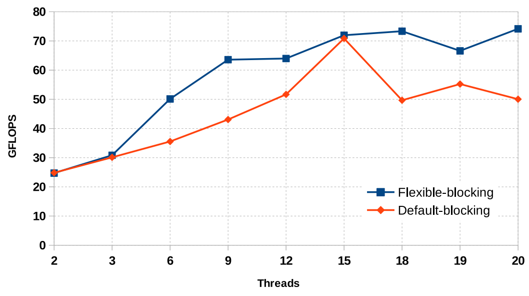
TABLE III: Threading pattern, blocking pattern and cache utilization by Flexible-blocking for the strictly thin matrix A with dimension $120K \times 25$. t denotes number of threads used.

t	Threading-pattern		Blocking-pattern			Cache utilization (%)	
	j_c	i_c	m_c	k_c	n_c	$L3$	$L2$
2	2	1	984	25	4096	7.75	75
3	3	1	984	25	4096	11.63	75
6	6	1	984	25	4096	23.25	75
9	9	1	984	25	4096	34.88	75
12	12	1	984	25	4096	46.51	75
15	15	1	984	25	4096	58.14	75
18	18	1	984	25	4096	69.76	75
19	19	1	984	25	4096	73.64	75
20	20	1	984	25	4096	46.26	75

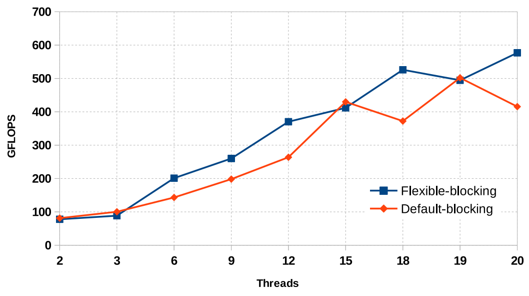
BLIS. First two depicts strictly fat cases, the third and forth figures show square and strictly thin cases, respectively.

In Figure 6a, the dimension of matrix A is $128 \times 51.2M$. At a fewer number of threads, both approaches behave similarly. At six threads, Flexible-blocking leads BLIS by 41% and the performance continues to increase for a bigger number of threads. Fortunately, at 15 threads, Default-blocking shows quite good GFLOPS, which is very similar to Flexible-blocking approach. In this situation, Default-blocking generates similar threading pattern like our approach, Which results in good utilization of L3 cache. However, at 20 threads, our approach leads BLIS by 48%.

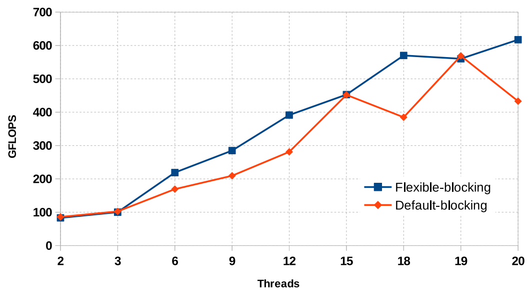
Figure 6b also represents a fat case, but we increase m by a bigger number and decrease k such that matrix A has considerably large m to split among the threads. The figure



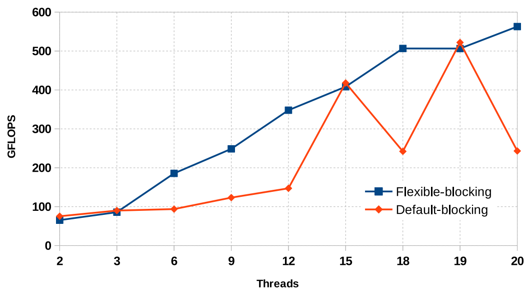
(a) $A = 128 \times 51.2M$



(b) $A = 3K \times 93K$



(c) $A = 20K \times 20K$



(d) $A = 120K \times 100$

Fig. 6: Experimental results on Intel Xeon E5-2680v4 CPUs for DSYRK routine. First two graphs represent $m \ll k$ case. Third graph shows the square case with dimension $20K \times 20K$. Final graph represents case $m \gg k$ case. Number of threads are used ranging from 2 to 20.

shows that the performance of our approach leads BLIS for almost any threads. In this situation, Flexible-blocking picks little bigger k_c. Thus, the dimension of $\hat{A}$ becomes 72×308, which makes a considerably good L2 utilization, 75% in this case.

A square case with dimension $20K \times 20K$ is presented in Figure 6c. Although A is a square matrix, our approach performs pretty well compared to the default case. Like the $3K \times 93K$ matrix situation, in this case, k_c is bigger than the default k_c. In Broadwell micro-architecture, BLIS selects m_c and k_c such that around 50% of L2 gets filled-up by $\hat{A}$. Whereas, in our case, we select the shape of $\hat{A}$ such that L2 utilization becomes 75%. However, in contrast to BLIS, our approach can maintain similar utilization of L2 cache regardless of strictly fat or strictly this shape of matrices.

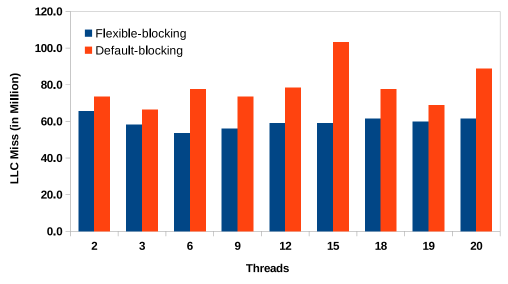
Finally, we present a strictly thin case of A matrix in Figure 6d, which has dimension $120K \times 100$. At the lower number of threads, performance is pretty close in both approaches. As we increase the number of threads, the default approach falls behind our approach. At 12 and 20 threads, we observe more than 100% improvement of GFLOPS by the Flexible-blocking. Although at 15 and 19 threads, Default-blocking shows similar performance, the reason is similar as we discussed in Figure 6a. However, the reason of showing significantly good performance for this thin case is that we formulate much bigger m_c (978) than the default m_c (72) to ensure considerably large L2 utilization.

C. Analysis Using Intel VTune

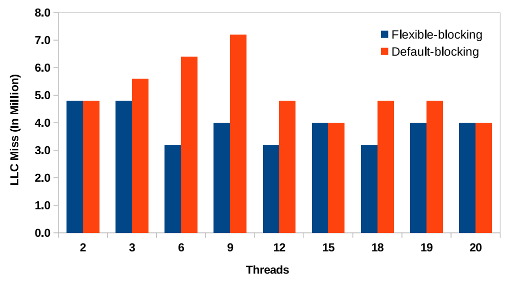
In this section, we discuss performance improvement of Flexible-blocking compared to default approach in respect to Last-level Cache (LLC) miss, which we measured using Intel VTune Amplifier 2018. We show two representative edge cases on the Intel Ivy Bridge machine.

Figure 7a shows a strictly fat case, where the matrix dimension is $100 \times 83.7M$. We notice that Flexible-blocking incurs less cache misses compared to BLIS for any number of threads. At 2 threads, our approach shows 12% less L3 cache miss than default approach. Again, at the higher number of threads, we find that our approach incurs fewer cache misses than BLIS by a good margin.

In Figure 7b, we show a strictly thin case with the dimension $120K \times 25$. We observe that although some situation both the approaches show similar cache misses. However, the majority of threading situations, Flexible-blocking incurs fewer cache misses, and sometimes cache misses reduced by a bigger margin. Like at 6 threads, cache misses reduced to 100% and at 9 threads, cache misses reduced to 80% by the Flexible-blocking. Thus, our approach reduces LLC misses and improves performance as we bring comparatively bigger blocks than default BLIS, which fill a considerably large portion of caches. These blocks remain in the caches until needed by the computation, thus cache hit rate goes up by our Flexible-blocking approach that boost-ups the performance.



(a) LLC Miss for $100 \times 83.7M$.



(b) LLC Miss for $120K \times 25$.

Fig. 7: Last-level cache (LLC) Miss count, measured by Intel VTune profiler, presented for strictly fat and strictly thin matrices.

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. 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.

VII. Conclusion

In this work, we have proposed a new blocking approach for poor cache utilization issues in the contemporary linear algebra libraries. For L2 utilization, we determine all possible threading patterns, then formalizes m_c, k_c, and n_c dimensions for the blocks. Finally, the algorithm chooses a suitable threading pattern, as well as a suitable blocking pattern based on the L3 cache utilization.

Our proposed approach shows significant performance improvement for strictly fat and strictly thin matrices in both Intel Ivy Bridge and Broadwell processors without compromising GFLOPS like BLIS for squarish cases. Finally, we have shown the VTune analysis on a representative strictly fat and a strictly thin case and observe that Flexible-blocking demonstrates considerably less cache misses compared to BLIS in almost all of the threading situations. Thus, the proposed approach is expected to optimize the existing linear algebra libraries that can lead an accelerated Big Data analytics for any shape of data.

References

[1] R. J. Hanson, F. T. Krogh, and C. Lawson, "A proposal for standard linear algebra subprograms," 1973.

[2] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[3] S. Hammarling, J. Dongarra, J. Du Croz, and R. Hanson, "An extended set of fortran basic linear algebra subprograms," *ACM Transactions on Mathematical Software*, vol. 14, no. 1, pp. 1–32, 1988.

[4] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of fortran basic linear algebra subprograms: Model implementation and test programs," Argonne National Lab., IL (USA), Tech. Rep., 1987.

[5] 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 (TOMS)*, vol. 16, no. 1, pp. 1–17, 1990.

[6] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," in *Cloud Computing and Big Data (CCBD), 2016 7th International Conference on*. IEEE, 2016, pp. 99–104.

[7] 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.

[8] K. Goto and R. A. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, p. 12, 2008.

[9] K. Goto and R. Van De Geijn, "High-performance implementation of the level-3 blas," *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, no. 1, p. 4, 2008.

[10] 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.

[11] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "Augem: automatically generate high performance dense linear algebra kernels on x86 cpus," in *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*. IEEE, 2013, pp. 1–12.

[12] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Supercomputing, 1998. SC98. IEEE/ACM Conference on*. IEEE, 1998, pp. 38–38.

[13] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick, "Self-adapting linear algebra algorithms and software," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 293–312, 2005.

[14] Q. Yi and R. C. Whaley, "Automated transformation for performance-critical kernels," in *Proceedings of the 2007 Symposium on Library-Centric Software Design*. ACM, 2007, pp. 109–119.

[15] Q. Yi and A. Qasem, "Exploring the optimization space of dense linear algebra kernels," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2008, pp. 343–355.

[16] M. Frigo and S. G. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[17] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko *et al.*, "Spiral: Code generation for dsp transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[18] G. Fursin, A. Cohen, M. OBoyle, and O. Temam, "A practical method for quickly evaluating program optimizations," in *International conference on high-performance embedded architectures and compilers*. Springer, 2005, pp. 29–46.

[19] A. Qasem, K. Kennedy, and J. Mellor-Crummey, "Automatic tuning of whole applications using direct search and a performance-based transformation system," *The Journal of Supercomputing*, vol. 36, no. 2, pp. 183–196, 2006.

[20] R. C. Whaley and D. B. Whalley, "Tuning high performance kernels through empirical compilation." in *ICPP*, vol. 5, 2005, pp. 89–98.

[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] F. G. V. Zee, T. M. Smith, B. Marker, T. M. Low, R. A. Geijn, F. D. Igual, M. Smelyanskiy, X. Zhang, M. Kistler, V. Austel *et al.*, "The blis framework: Experiments in portability," *ACM Transactions on Mathematical Software (TOMS)*, vol. 42, no. 2, p. 12, 2016.

[23] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, "Analytical modeling is enough for high-performance blis," *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 2, p. 12, 2016.

[24] S. Blackford and J. Dongarra, "Lapack working note 41 installation guide for lapack1."

[25] R. A. Van De Geijn and E. S. Quintana-Ortí, "The science of programming matrix computations," 2008.