GEMM3: Constant-Workspace High-Performance Multiplication of Three Matrices for Matrix Chaining

Honors Thesis in Partial Fulfillment of the Turing Scholars Degree Program

Krzysztof A. Drewniak

March 27, 2018

Abstract

The generalized matrix chain problem considers how to efficiently compute a series of matrix multiplications where some of the operands may be inverted, transposed, or have mathematical properties that can be used to increase performance. Solutions to this problem often involve series of calls to high-performance implementations of matrix multiplication (GEMM). We introduce a primitive, GEMM3, which multiplies three general matrices, to aid in solving this problem. By taking advantage of the structure of modern algorithms for GEMM, we derive an algorithm for GEMM3 which can multiply three matrices using only a constant amount of additional memory. Current approaches to such multiplications require a temporary buffer than can be arbitrarily large. We present experimental results which show that our algorithm retains performance comparable to that of current methods.

1 Introduction

The matrix chain problem asks how to efficiently parenthesize a matrix product $A_1 A_2 \cdots A_n$. $O(n \log n)$ algorithms for solving this problem are known [8], as are simpler $O(n^3)$ dynamic-

programming solutions [3]. These algorithms all assume that the only operation available is the multiplication of two matrices.

In practice, a more general version of this problem appears, where matrices can be transposed and/or inverted, or have properties such as being upper triangular [3]. For example, an algorithm to compute the ensemble Kalman filter [11] (which computes parameters for physical systems from noisy data) requires the computation of the expression $X_i^b S_i (Y_i^b)^T R_i^{-1}$, which features one transposition and one inverse. Other examples of the generalized matrix chain problem include the expression $L^{-1}AL^{-H}$ (where L is lower triangular and A is symmetric), which is knows as the two-sided triangular solve [10] and $\tau_u \tau_v v v^T A u u^T$ (where the τ_i are scalars and v and u are vectors), which appears in an algorithm for reducing a matrix to tridiagonal form [4].

The BLAS [5] and LAPACK [1] libraries provide many high-performance functions that can assist in the computation of such matrix chains, such as specialized functions for the multiplication of triangular matrices (TRMM). However, manually determining how to use these functions and coding in term of them requires expert knowledge. Existing high-level systems, such at Matlab and Julia, which are commonly used by non-experts, generally compute generalized matrix chains in a naive way, such as proceeding from left to right, which is often ineffecient. Systems such as Linnea [2] have emerged recently that attack the generalized matrix chain problem and automatically generate a high-performance program for a given expression in term of high-performance operations such as matrix-matrix multiply.

Returning to the example matrix chains above, we can observe that they often contain the multiplication of three matrices as a subproblem, often after some preprocessing such as an inversion or an outer product. The general form of such a problem is $G \coloneqq \alpha DEF + \beta G$ (with some of the operands optionally transposed), which we will summarize as $G \mathrel{+}= DEF$ and denote GEMM3, by analogy with the notation for general matrix-matrix multiply (GEMM) in the BLAS library. The letters D, E, F, and G were chosen to avoid confusion with discussions of GEMM, which has the form $C \mathrel{+}= AB$. The only approach available for this

subproblem with current methods is to parenthesize the expression DEF and perform a pair of matrix multiplications, storing a result in a temporary matrix T. This method has two drawbacks. The first is that T is often rather large (and could be larger than some or all the input matrices) and thus requires significant amounts of storage space. In addition, if T is sufficiently large, creating and using it incurs a non-negligible overhead due to the large number of slow main-memory operations required.

To combat these issues, we have developed an algorithm for GEMM3 which does not require an entire intermediate product to be computed and stored at one time. Our algorithm takes advantage of the blocking performed by modern GEMM implementations to only store a cache-sized block of EF at any given time. By doing so, we perform the multiplication in $O(1)$ additional space and maintain performance comparable with a pair of GEMM calls. Our algorithm can therefore be used as a primitive in generalized matrix chain solvers (or hand-written matrix code) to reduce memory usage, decrease the complexity of the solution, and potentially provide a slight performance increase.

2 Background

2.1 High-performance GEMM

Before discussing our approach to GEMM3, it is important to review the implementation of high-performance GEMM. High-performance GEMM algorithms operate by repeatedly reducing the multiplication to a series of multiplications of regions of the inputs. These regions are sized to allow an operand to one of these subproblems to utilize a level of the CPU's cache effectively and prevent it from being evicted during further subdivision. Multiple levels of such subdivision are required to effectively utilize all levels of the cache. These steps are necessary because of the memory hierarchy of modern CPUs, where there are multiple (currently three) levels of cache, which increase in size but decrease in speed, between registers and main memory. Subdividing the matrices in a matrix multiplication along some

dimension and then computing each resulting subproblem is known as *partitioning* along that dimension.

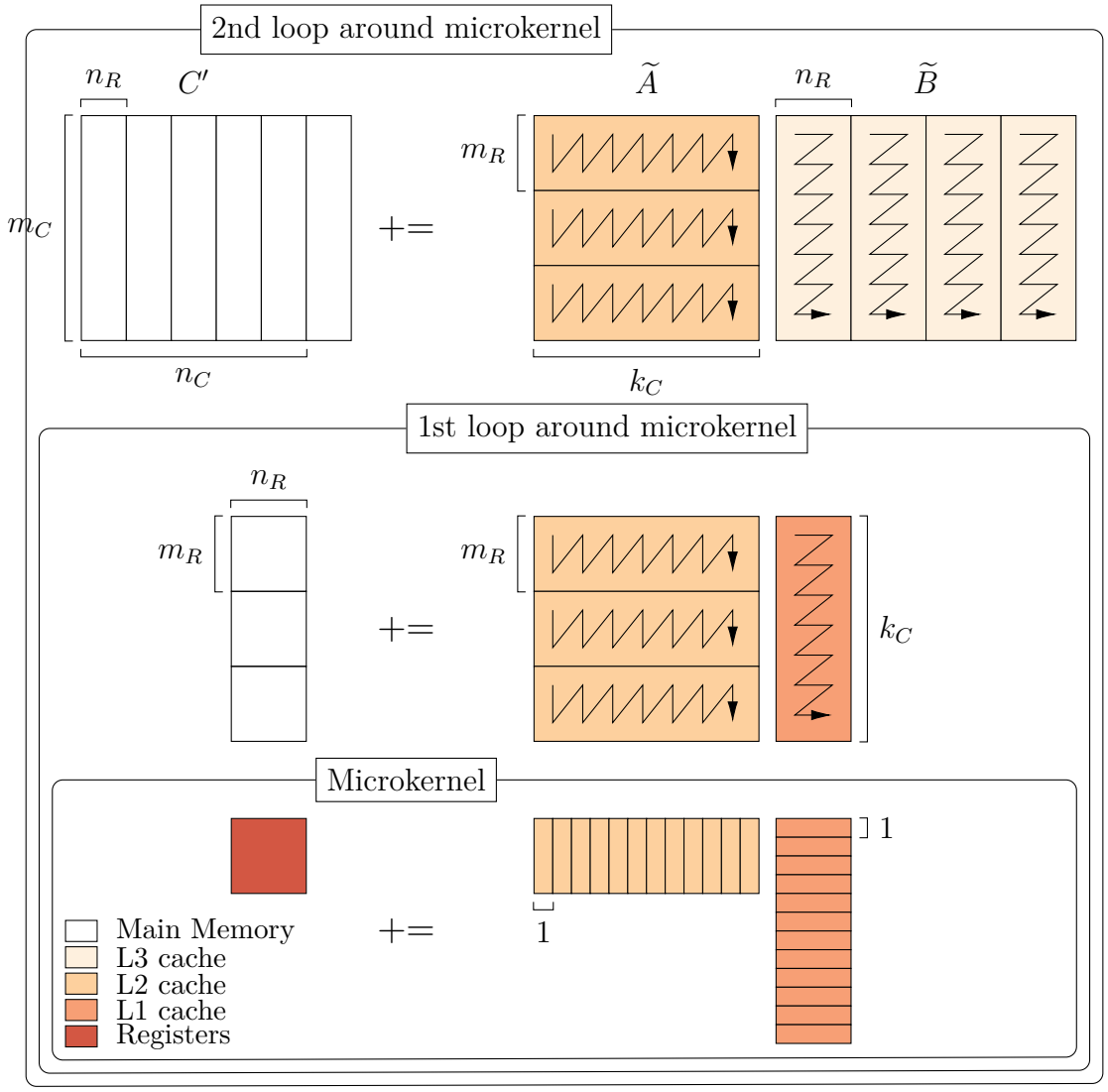
The process of subdividing the operands to GEMM is intended to produce regions with particular shapes that can be computed with high performance or will lead to such shapes. If we consider a $m' \times n'$ subregion M' of a matrix M, we will say that M' is a *block* of M if m' and n' are approximately equal. If m' is much smaller than n', then M' is a *row panel*, while having n' much smaller than m' makes M' a *column panel* of M.

There are two specialized primitives that appear in high-performance GEMM: the *microkernel* and *macrokernel*. The microkernel is a highly-tuned, hand-written function (almost always implemented in assembly) that multiplies an $m_R \times k$ row panel of A by an $k \times n_R$ column panel of B to update an $m_R \times n_R$ block of C, which is updated in registers and written to memory. An illustration of the microkernel is located in Algorithm 1. (In this illustration, as with all others in this work, constants that appear twice indicate the dimension along which each loop proceeds, along with the stride.) The parameters m_R and n_R are based on the microarchitecture of the CPU, which can be derived analytically [9] or by autotuning [13].

Before being passed to the microkernel, the values in the input matrices must undergo *packing*, that is, rearrangement into the specialized packed storage order also illustrated in Algorithm 1. One reason for this is that, to ensure efficiency, the data in each panel on which the microkernel operates must be arranged so that it is read in a linear fashion, which is the pattern of reads best suited to the CPU's memory prefetcher. This is achieved by storing each panel of A in column-major form with columns of height m_R and keeping each panel of B row-major with row width n_R. Since this is not the typical storage format of a matrix, we must rearrange a series of panels from the input into this packed format before calling the microkernel.

A larger contributor to the efficiency of the microkernel is that the packed panels it operates on were resident in cache before the microkernel was called. Since caches are fixed-

Algorithm 1 The macrokernel of a high-performance GEMM implementation.



procedure MACROKERNEL($\widetilde{A}, \widetilde{B}, C'$)
  for $j \leftarrow 0, n_R, \ldots$ **to** n_C **do**
    for $i \leftarrow 0, m_R, \ldots$ **to** m_C **do**
      using the microkernel
      $C'[i : i + m_R, j : j + n_R] \mathrel{+}= \widetilde{A}[i : i + m_R, :] \cdot \widetilde{B}[:, j : j + n_R]$

size, we can only store a fixed number of these panels at any given time. Specifically, we can only store an $m_C \times k_C$ subregion of A and a $k_C \times n_C$ subregion of B.

We will use $\widetilde{A}$ and $\widetilde{B}$ to denote the packed regions of A and B, respectively, and C' to denote the corresponding block of C. The loops that perform $C' \mathrel{+}= \widetilde{A}\widetilde{B}$ form the macrokernel, which is depicted as Algorithm 1. (In all the algorithms presented in this work, we will use Python-style notation for indexing, that is, matrices are 0-indexed and $M[a:b, c:d]$ selects rows $[a, b)$ and columns $[c, d)$, with omitted operands spanning to the beginning/end of the dimension.)

The parameters m_C, k_C, and n_C are chosen to ensure all levels of cache are used efficiently. The process of choosing these parameters, described in Section 2.2, leaves m_C close in size to k_C and keeps n_C much larger then both of these. This means that $\widetilde{A}$ corresponds to a block of A, while $\widetilde{B}$ is a rearranged row panel of B.

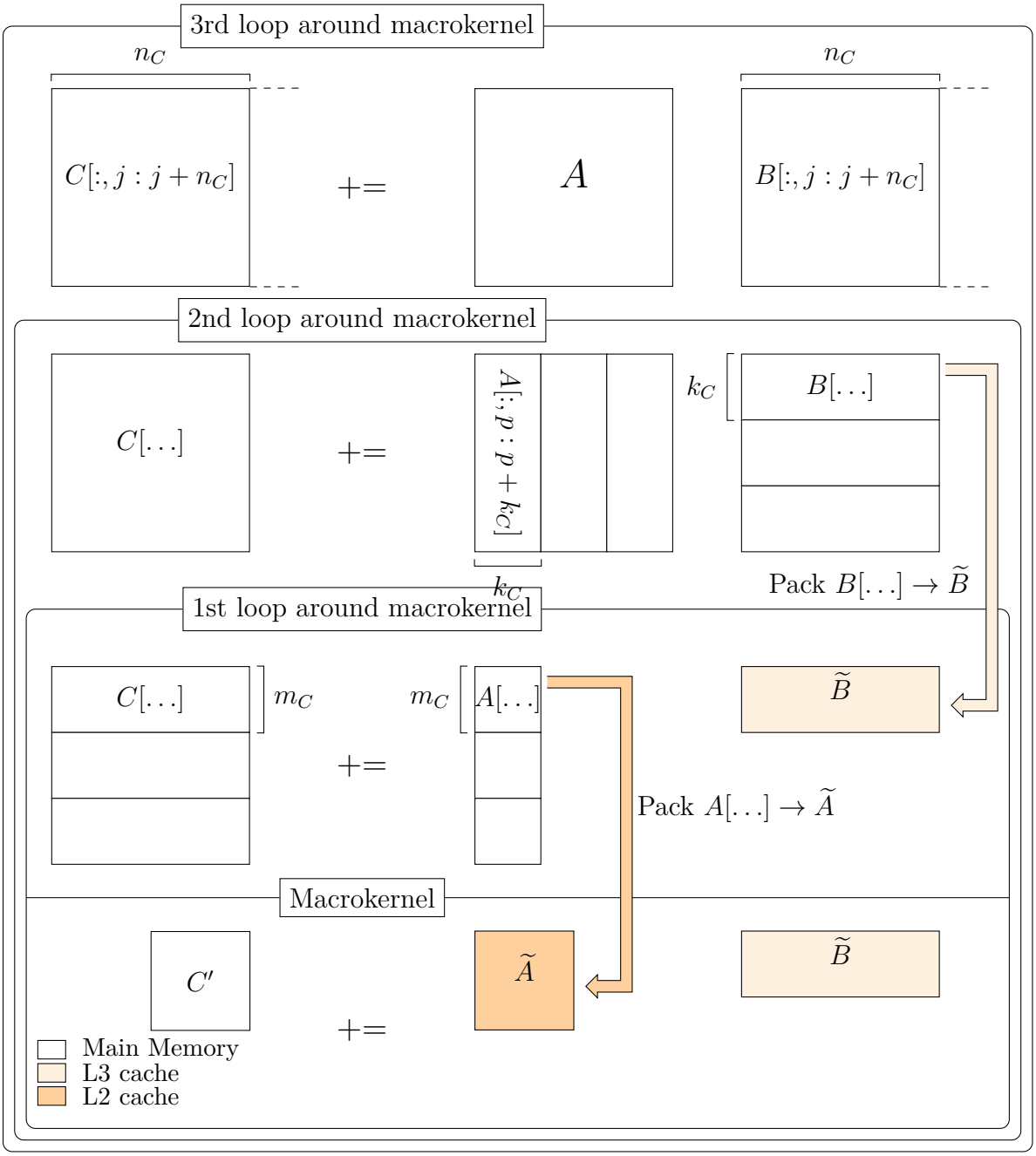
Many high-performance GEMM implementations in use today are based on Goto's algorithm [6]. One such implementation, which we have based our work on, is BLIS [14]. The BLIS algorithm, which is a refactoring of Goto's algorithm, is presented as Algorithms 1 and 2. It brings the row panel of B into the $L3$ cache (later moving its constituent column panels into $L1$), while storing a block of A in $L2$.

2.2 Parameter selection for the BLIS algorithm

The parameters m_R, n_R, k_C, m_C, and n_C that appear in the BLIS algorithm are computed using an analytical model from [9]. We will summarize the process of deriving some of these parameters here. The values of m_R and n_R determine the structure of the microkernel, and are derived from the width of vector registers on a given CPU along with the latency and throughput of fused-multiply-add (FMA) instructions. Since we are reusing existing optimized microkernels, we will not detail the process of selecting these parameters in this work.

The process of selecting the remaining parameters assumes that, for each cache level i,

Algorithm 2 The BLIS algorithm



procedure BLIS_GEMM(A, B, C)
  for $j \leftarrow 0, n_C, \ldots$ **to** n **do**
    for $p \leftarrow 0, k_C, \ldots$ **to** k **do**
      pack $B[p : p + k_C, j : j + n_C] \rightarrow \widetilde{B}$
      for $i \leftarrow 0, m_C, \ldots$ **to** m **do**
        pack $A[m : m + m_C, p : p + k_C] \rightarrow \widetilde{A}$
        MACROKERNEL($\widetilde{A}, \widetilde{B}, C[i : i + m_C, j : j + n_C]$)

the L_i cache has is a W_{Li}-way set-associative cache with N_{Li} sets and C_{Li} bytes per cache line. All the caches are also assumed to have a least-recently-used replacement policy. S_{elem} represents the size of a single element of the matrix for generality.

The first parameter we can derive is k_C, as it appears deepest in the loop structure of the algorithm. We know that we will load a different $n_R \times k_C$ panel of $\widetilde{B}$ from $L1$ during each call to the microkernel. We also know that the microkernel's reads from an $m_R \times k_C$ panel of $\widetilde{A}$ will cause it to be resident in the $L1$ cache. Finally, some values from C' will also enter the cache, and must not evict the panels.

To prevent the panels of $\widetilde{A}$ and $\widetilde{B}$ from evicting each other from the $L1$ cache, we want them to reside in different ways within each cache set. To achieve this, the panels of $\widetilde{A}$ and $\widetilde{B}$ must both have sizes that are integer multiples of $N_{L1}C_{L1}$. This restriction, along with the size of the data in each panel, tells us that

$$m_R k_C S_{elem} = C_A N_{L1} C_{L1}$$

for some integer C_A, and that the same relationship holds for n_R and B.

Given the three sources of data in the $L1$ cache, it must be the case that $C_A + C_B + 1 \leq W_{L1}$ (the 1 is for elements of C'), and that we want to maximize C_B given this constraint. Therefore, we have

$$C_B = \left\lceil \frac{n_R k_C S_{elem}}{N_{L1} C_{L1}} \right\rceil = \left\lceil \frac{n_R}{m_R} C_A \right\rceil$$

Manipulating the inequality further shows us that

$$C_A \leq \left\lfloor \frac{W_{L1} - 1}{1 + \frac{n_R}{m_R}} \right\rfloor$$

Choosing the largest possible value of C_A that leaves k_C an integer will maximize k_C, improving performance by increasing the amount of time spent in the microkernel. Here we note that the process of deriving m_R and n_R leaves us the freedom to swap m_R and n_R (thus

inverting the n_R/m_R term above) if it would improve the value of C_A without incurring a performance penalty.

Now that we have k_C, we can compute m_C and n_C. For m_C, we know that we need to reserve one way in the $L2$ cache for elements of C', and $\lceil (n_R k_C S_{elem})/(N_{L2} C_{L2}) \rceil = C_{B2}$ ways for the panel of B in the $L1$ cache. This allows us to bound m_C by

$$m_C \leq \left\lfloor \frac{(W_{L2} - C_{B2} - 1) N_{L2} C_{L2}}{k_C S_{elem}} \right\rfloor$$

and then select m_C as large as possible, keeping in mind that it must be divisible by m_R for high performance.

Since the $L3$ cache is, in practice, very large and somewhat slow, n_C is computed by finding the largest value such that $n_C k_C$ is less that the size of the $L3$ cache, excluding an $L1$ cache's worth of space to allow elements of C' or $\widetilde{A}$ to pass through.

2.3 Data reuse in matrix multiplication

Examining the loops in the BLIS algorithm, as was done in [9], shows that each loop causes some data from the computation to be reused, that is, read or written in its entirety during each iteration of the loop. For example, the micro-kernel reuses the small block of C' that is stored in registers while looping over the panels.

Even though reuse occurs at each level of the algorithm, we will focus on the reuse of the packed buffers, as this is a key consideration for maintaining the performance of the algorithm, as shown by [7]. Packing into $\widetilde{B}$ and $\widetilde{A}$ introduces overhead, which is time that is not usable for computation. Therefore, reusing the packed buffers many times will lower the relative proportion of time spent on packing, thus increasing performance.

Consider the multiplication $C \mathrel{+}= AB$, where A is $m \times k$ and B is $k \times n$. To improve reuse of $\widetilde{B}$ during this computation, we need the first loop around the macrokernel, which packs into $\widetilde{A}$, to run many times, since each of those iterations covers computations that read

the entirety of $\widetilde{B}$. That is, we need m to be large, as small values of m will hurt performance by reducing the time between successive $\widetilde{B}$ packing operations. Similarly, large values of n allow the second loop around the microkernel to execute many times, thus improving reuse of $\widetilde{A}$, which is consumed by the first loop around the microkernel.

The only loops that iterate over k are those that pack $\widetilde{B}$ (the second loop around the macrokernel) and the microkernel. The second loop around the macrokernel does not result in the reuse of any data that has been loaded into cache because it is the first loop that performs such loads. The microkernel only reuses data in registers, and so does not reuse cached data. However, if k is much less than k_C, the microkernel will only iterate a few times, increasing the proportion of time spent on non-computational overhaed. Therefore, the size of k has little impact on the reuse of packed blocks, except when $k \ll k_C$.

This reasoning shows that the BLIS algorithm achieves its best performance when m and n are large and k near a multiple of k_C.

3 Algorithm

For our discussion of GEMM3, we well be considering the operation $G \mathrel{+}= DEF$, where D is $m \times k$, E is $k \times l$ and F is $l \times n$.

3.1 Deriving GEMM3

To derive our algorithm for GEMM3, we considered GEMM(D, EF, G), with the BLIS algorithm, looking for when elements of EF were needed and whether it was possible to only compute parts of that product as necessary. We define the computation of $G \mathrel{+}= D(EF)$ to be the *outer algorithm*, and the operations needed to compute parts of $E \cdot F$ when needed to be the *inner algorithm*. Analyzing the outer algorithm showed that the earliest we would need to read from EF was the step where GEMM packs B to $\widetilde{B}$, and so the inner algorithm would need to compute the appropriate part of $B = EF$ before this step.

The first step of the GEMM algorithm reduces $C \mathrel{+}= AB$ (or, in our case, $G \mathrel{+}= D(EF)$) to the following subproblems:

$$\left[\begin{array}{c|c|c|c|c|c} C_{,1} & C_{,2} & \cdots & C_{,j} & \cdots & C_{n \mod n_C} \end{array}\right] = A \left[\begin{array}{c|c|c|c|c|c} B_{,1} & B_{,2} & \cdots & B_{,j} & \cdots & B_{n \mod n_C} \end{array}\right]$$

If we substitute in the values we're computing with, we have:

$$\left[\begin{array}{c|c|c|c|c|c} G_{,1} & G_{,2} & \cdots & G_{,j} & \cdots & G_{n \mod n_C} \end{array}\right] = D\left(E \left[\begin{array}{c|c|c|c|c|c} F_{,1} & F_{,2} & \cdots & F_{,j} & \cdots & F_{n \mod n_C} \end{array}\right]\right)$$

Now, for the jth subploblem, the matrices will be partitioned in the k dimension as follows:

$$C_{,j} = \left[\begin{array}{c|c|c|c|c|c} A_{,1} & A_{,2} & \cdots & A_{,p} & \cdots & A_{,k \mod k_C} \end{array}\right] \left[\begin{array}{c} B_{1,j} \\ \hline \vdots \\ \hline B_{p,j} \\ \hline \vdots \\ \hline B_{k \mod k_C,j} \end{array}\right]$$

For the GEMM3 computation, we have

$$G_{,j} = \left[\begin{array}{c|c|c|c|c|c} D_{,1} & D_{,2} & \cdots & D_{,p} & \cdots & D_{,k \mod k_C} \end{array}\right] \left(\left[\begin{array}{c} E_{1,} \\ \hline \vdots \\ \hline E_{p,} \\ \hline \vdots \\ \hline E_{k \mod k_C,} \end{array}\right] F_{,j}\right)$$

At this point in the algorithm, the subregion $B_{p,j}$ is a row panel of B with a fixed size that will be packed into $\widetilde{B}$, which resides in $L3$ cache. We cannot, however, pack from $(EF)_{p,j}$, as we have not computed these values. However, if we examine the regions of E and F that correspond to $B_{p,j}$, we notice that $E_{p,}$ is a $k_C \times l$ row panel of E and that $F_{,j}$ is an $l \times n_C$ block of F (under typical conditions), which is a multiplication that can be performed

efficiently.

Since the next step in the BLIS algorithm requires the region $(EF)_{p,j}$ to be computed, we must either compute this region now or compute $EF_{,j}$ during the previous loop. Computing during the previous loop would result in an arbitrarily large product, preventing us from attaining the memory savings we were searching for. However, the product $E_{p,}F_{,j}$ comprises $k_C n_C$ floats, which can be stored in $O(1)$ memory. This shows that the current point in the GEMM algorithm is the only time in which we can compute a subregion of EF while keeping the algorithmic properties we sought.

3.2 The inner algorithm

Now that we have identified the computations that the inner algorithm must perform, we must turn to the problem of deriving it.

The multiplication of a row panel by a block is one that can be carried by the BLIS algorithm without a significant performance loss despite a somewhat suboptimal shape. Therefore, an initial approach to computing $(EF)_{p,j} = E_{p,}F_{,j}$ would be to reuse the BLIS algorithm, nesting it within the outer computation. However, in order to ensure performance, we need to make several changes to the BLIS algorithm. The simplest change is to remove the redundant outermost loop of the inner algorithm, since n cannot be more than n_C for its inputs.

The next change we must make is to instruct the microkernel to write its output in the packed format by informing it the output matrix is stored in row-major form with width n_R. This allows us to avoid having to pack the output of the inner algorithm, eliminating a large number of otherwise required memory operations and the corresponding overhead. The change allows us to write $\widetilde{EF}$ directly, eliminating the "pack $\widetilde{B}$" step from the outer algorithm. Therefore, we can simply consider the inner algorithm to be performing the operation $\widetilde{EF} = E_{p,}F_{,j}$.

We must also use an n_C value equal to half of the n_C value derived for the BLIS algorithm

for both the inner and outer algorithms. This prevents the packed panels of $\widetilde{F}$ from evicting the output $\widetilde{EF}$ from $L3$ cache, where it needs to reside in order to satisfy the assumptions of the outer algorithm. The inner algorithm must place its output in $L3$ cache to prevent unnecessary memory operations from being performed. This change to n_C does not break the assumptions of the BLIS algorithm, as $n_C/2$ is still much larger than k_C or m_C in practice.

The other parameter adjustment that needs to be performed is to reduce k_C so that it is divisible by m_R if that is not already the case. If this is not done, the BLIS macrokernel's loops will a final iteration that considers fewer than m_R rows during each invocation of the inner algorithm since k_C becomes the m dimension for the inner problem. Computation of this fringe iteration has a performance impact, as special processing is required to account for the unusual sizes, especially since the loop that this cost would normally be amortized over only runs for a few iterations in the inner algorithm. This gains from eliminating this source of low performance outweigh the losses from having a slightly suboptimal k_C value. However, l_C, which is the inner algorithm's version of k_C, can be kept at the BLIS value, as these performance considerations do not arise in this stark fashion there.

It should be noted that one flaw in our approach is that the inner computation is a panel-matrix multiply, that is, the m dimension is small, and $\widetilde{F}$ sees little reuse. This suboptimal shape did have a performance impact, though it turned out to be small in practice.

The algorithm that results from this derivation is Algorithm 3, which is illustrated in Figure 1.

3.3 Variations for similar problems

It is important to note that this algorithm computes $G \mathrel{+}= D(EF)$. In some cases, it is much more efficient (on account of the dimensions of the inputs) to compute $G \mathrel{+}= (DE)F$ instead. Attempting to derive an algorithm for that problem directly does not work, as the block $\widetilde{DE}$ in $L2$ is not of a shape that can be multiplied efficiently by the BLIS algorithm. Fortunately, we can transform problems of this form to the equivalent problem $G^T \mathrel{+}= F^T(E^T D^T)$. This

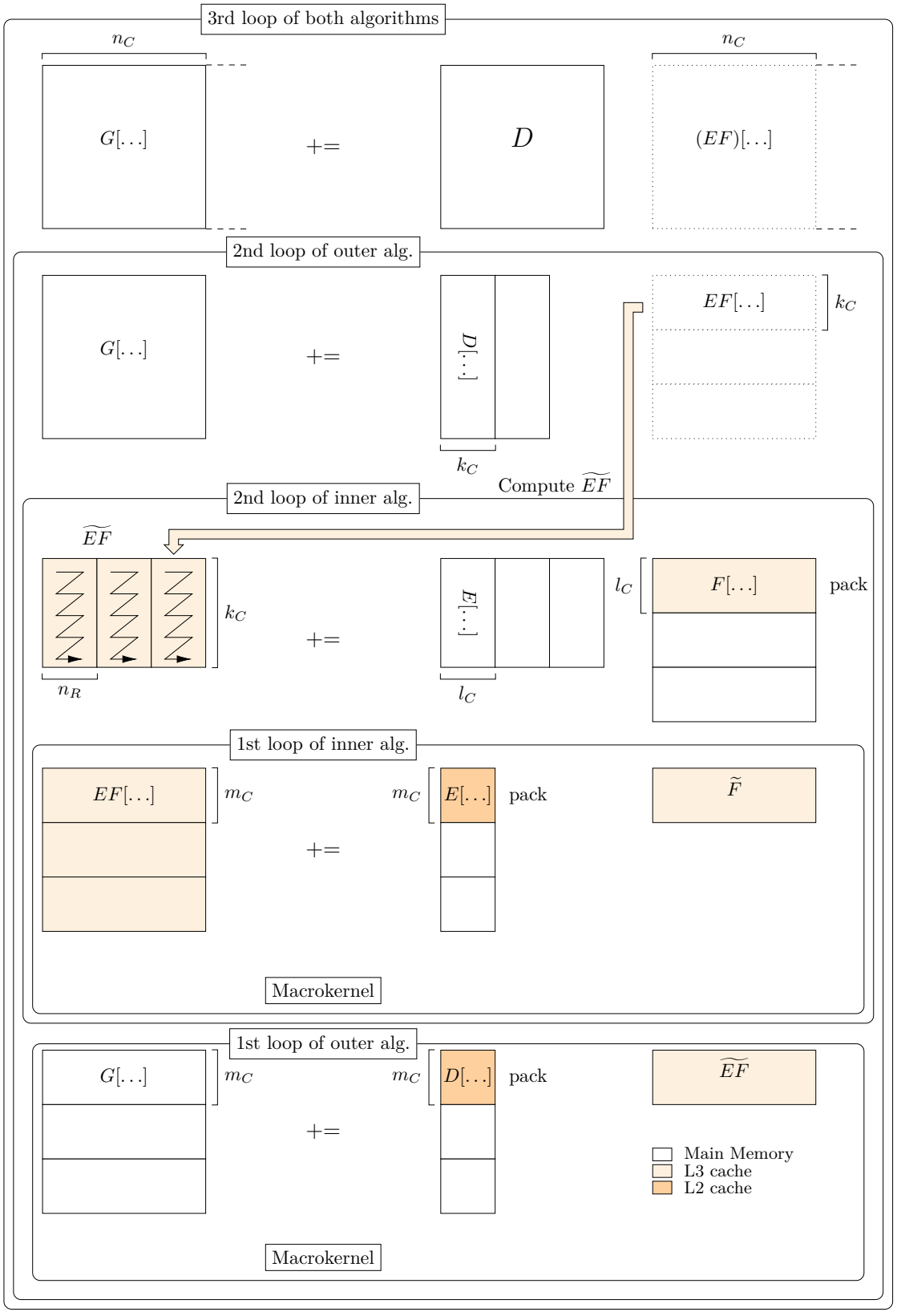


Figure 1: Illustration of algorithm for GEMM3. Constants that appear twice indicate the dimension along which each loop proceeds, along with the stride.

Algorithm 3 Algorithm for GEMM3

```
procedure GEMM3(D, E, F, G)
    for j ← 0, n_C, ... to n do
        for p ← 0, k_C, ... to k do
            for q ← 0, l_C, ... to l do
                pack F[q : q + l_C, j : j + n_C] → F̃
                for i ← 0, m_C, ... to k_C do
                    pack E[i : i + m_C, q : q + l_C] → Ẽ
                    MACROKERNEL(Ẽ, F̃, ẼF̃)                ▷ writes in packed form
            for i ← 0, m_C, ... to m do
                pack D[m : m + m_C, p : p + k_C] → D̃
                MACROKERNEL(D̃, ẼF̃, G[i : i + m_C, j : j + n_C])
```

transformation can be performed at effectively no cost by re-interpreting the memory the matrices are stored in (treating a matrix stored in row-major form as column-major and vice-versa).

4 Experiments and Results

4.1 Implementation

To test our algorithm's performance, we implemented both it and the BLIS algorithm in the Multilevel Optimization for Matrix Multiply Sandbox (MOMMS), which was developed for [12]. This framework allowed us to declaratively generate the code for an algorithm by specifying the series of loops and packing operations required, and allowed us to interface to the BLIS microkernels. Initially, we verified that the MOMMS implementation of the BLIS algorithm had similar performance to the reference implementation, which was written in C. Then, we modified the MOMMS framework to support generating code for multiplying three matrices together. These modifications allowed us to express both our GEMM3 algorithm and the typical $T = EF; G \mathrel{+}= DT$ method we were comparing against.

We tested the performance, in billions of floating point operations (GFLOPS) per second, of both our algorithm and the typical approach. The target machine had an Intel Xeon E3-

	GEMM3	BLIS algorithm
m_R	6	6
n_R	8	8
m_C	72	72
k_C	252	256
l_C	256	
n_C	2040	4080

Table 1: Blocking parameters for Haswell

1270 v3 CPU, which ran at 3.5 GHz and implemented the Haswell microarchitecture. The experimental system had 15 GB of RAM, 8 MB of $L3$ cache, 256 KB of $L2$ cache (which was an 8-way set-associative cache with 64 byte cache lines) and 32 KB of $L1$ cache (per core, with the same associativity and line sizes as the $L2$ cache). The theoretical peak performance of this system on matrix multiplication is 56 GFLOPS per second, which is the top of the y-axis on all out performance plots. The blocking parameters that arose from this data can be found in Table 1

4.2 Experiments performed

Our experiments were run both on square matrices ($m = n = k = l = N$, where D is $m \times k$, E is $k \times l$, F is $l \times n$, and G is $m \times n$) and on problems where a particular dimension was fixed at 9 while the others varied together, to evaluate the performance of these algorithms on multiple input shapes. We sampled the performance at multiples of sixteen from $N = 16$ to 4912 to test the suitability of our approach on many input sizes. All the inputs were stored in column-major form, while the temporaries and outputs were stored in row-major form, since the BLIS microkernel has noticably increased performance when writing to row-major matrices (and since the packed buffer is row-major). This unusual combination of input formats ensured a fair comparison between the two algorithms. In addition, we tested the $G^T = F^T(E^T D^T)$ case, in which all matrices were row major, for square inputs.

a plot of the computed additional memory consumption (excluding inputs and outputs) for multiplying square matrices using both algorithm is shown in Figure 2, demonstrating

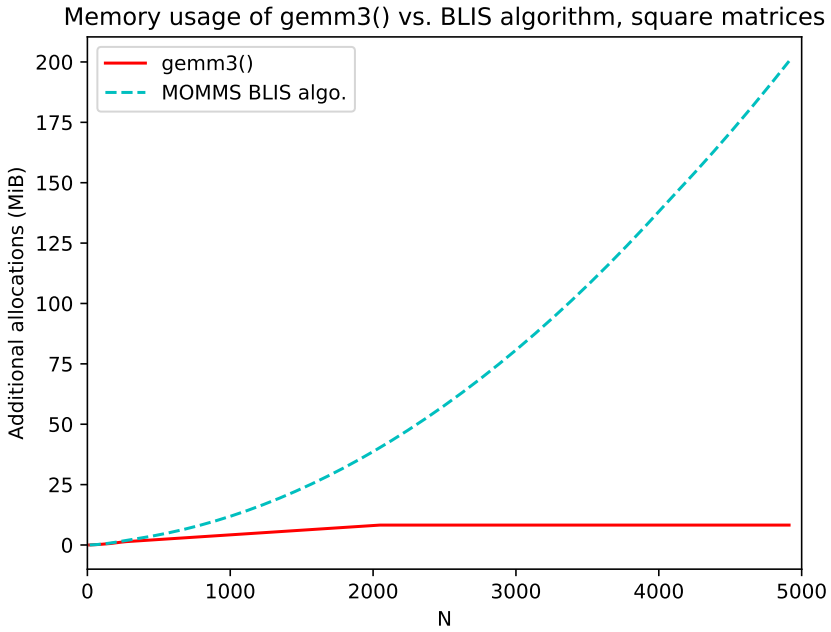


Figure 2: Memory usage of our GEMM3 implementation compared to a pair of GEMM calls

our approach's much lower memory usage.

The results of these experiments can be found in Figure 3 for square matrices, and Figure 4 for the rectangular inputs. The $G \mathrel{+}= (DE)F$ experiment's results are in Figure 5. Data for the square case shows that, once performance has stabilized around $N = 512$, the two algorithms are almost always within 5% of each other in GFlops/s, showing comparable performance. In addition, a plot of the computed additional memory consumption (excluding inputs and outputs) for multiplying square matrices using both algorithm is shown in Figure 2, demonstrating our approach's much lower memory usage.

4.3 Analysis

From the figures in the previous section, it is clear that GEMM3 attains similar performance to a pair of GEMM calls with significantly lower memory usage. However, it is important to dissect these performance results and determine why they have arisen.

For the square case, which was shown in Figure 3, we can observe that, for most of

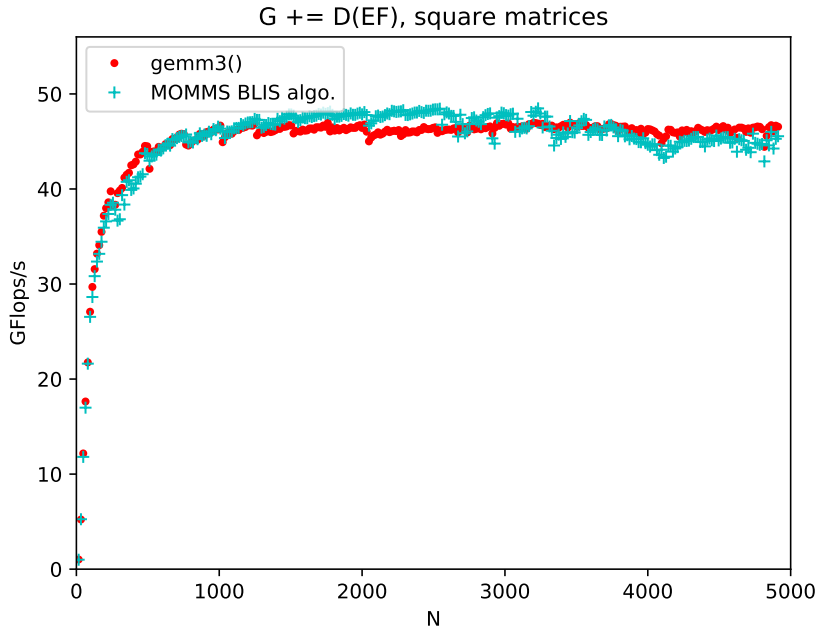


Figure 3: Performance of our GEMM3 implementation compared to a pair of GEMM calls, square matrices

the 1000s and 2000s, the pair of GEMM calls has somewhat increased performance, but that the GEMM3 algorithm has comparable or higher performance afterwards. The higher performance for the GEMM calls arises from the lack of reuse of $\widetilde{F}$ implied by the $252 \times l \cdot l \times n$ input to the inner algorithm for GEMM3. Specifically, the first loop around the macrokernel in that algorithm always iterates at most four times, compared to the unbounded number of iterations in a call to GEMM to compute EF. However, around when the temporary grows to 64 MB, the memory cost of writing it to memory becomes large enough that the performance of two GEMM calls begins to drop to and eventually falls below the performance of our GEMM3 algorithm. This shows that our approach, in addition to the memory savings, provides performance benefits for large matrix multiplications.

For small input sizes ($N < 512$), there is often a significant performance benefit attained by the GEMM3 algorithm. This performance improvement arises from the additional packing step that a pair of GEMM calls must perform, which has a non-negligible cost for these inputs, even if both the packed and unpacked data remain resident in the $L3$ cache.

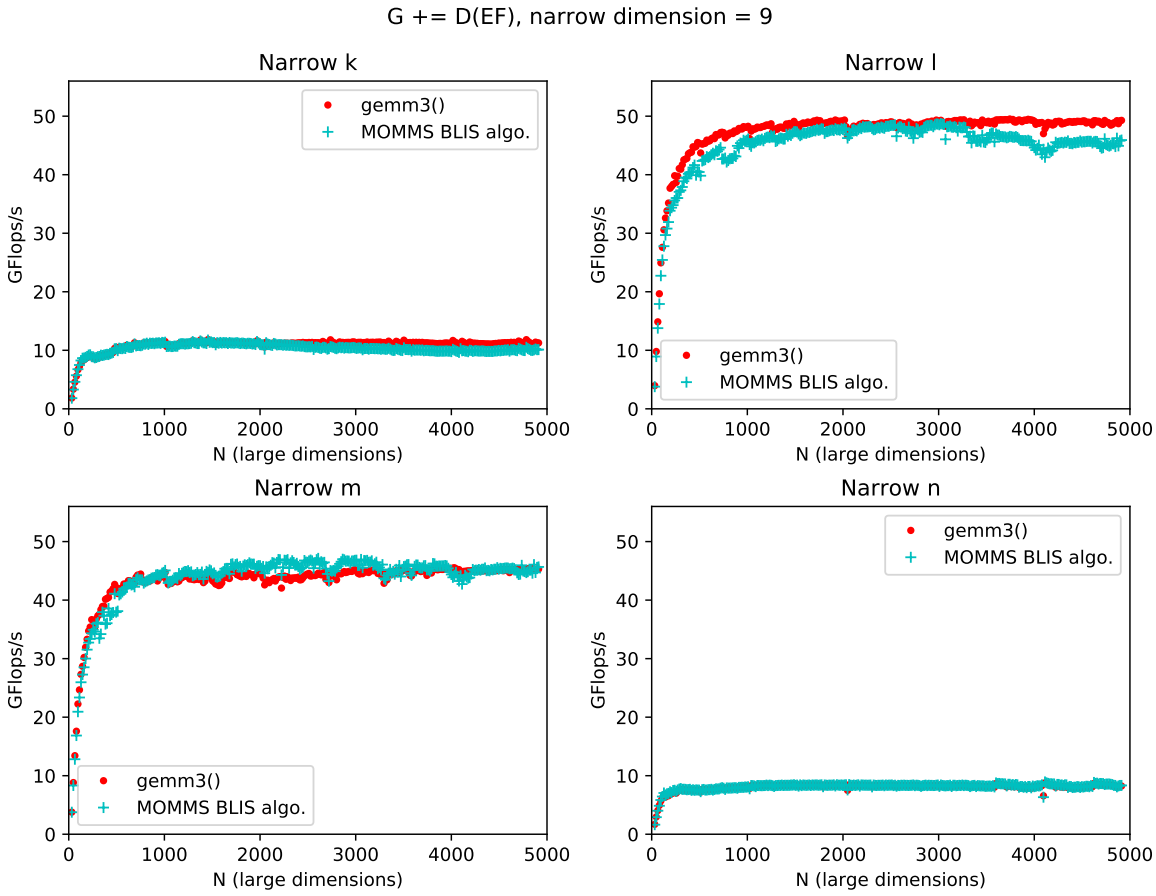


Figure 4: Performance of our GEMM3 implementation compared to a pair of GEMM calls, rectangular matrices

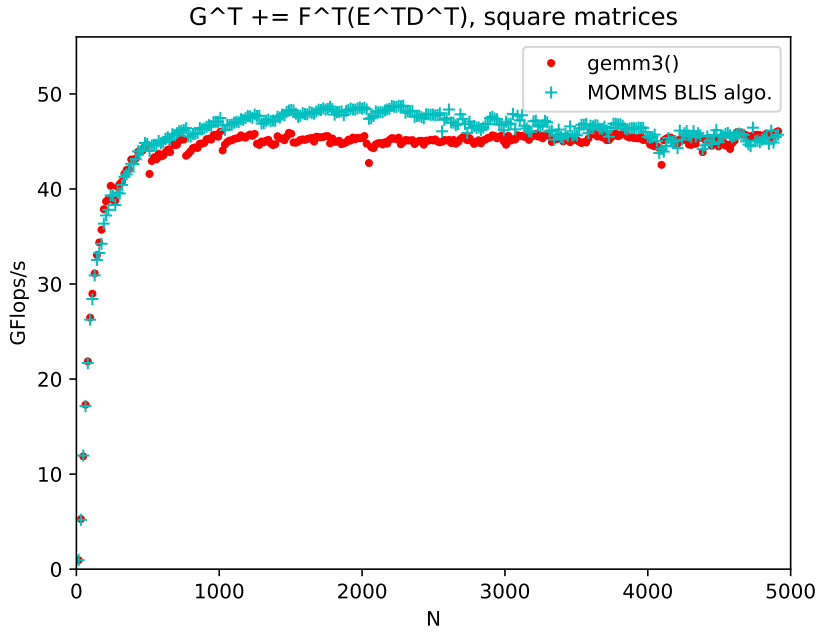


Figure 5: Performance of our GEMM3 implementation compared to a pair of GEMM calls, square transposed matrices

Similar trends can be seen in Figure 5, which shows the results for $G \mathrel{+}= (DE)F$, implemented as $G^T \mathrel{+}= F^T(E^T D^T)$. However, the pair of GEMM calls retains higher performance for a longer period of time on this workload. This occurs because packing from row-major matrices to row-major packed buffers, which is something that occurs more frequently in this experiment, is much more amenable to the CPU's prefetcher than packing from column-major matrices.

The rectangular-matrix results from Figure 4 also require examination. When l is small, the GEMM3 algorithm performs better on all inputs, as the memory-writing overhead in the BLIS algorithm remains, while the advantage of reuse in computing EF is reduced (because the k dimension is very narrow, creating a memory-intensive outer product). When m is small, the costs of low $L3$ reuse are imposed at least once on both algorithms, decreasing the performance of both.

When n is narrow, the problem is effectively converted to a series of several matrix-vector multiplications, which cannot achieve high performance with either algorithm. For narrow

k, the outer problem is an outer product, while the inner problem is a panel-matrix multiply (wide matrix times square matrix). These shapes do not generally promote data reuse within the BLIS algorithm. The eventually increased performance of GEMM3 with $k = 9$ is due to the lowered number of memory operations that algorithm performs, since memory operations dominate this computation.

These results demonstrate the general suitability of our algorithm for GEMM3 and its suitability as a high-performance primitive.

5 Conclusions

We have demonstrated that, by exploiting the structure of the BLIS algorithm for matrix-matrix multiplication, we can derive an algorithm for GEMM3 (multiplying three matrices together) that only requires a constant amount of additional memory. We have shown that, while the basic concept of our algorithm arises naturally from the structure of GEMM, several tweaks are needed to retain high performance.

Experimental results have indicated that, on a variety of workloads, our algorithm maintains a similar level of performance to existing approaches to this problem. We know that, on sufficiently small or large problems of certain shapes, our approach offers increased performance because it eliminates a source of memory operation overhead. However, there are input sizes that are known to have a small negative performance impact on our approach due to the suboptimal multiplication shape our approach used as a subproblem. Neither these gains or losses are particularly large, so we conclude that the main advantages of our approach are the memory savings it offers.

An additional improvement offered by GEMM3 is that it introduces a cleaner programming interface for the multiplication of three matrices, which is a problem that often arises in practice. Therefore, our algorithm will give both programmers and authors of linear algebra code generators (such as Linnea [2]) an interface that is easier to invoke and does not require

the manual allocation of a temporary buffer.

Future Work This work only investigates a sequential implementation of GEMM3. In the future, we intend to investigate how we can efficiently parallelize our GEMM3 algorithm to take advantage of the multiple CPU cores available on modern CPUs. We also plan to verify the portability of our results by testing on multiple different CPU microarchitectures, such as the recent Knight's Landing processors from Intel.

Acknowledgments We would like to thank Prof. Robert van de Geijn for advising this research and directing us towards the research question. We also wish to thank Dr. Tyler Smith for writing MOMMS and assisting with the algorithm design, and Prof. Tze Meng Low for his advice on resolving certain performance issues. Finally, we wish to acknowledge **TODO get grant info** National Science Foundation for funding this work through grant #NNNNNN.

References

[1] E. Anderson et al. *LAPACK Users' Guide*. Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999. ISBN: 0-89871-447-8 (paperback).

[2] Henrik Barthels and Paolo Bientinesi. "Linnea: Compiling Linear Algebra Expressions to High-Performance Code". In: *The 8th International Workshop on Parallel Symbolic Computation*. Kaiserslautern: ACM, 2017.

[3] Henrik Barthels, Marcin Copik, and Paolo Bientinesi. "The Generalized Matrix Chain Algorithm". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 2018, pp. 138–148. ISBN: 9781450356176. DOI: 10.1145/3168804. URL: https://dl.acm.org/citation.cfm?id=3168804.

[4] Jaeyoung Choi, Jack J Dongarra, and David W Walker. *The Design of a Parallel Dense Linear Algebra Software Library: Reduction To Hessenberg, Tridiagonal, and Bidiagonal Form*. Tech. rep. Oak Ridge National Laboratory, 1995.

[5] Jack Dongarra. "Basic linear algebra subprograms technical (BLAST) forum standard". In: *The International Journal of High Performance Computing Applications* 16.2 (2002), pp. 115–115.

[6] Kazushige Goto and Robert A. van de Geijn. "Anatomy of high-performance matrix multiplication". In: *ACM Transactions on Mathematical Software* 34.3 (2008), pp. 1–25. ISSN: 00983500. DOI: 10.1145/1356052.1356053. URL: http://portal.acm.org/citation.cfm?doid=1356052.1356053.

[7] Greg Henry. *BLAS based on block data structures*. Theory Center Technical Report CTC92TR89. Cornell University, Feb. 1992.

[8] TC Hu and MT Shing. "Computation of matrix chain products. Part II". In: *SIAM Journal on Computing* 13.2 (1984), pp. 228–251. DOI: 10.1137/0213017. URL: http://epubs.siam.org/doi/pdf/10.1137/0213017.

[9] Tze Meng Low et al. "Analytical Modeling Is Enough for High-Performance BLIS". In: *ACM Transactions on Mathematical Software* 43.2 (2016), pp. 1–18. ISSN: 00983500. DOI: 10.1145/2925987. URL: http://dl.acm.org/citation.cfm?doid=2988256.2925987.

[10] Devangi N Parikh, Margaret E Myers, and Robert A Van De Geijn. "Deriving Correct High-Performance Algorithms". Austin, 2017. URL: https://arxiv.org/abs/1710.04286.

[11] Vishwas Rao et al. "Robust Data Assimilation Using L_1 and Huber Norms". In: *SIAM Journal on Scientific Computing* 39.3 (2017), pp. 548–570. DOI: 10.1137/15M1045910.

[12] Tyler Michael Smith. “Theory and Practice of Classical Matrix-Matrix Multiplication for Hierarchical Memory Architectures”. PhD. The University of Texas at Austin, 2017. URL: https://repositories.lib.utexas.edu/bitstream/handle/2152/63352/SMITH-DISSERTATION-2017.pdf?sequence=1%7B%5C&%7DisAllowed=y.

[13] Clint R Whaley and Jack J Dongarra. “Automatically Tuned Linear Algebra Software”. In: *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. ACM, 1998, pp. 1–27.

[14] Field G van Zee et al. “The BLIS Framework: Experiments in Portability”. In: *ACM Transactions on Mathematical Software* 42.2 (2016).