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

Krzysztof A. Drewniak, Tyler M. Smith, Robert van de Gejin

February 20, 2018

1 Introduction

High-performance matrix multiplication is an important primitive for high-performance computing. Significant research effort, both academic (**TODO cite a few**) and commercial has gone in to optimizing this operation. The typical interface for such multiplication is the function GEMM() from the Basic Linear Algebra Subprograms (BLAS) specification, which computes $C \coloneqq \beta C + \alpha AB$ for matrices A, B, and C and scalars α and β, optionally taking the transpose of one or both of the input operands.

In several applications, such as **TODO, I think there's a chemistry thing Devin would know about** and **TODO another application**, operations of the form $D \coloneqq \beta D + \alpha ABC$ occur. These often come from needing to recompose a matrix after it has been decomposed and transformed **TODO check this, give examples**. To perform these types of operations (which we'll summarize as $D \mathrel{+}= ABC$) performantly using GEMM(), the programmer must allocate a temporary buffer T and perform $T = BC; D \mathrel{+}= AT$ (or $T = AB; D \mathrel{+}= TC$). This has two drawbacks: the first is that T is often a rather large matrix, which would require significant amounts of memory to store. In addition, reading and writing T incurs a performance cost associated with reading and writing main memory,

which is non-negligible for sufficiently large inputs.

To combat these issue, we have developed an algorithm for GEMM3(), that is, the computation of $D \mathrel{+}= ABC$, 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.

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.

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[2] or by autotuning**TODO cite someone. ATLAS?**.

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.

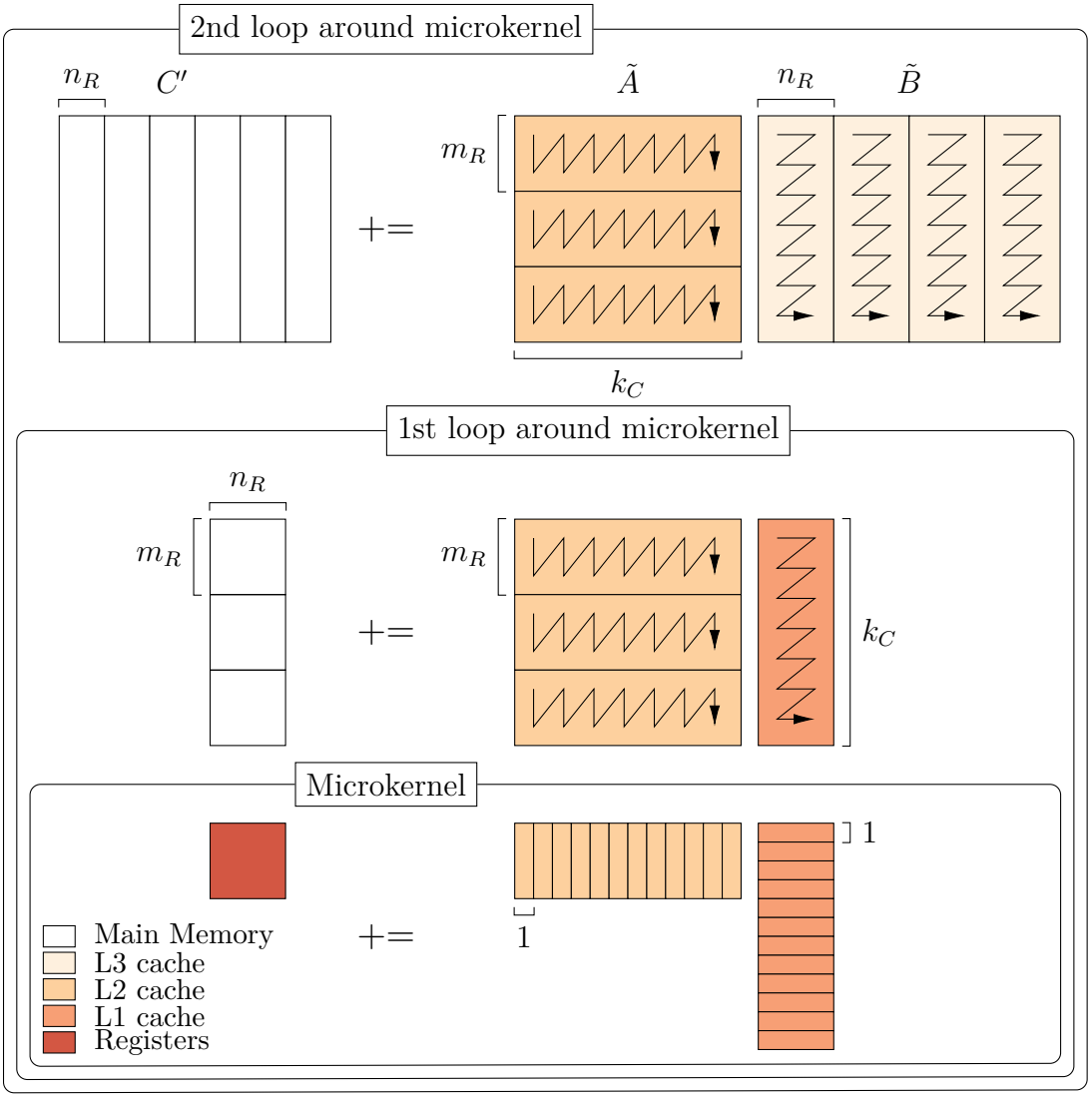
We will use $\tilde{A}$ and $\tilde{B}$ to denote the packed blocks of A and B, respectively, and C' to denote the corresponding block of C. The loops that perform $C' \mathrel{+}= \tilde{A}\tilde{B}$ form the macrokernel, which is depicted as Algorithm 1. Since the loops that perform $C' \mathrel{+}= \tilde{A}\tilde{B}$ 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[1]. One such implementation(**algorithm?**), which we have based our work on, is BLIS**TODO cite**, 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

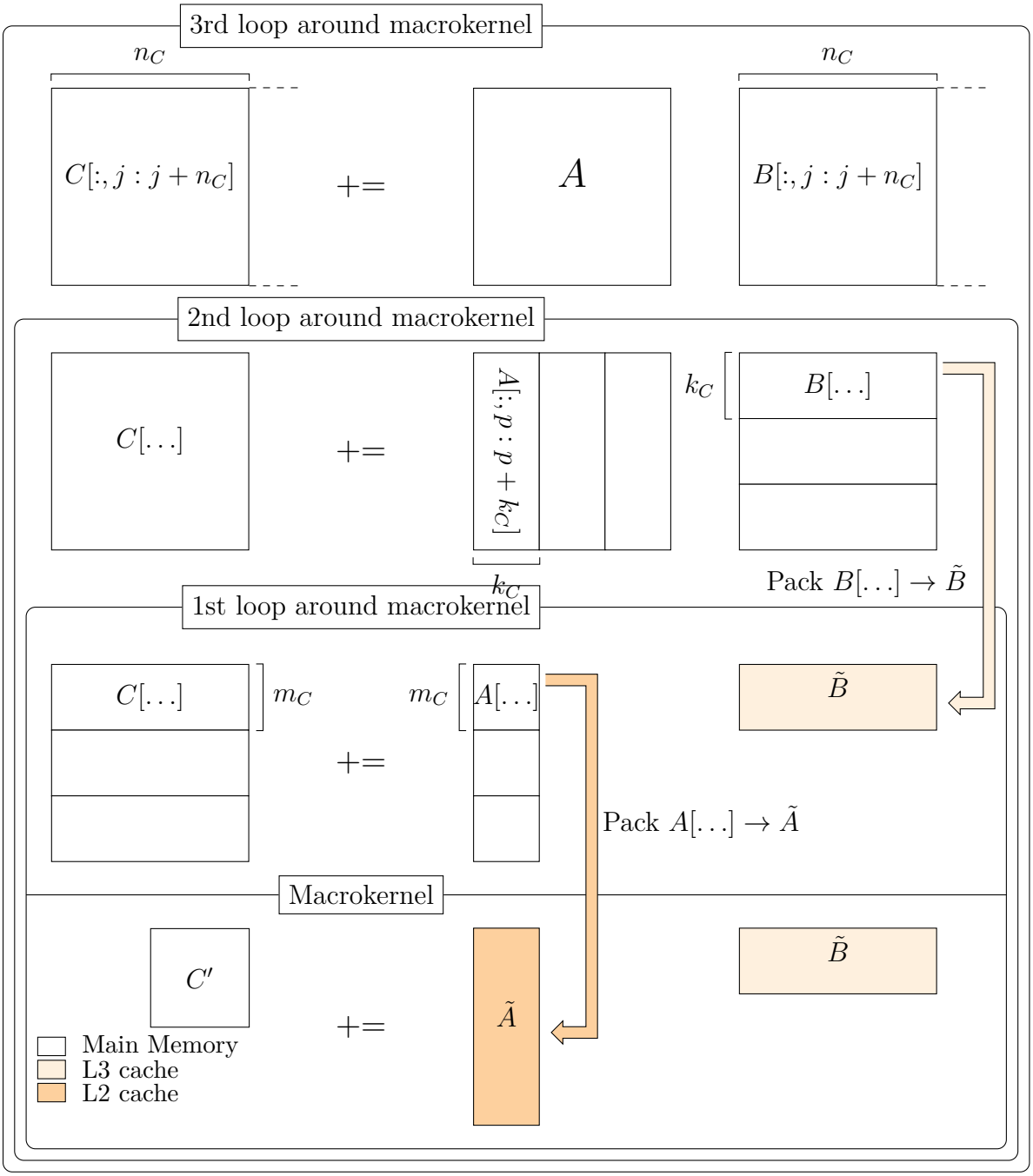
The constants m_R, n_R, k_C, m_C, and n_C that appear in the BLIS algorithm are computed using an analytical model from [2]. We will summarize the process of deriving some of these constants here. The values of m_R and n_R determine the structure of the microkernel, and are derived from the width of vector registers on a given CPU along with the latency and throughput of fused-multiply-add (FMA) instructions. Since we are reusing existing

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($\tilde{A}, \tilde{B}, C[i : i + m_C, j : j + n_C]$)

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 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[2], 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 $D \mathrel{+}= ABC$, where A is $m \times k$, B is $k \times l$ and C is $l \times n$.

Algorithm 3 Algorithm for GEMM3()

```
procedure GEMM3(A, B, C, D)
    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 C[q : q + l_C, j : j + n_C] → C̃
                for i ← 0, m_C, ... to k_C do
                    pack B[i : i + m_C, q : q + l_C] → B̃
                    MACROKERNEL(B̃, C̃, B̃C̃)          ▷ writes in packed form
            for i ← 0, m_C, ... to m do
                pack A[m : m + m_C, p : p + k_C] → Ã
                MACROKERNEL(Ã, B̃C̃, D[i : i + m_C, j : j + n_C])
```

Our algorithm computes $D \mathrel{+}= A \cdot (BC)$ by replacing the packing of $\tilde{BC}$ in the BLIS algorithm for this problem by another copy of the BLIS algorith, which computes $\tilde{BC} = B[\ldots]C[\ldots]$. The second copy, known as the inner algorithm, has its outermost loop (over n) removed, as the output must be at most $k_C \times n_C$. This algorithm is illustrated in Figure

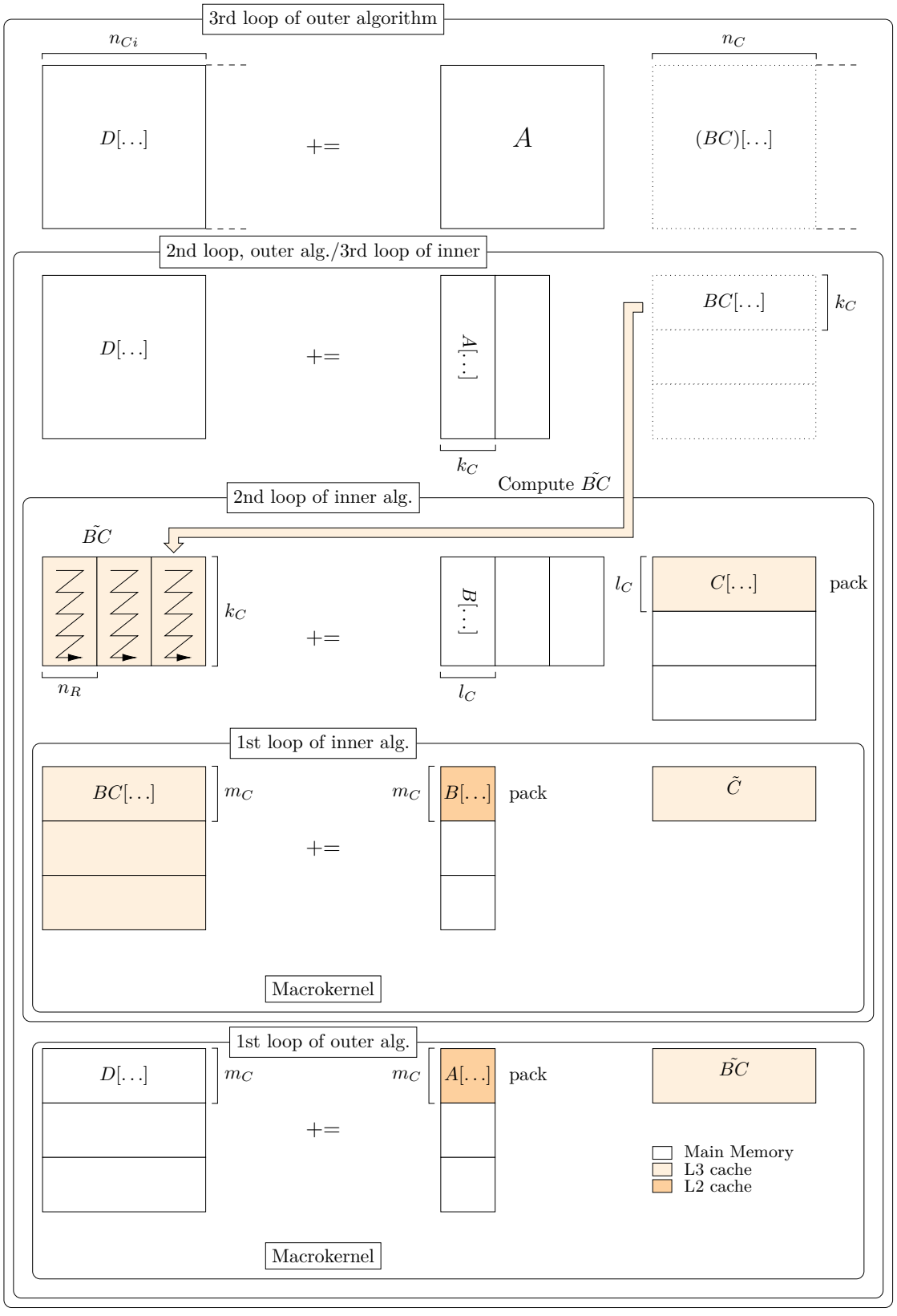


Figure 1: Illustration of algorithm for GEMM3()

1 and the pseudocode is Algorithm 3.

The constants (m_C, k_C, l_C and n_C) in the GEMM3() algorithm are the BLIS constants with some alterations. First, it may be necessary for k_C (which appears in the outer algorithm) to be somewhat different from l_C in the inner algorithm. This is because, to achieve performance, we need k_C (which is the m dimension for the inner algorithm) to be evenly divisible by m_R, to limit the number of cases where the microkernel cannot be used due to overhang. To achieve this goal, l_C is kept at the model's value for k_C in the BLIS algorithm, while the k_C for GEMM3() is decreased just enough to make it divisible for m_C.

Second, we want both $\tilde{C}$ and $\tilde{BC}$ to be resident in the $L3$ cache. The easiest way to achieve this is to set n_C for GEMM3() to be half of what it is in the BLIS algorithm. This still retains the property that $n_C \gg m_C, k_C, l_C$ while allowing all the buffers that should be in L3 to coexist.

This algorithm specifically computes $D \mathrel{+}= A(BC)$. In some cases, it may be a better idea to compute $D \mathrel{+}= (AB)C$. Formulating another nested-BLIS algorithm for that problem will not work, as the block being computed is the small $m_C \times k_C$ block, which will have to recomputed many times due to the iteration over k that surrounds it. However, we can apply algorithm to problems of this form by observing that $D \mathrel{+}= (AB)C$ can be transformed to $D^T \mathrel{+}= C^T(B^T A^T)$. This transformation can be done at almost no cost by interpreting column-major inputs as row-major or vice-versa.

In the GEMM3() algorithm we present, the inner algorithm writes in the packed format. This can easily be achieved by informing the microkernel that "C" is a row-major matrix with leading dimension n_R.

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

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

Table 1: Blocking constants for Haswell

for[3]. 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. 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,

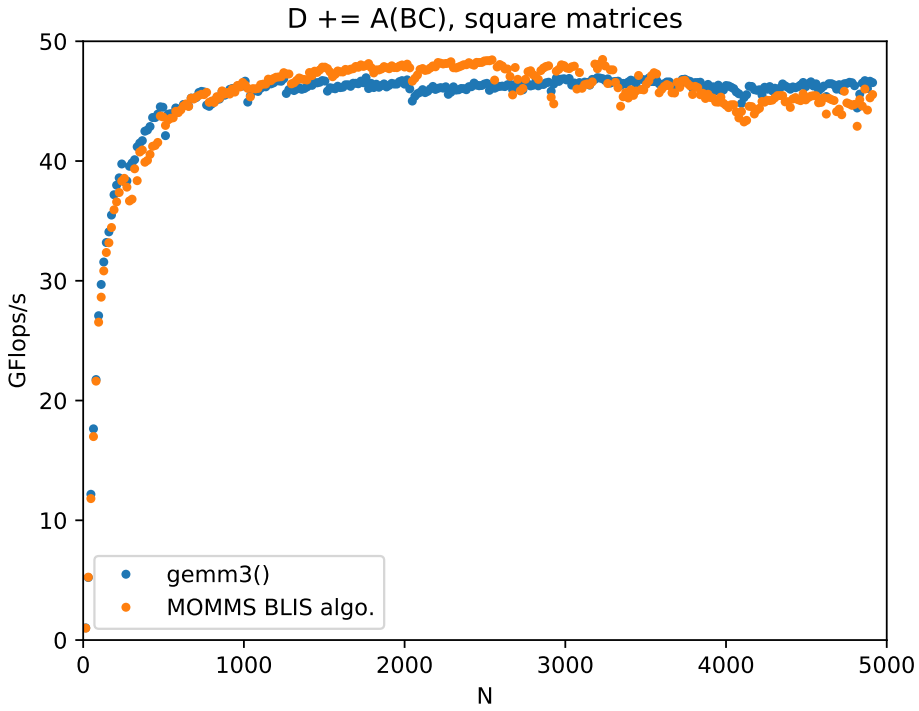


Figure 2: Performance of our GEMM3() implementation compared to a pairr of GEMM() calls, square matrices

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.

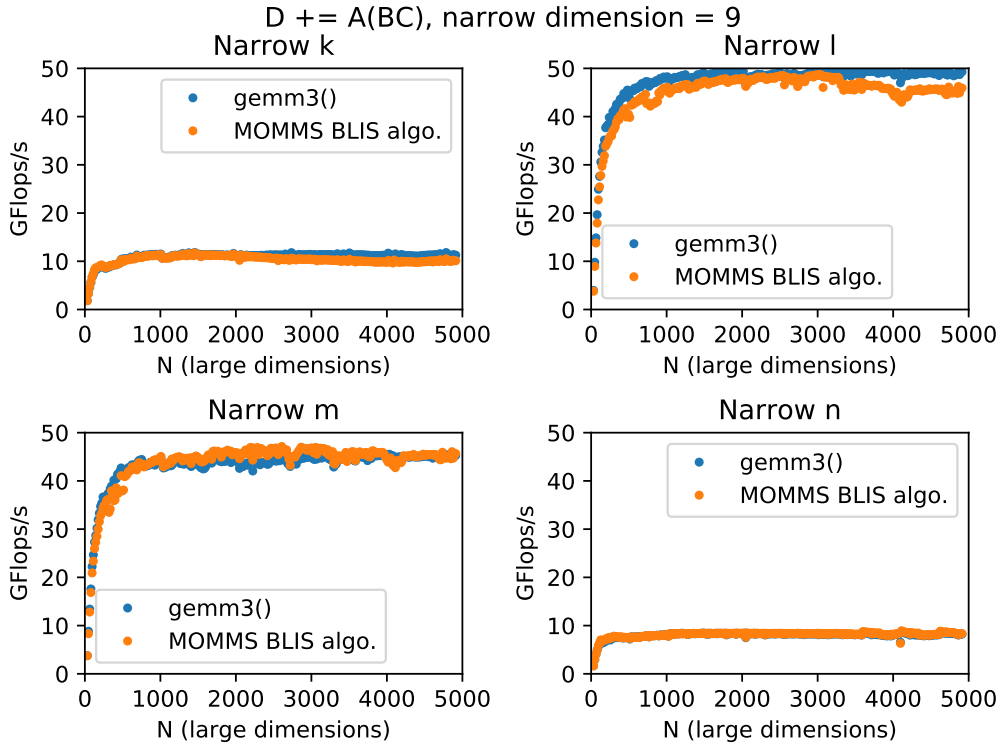


Figure 3: Performance of our GEMM3() implementation compared to a pairr of GEMM() calls, rectangular matrices

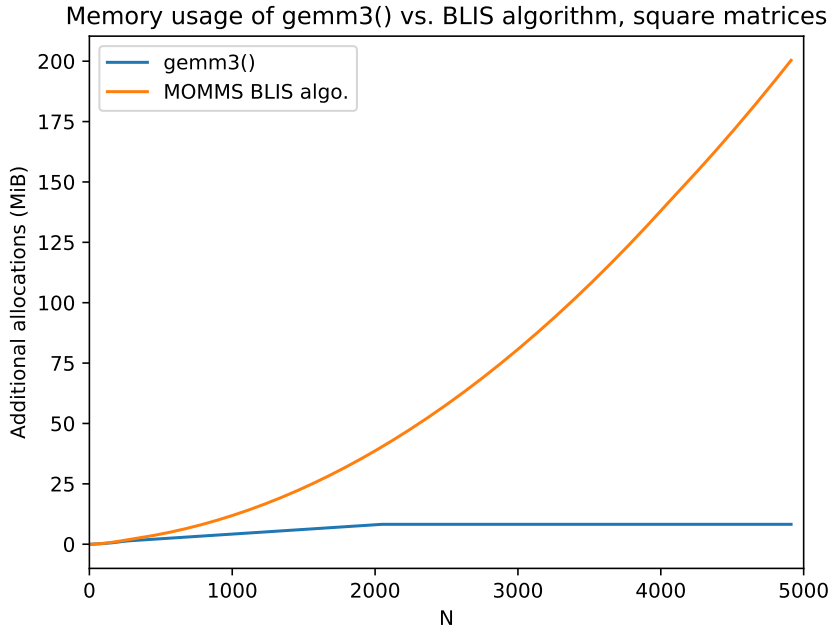


Figure 4: Memory usage of our GEMM3() implementation compared to a pairr of GEMM() calls

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] 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.

[2] 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.

[3] 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.