GEMM3(): Constant-workspace high-performance multiplication of three matrices for matrix chaining

Krzysztof A. Drewniak

February 22, 2018

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[5], as are simpler $O(n^3)$ dynamic-programming solutions[2] These algorithm 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[2]. For example, an algorithm to compute the ensemble Kalman filter[8] (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 $L^{-1} A L^{-H}$ (where L is lower triangular and A is symmetric), which arises in blocked algorithms for the two-sided triangular linear solve[7] 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[3].

The BLAS and LAPACK 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 against them requires expert knowledge. Existing high-level systems, such at Matlab and Julia, which are commonly used by non-experts, generally compute the expressions in a naive way, such as proceeding from left to right, which often yields suboptimal performance. Systems such as Linnea[1] have emerged recently that attack the generalized matrix chain problem and automatically generate a high-performance program for a given expression.

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 inverson or an outer product. The general form of such a problem is $D \coloneqq \alpha ABC + \beta D$, which we will summarize as $D \mathrel{+}= ABC$ and denote GEMM3(), by analogy with the notation for general matrix-matrix multiply (GEMM()) in the BLAS library. The only approach available to this subproblem with current methods is to paranthesize the expression ABC and perform a pair of matrix multiplications, storing a result in a temporary matrix T. This method two drawbacks: the first is that T is often rather large and requires significant amounts of storage space. In addition, if T is sufficiently large, reading and writing it incurs a non-negligible performance cost associated with reading and writing main memory.

To combat these issues, we have developed an algorithm for GEMM3() which does not require an entire intermediate product to be stored at one time. Our algorithm takes advantage of the blocking performed by modern GEMM() implementations to only store a cache-sized block of BC 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, the complexity of the solutino, 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 problem to a series of multiplications of blocks from the inputs. These blocks 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 from cache durinf further blocking. The multiple reductions are required to effectively utilize all levels of the cache. These steps are necessary because of the memory hierarchy of modern CPU, where there are multiple (typically) three levels of cache, which increase in size but decrease in speed, between registers and main memory.

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 $m_R \times k$ panel of A by an $k \times n_R$ panel of B to update an $m_R \times n_R$ region of C, which is computed in registers and written to memory. m_R and n_R are constants based on the microarchitecture of the CPU, which can be derived analytically[6] or by autotuning[10].

In order for the microkernel to operate efficiently, the panels it operates on must be loaded into the system's caches beforehand. In addition, he data within the panels must be arranged so that the microkernel's memory reads will operate contiguously. This is achieved by having each panel of A be stored row-major with rows of width m_R and each panel of B stored column-major with height n_R, which causes the data needed by the microkernel to be laid out contiguously.

To allow the microkernel to stream the data it operates on efficiently, the panels it examines must be stored in cache. Since caches are fixed-size, we can store a fixed number of such panels at any given time. Specifically, we can only store an $m_C \times k_C$ block of A and a $k_C \times n_C$ block of B. The process of rearranging these blocks into the format needed by the microkernel is known as *packing*. The constants m_C, k_C, and n_C are chosen to ensure all levels of cache are used efficiently.

We will use $\tilde{A}$ and $\tilde{B}$ to denote the packed blocks of A and B, respectively, and C' to denote the corresponding block of C. The loops that perform $C' \mathrel{+}= \tilde{A}\tilde{B}$ form the macrokernel, which is depicted as Algorithm 1. Since the loops that perform $C' \mathrel{+}= \tilde{A}\tilde{B}$ are common across all the algorithms we will discuss, we will abstract them into the *macrokernel*, which is shown 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.)

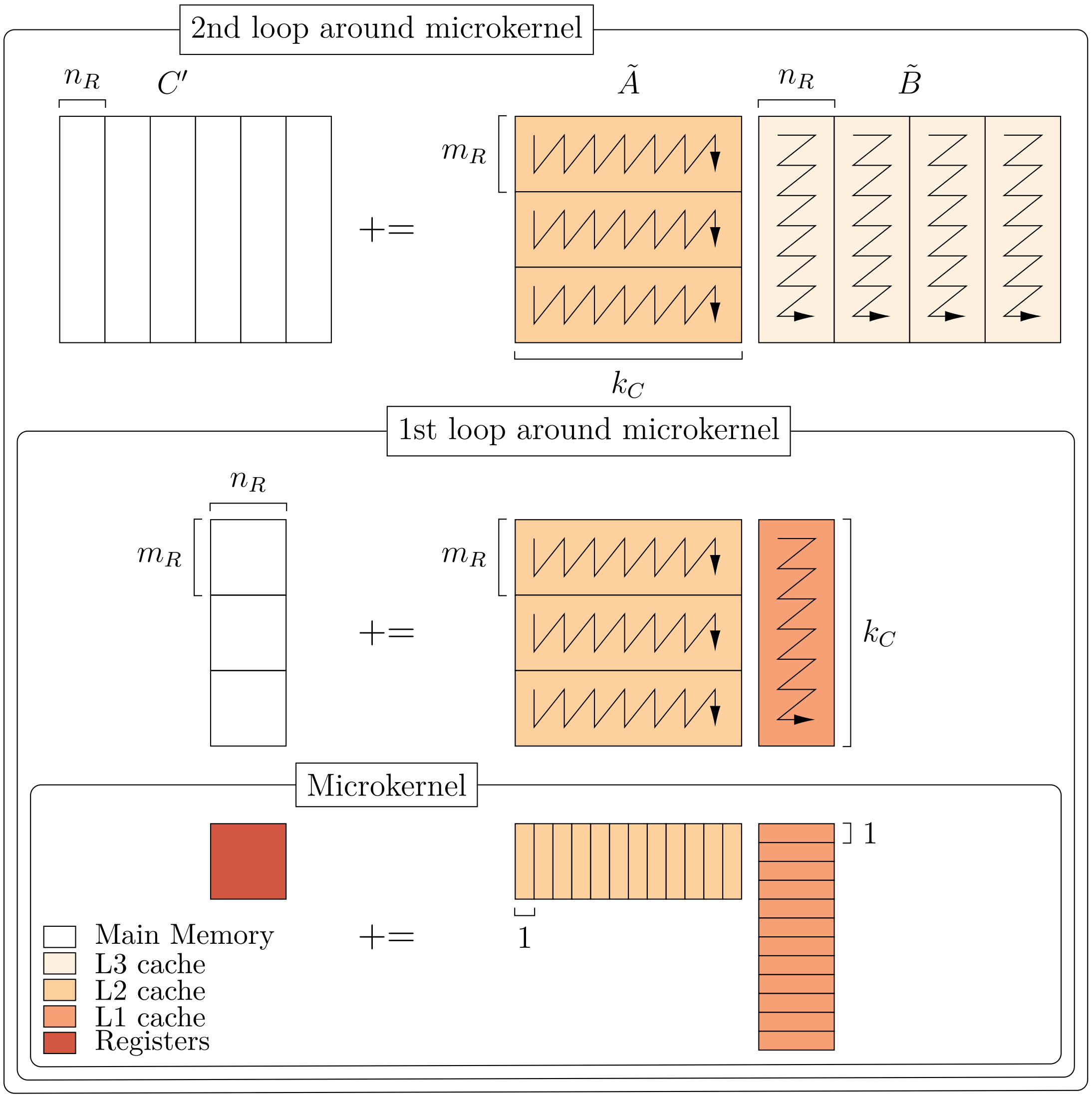
Many high-performance GEMM() implementations in use today are based on Goto's algorithm[4]. One such implementation, which we have based our work on, is BLIS[11]. The BLIS algorithm, which is an elaboration on Goto's algorithm, is presented as Algorithm 2.. It brings elements of B into the $L3$ (and later $L1$) cache, while storing elements of A in $L2$.

2.2 Constant selection for the BLIS algorithm

The constants m_R, n_R, k_C, m_C, and n_C that appear in the BLIS algorithm are computed using an analytical model from [6]. We will summarize the process of deriving some of these constants 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 constants except to note that m_R and n_R can be swapped during the cache-constant selection process without sacrificing performance.

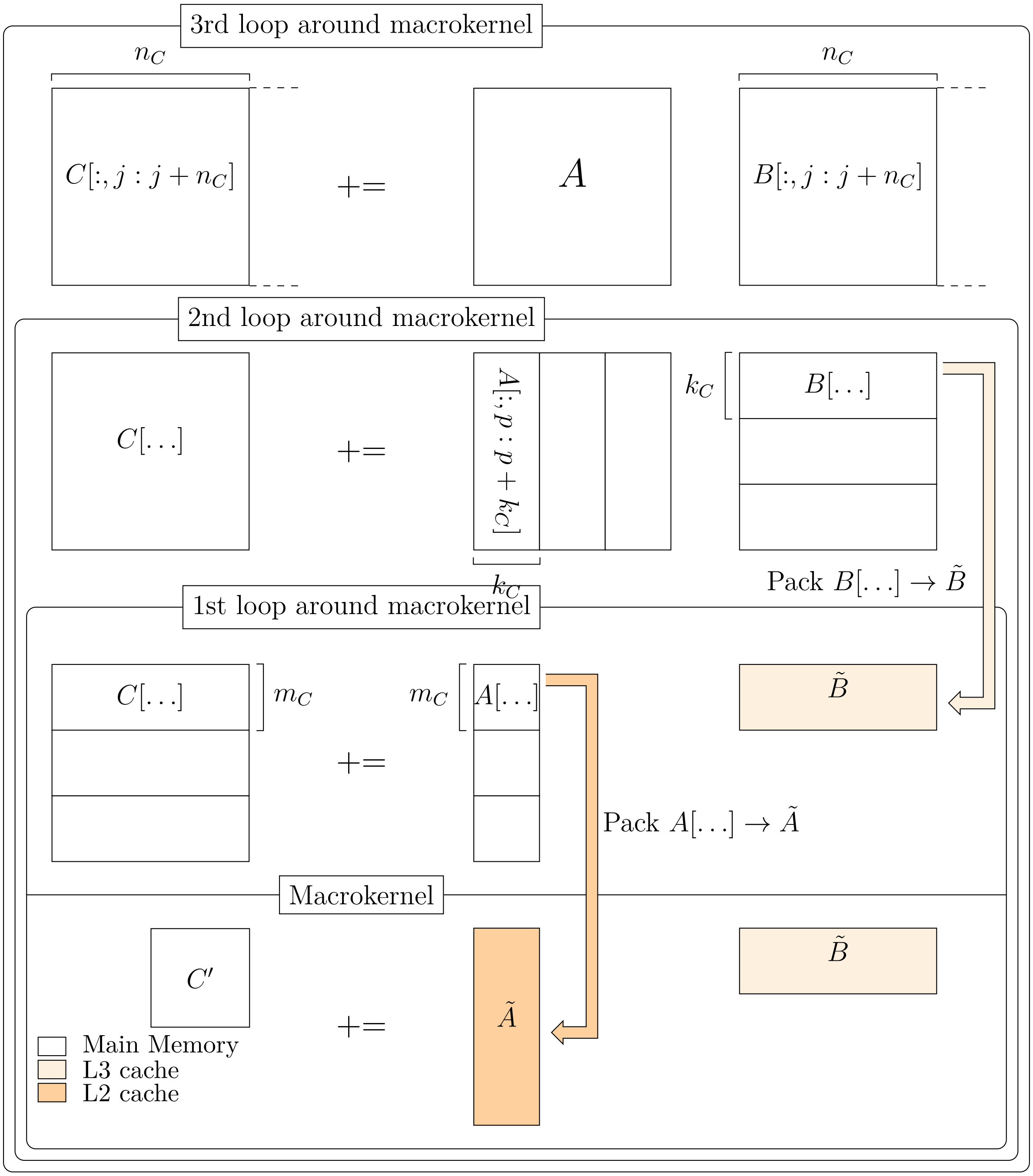
The process of selecting the remaining constants assumes that, for each cache level i, the Li cache has is a W_{Li}-way set-associative cache with N_{Li} sets and C_{Li} bytes per cache

Algorithm 1 The macrokernel of a high-performance GEMM() implementation



procedure MACROKERNEL($\tilde{A}, \tilde{B}, C'$)
  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
      $C'[i : i + m_R, j : j + n_R] \mathrel{+}= \tilde{A}[i : i + m_R, :] \cdot \tilde{B}[:, j : j + n_R]$

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

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 constant 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 $\tilde{B}$ 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 $\tilde{A}$ will cause it to be resident in the $L1$ cache. Finally, some values from C' will also need locations.

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

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

. Manipulating the inequality further shows us that

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

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. It is as this point where m_R and n_R may need to be swapped.

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

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

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 $\tilde{A}$ to pass through.

2.3 Data reuse in matrix multiplication

Examining the loops in the BLIS algorithm, as was done in[6], shows that each loop causes some data from the computation to be reused, that is, read or written it its entirety during each iteration of the loop. For example, the micro-kernel reuses the small block of C' that is stored in registers

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**TODO cite this?**. Packing into $\tilde{B}$ and $\tilde{A}$ requires time that is not usable for computation. Therefore, reusing the packed buffers many times during the computation will lower the proportion of computation time spent on packing, increasing performance.

To improve reuse of $\tilde{B}$, we need the 1st loop around the macrokernel, which packs into $\tilde{A}$, to run many times, since each of those iterations covers computations that read the entirety of $\tilde{B}$. That is, we need m to be large, since small values of m will hurt performance. Similarly, large values of n allow the 2nd loop around the microkernel to execute many times, thus improving reuse of $\tilde{A}$.

The only loops that iterate over k are those that pack $\tilde{B}$ (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 loads anything into cache. The microkernel also only reuses data in registers, without reusing cache.

This reasoning shows that the BLIS algorithm achieves its best performance when m and n are large.

3 Algorithm

For our discussion of GEMM3(), we well be considering the operation $D += ABC$, where A is $m \times k$, B is $k \times l$ and C is $l \times n$.

To derive our algorithm for GEMM3(), we began by considering the computation of GEMM(A, BC) with the BLIS algorithm and attempting to find ways to not need all of BC at once. We observed that elements of BC are not needed until that matrix has been partitioned both horizontaly and vertically, creating a block of BC thas is packed into $L3$ cache. This led us to consider how we might compute a $k_C \times l$ by $l \times n_C$ block of BC and pack it into $\tilde{BC}$ efficiently.

Looking again at the BLIS algorithm, but this time considering the computation of our block, we noticed that the outer loop (over n) could be removed for this "inner" computation, since our matrices could never be larger than n_C. Simply nesting two copies of the BLIS algorithm, however, would cause issues, even with one loop removed. One such issue was that there was no guarantee that the inner algorithm (which computed blocks of BC) would place its results into cache. Another difficulty came from the fact that we would still need to pack blocks of BC into $\tilde{BC}$, incurring a cost in memory operations we hoped to avoid.

Fortunately, these problems were resolvable. The microkernel could be made to write in the packed format, eliminating a large number of memory operations, by informing it that the output matrix was row-major with rows of width n_R. To solve the cache-residency issue, we observed that, if we used an n_C value that was half of the one from BLIS, both a packed block of C and the output block of BC could reside in the $L3$ cache simultaneously. This change 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.

One possible issue with this approach is that the inner computation is a panel-matrix multiply, that is, the m dimension is small, and $\tilde{C}$ would see little reuse. This likely did have a performance impact, though it turned out to be small in practice.

There was one other adjustments to the BLIS algorithm's constants than needed to be made to enable the fusion to proceed efficiently. In some cases, the BLIS microkernel's loops have a final iteration that considers fewer than m_R rows (or n_R columns) because not all matrices have sizes divisible by m_R and n_R. Computation of this fridge iteration can have a performance impact, as special processing is required to account for the unusual sizes. In the fused BLIS algorithm, the value of k_C (which is used for partitioning in the outer algorithm) becomes the m dimension of the problem computed by the inner algorithm. If k_C is not divisible by m_R, we must (at the cost of slightly less optimal cache usage) reduce it slightly so that it is a multiple of m_R, thus ensuring we do not incur performance penalties every time we compute $\tilde{BC}$. However, l_C, which is the inner algorithm's version of k_C, can be kept at the BLIS value, as these considerations are not relevant there.

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

It should be noted that this algorithm computes D += $A(BC)$. In some cases, it is much more efficient (on account of the dimensions of the inputs) to compute D += $(AB)C$ instead. Attempting to derive an algorithm for that problem directly does not work, as the block $\tilde{AB}$ in $L2$ cache would need to be recomputed multiple times for each iteration of the loop over n, and would also not see much reuse. Fortunately, we can transform problems of this form to the equivalent problem D^T += $C^T(B^TA^T)$. This transformation can be performed at effectively no cost by re-interpreting the memory the matrices are stored in

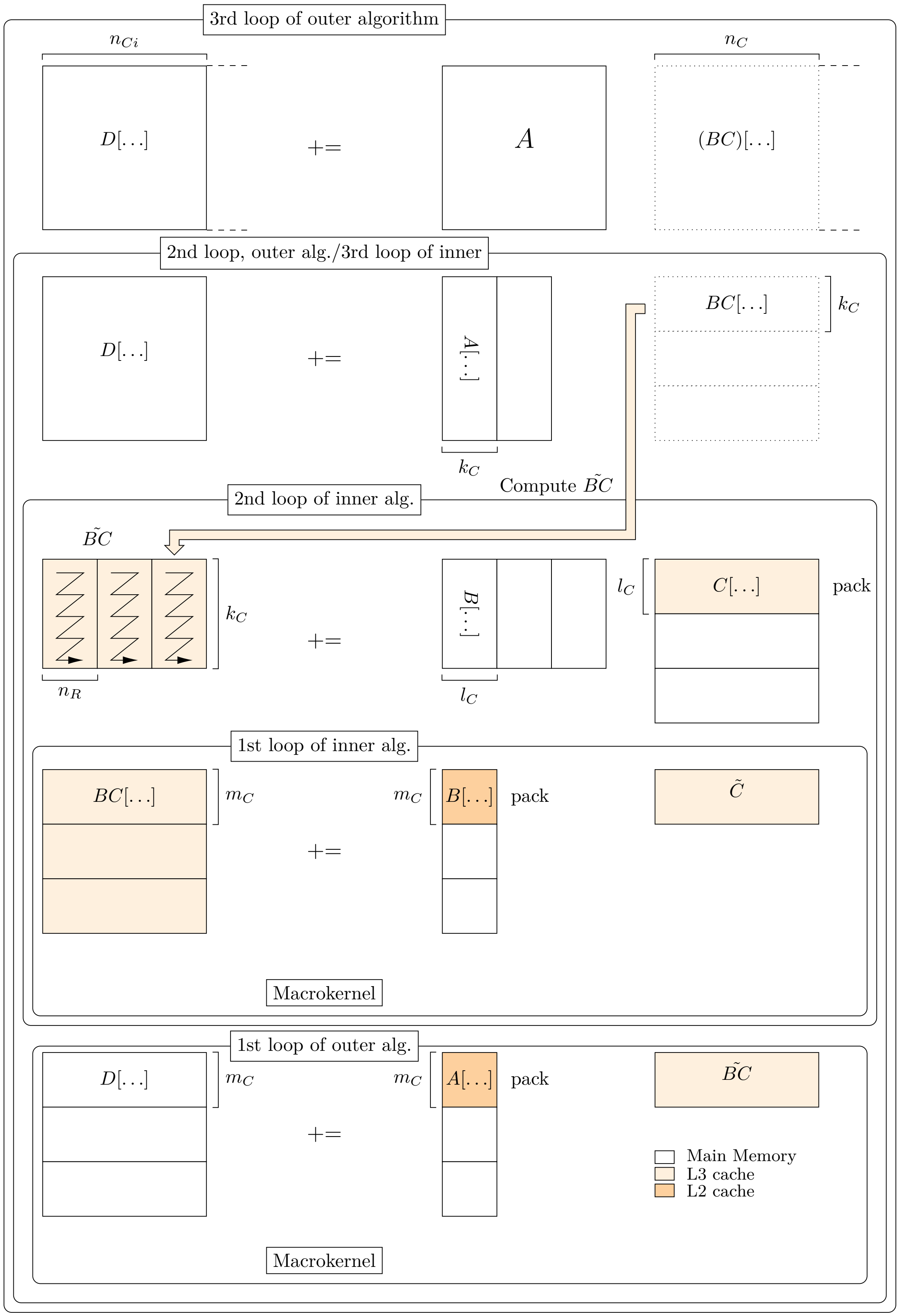


Figure 1: Illustration of algorithm for GEMM3()

Algorithm 3 Algorithm for GEMM3()

---

procedure GEMM3(A, B, C, D)
    for $j \leftarrow 0, n_C, \ldots$ **to** n **do**
        for $p \leftarrow 0, k_C, \ldots$ **to** k **do**
            for $q \leftarrow 0, l_C, \ldots$ **to** l **do**
                pack $C[q : q + l_C, j : j + n_C] \rightarrow \tilde{C}$
                for $i \leftarrow 0, m_C, \ldots$ **to** k_C **do**
                    pack $B[i : i + m_C, q : q + l_C] \rightarrow \tilde{B}$
                    MACROKERNEL($\tilde{B}, \tilde{C}, \tilde{BC}$) ▷ writes in packed form
            for $i \leftarrow 0, m_C, \ldots$ **to** m **do**
                pack $A[m : m + m_C, p : p + k_C] \rightarrow \tilde{A}$
                MACROKERNEL($\tilde{A}, \tilde{BC}, D[i : i + m_C, j : j + n_C]$)

(treating a row-major matrix as column-major and vice-versa).

4 Experiments and Results

To test our algorithm's performance, we implemented both it and the BLIS algorithm in the Multilevel Optimization for Matrix Multiply Sandbox (MOMMS), which wes developed for[9]. 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.

We modified the framework to allow for "subcomputations" which are virtual objects that represent a GEMM() operation. Subcomputations pass through partitionings to their respective input matrices, and then can be forced to fill a given output matrix. This both allowed us to cleanly express the GEMM3() algorithm, and allowed the typical $T = BC; D \mathrel{+}= AT$ algorithm for GEMM3() to be expressed as the BLIS algorithm where one operand is the BLIS algorithm as a subcompuation, with that operand being forced immediately.

We tested the performance, in gigaflops per second, of both our algorithm and the typical approach.

The experiments were run on a public lab machine at the University of Texas at Austin.

	GEMM3()	BLIS algorithm
m_C	72	72
k_C	252	256
l_C	256	
n_C	2040	2080

Table 1: Blocking constants for Haswell

The machine had an Intel Xeon E3-1270 v3 CPU, which runs at 3.5 GHz and implements the Haswell microarchitecture. Thi The experimental system has 15 GB of RAM, 8 MB of $L3$ cache, 256 KB of $L2$ cache (which is an 8-way set-associative) and $32KM$ of $L1$ cache (per core, 8-way). The blocking constants that arise from this system can be found in Table 1

Our experiments were run both on square matrices ($m = n = k = l$) and with each coordinate fixed at 9 in turn, to evaluate the performance of these algorithms on multiple input shapes. We sampled the performance at multiples of sixteen from 16 to 4912 to test the suitability of our approach on many input sizes. All the inputs were stored column-major, while the temporaries and outputs were stored row-major, 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.

The results of these experiments can be found in Figure 2 for square matrices, and Figure 3 for the rectangular inputs. 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.

The computed additional (excluding inputs and outputs) memory consumption of both algorithms for square matrices of various input sizes is shown in Figure 4, demonstrating our approach's much lower memory usage.

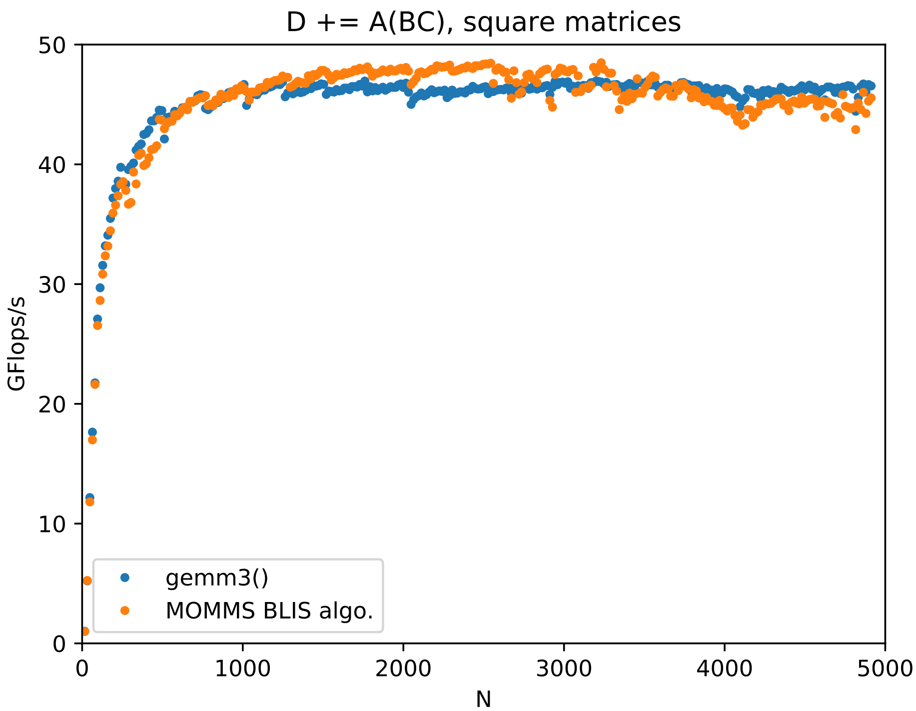


Figure 2: Performance of our GEMM3() implementation compared to a pairr of GEMM() calls, square matrices

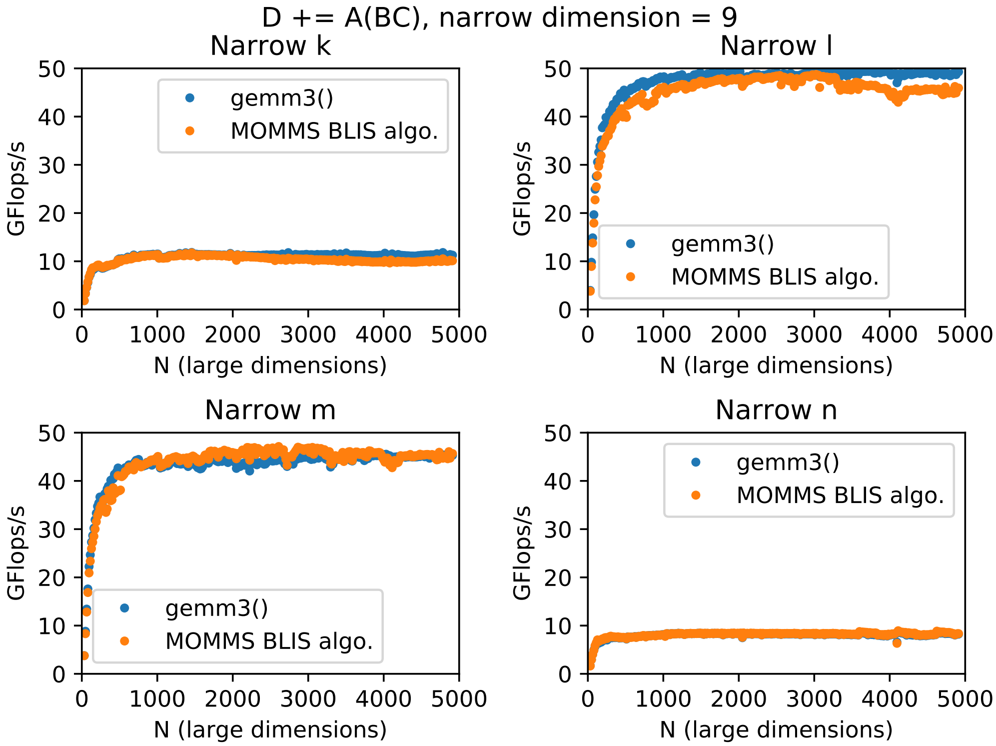


Figure 3: Performance of our GEMM3() implementation compared to a pairr of GEMM() calls, rectangular matrices

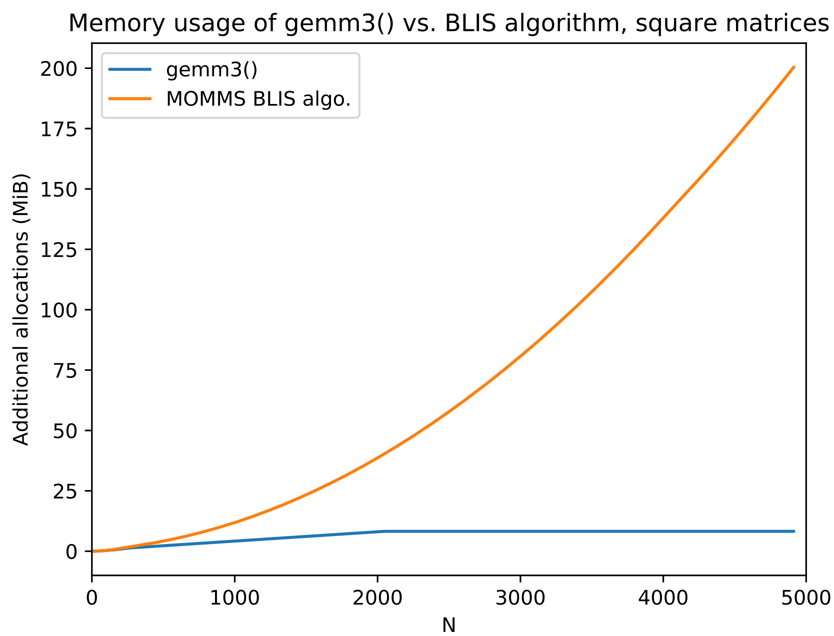


Figure 4: Memory usage of our GEMM3() implementation compared to a pairr of GEMM() calls

5 Discussion

From the figures above, 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, we can observe that, for most of the 1000s and 2000s, the pair of GEMM() calls has somewhat increased performance, but that the GEMM()3 algorithm is more perform ant afterwards. The higher performance for the GEMM() calls arises from the small lack of reuse of $\tilde{C}$ 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 BC. However, around when the temporary grows to 64 MB, the memory cost of writing it to memory becomes large enough that the GEMM3() algorithm begins to have higher performance. This shows that our approach, in addition to the memory savings, provides performance benefits for large matrix multiplications.

The rectangular results also require examination. We can see that, when l is small, the GEMM3() algorithm performs better on all inputs, as the memory-writing overhead in the BLIS algorithm remains, while the advantege of reuse in computing BC is reduced (because the k dimension is very narrow). When m is narrow, the costs of low $L3$ reuse are imposed at least once on both algorithms, decreasing the performance of both.

When n is narrow, we have effectively converted the problem to several matrix-vector multiplications, which cannot be performant for 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 operatons dominate this computation.

These results demonstrate the general suitability of our algorithm for GEMM3().

6 Things that still might need doing

- Portability (run on KNL)
- Clean experiments (public lab machines are kinda a bad place)
- Parallel case (the square-case performance behavior we have is still there, just much more exaggerated on multiple cores)
- Buffing the $D^T \mathrel{+}= C^T(B^T A^T)$ result or explaining it so the figure isn't too embarrasing
- Minor lingering performance things, maybe
- Make this longer to keep the writing flag people happy

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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