gemm3(): Constant-workspace high-performance multiplication of three matrices

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

February 9, 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. To perform this (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.

To combat this issue, we have developed an algorithm for GEMM3(), that is, the computation of $D \mathrel{+}= ABC$, that does not require the entire intermediate product to be stored at one time. This algorithm exploits the blocked structure of modern matrix multiplication algorithm to only compute a cache-sized block of (BC) at a time, and uses a recent algorithm that meets a theoretical lower-bound on memory I/O when the output matrix is a square that fits in the highest level of cache[4]. It has attained performance gains of 5–6% (in GFlops/s) over a pair of GEMM() calls. **TODO, more intro?**

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. The multiple reductions are required to effectively utilize all levels of the cache.

There are two specialized primitives that further contribute to efficient GEMM(). The first is the *microkernel*, 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. 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. The data within the panels must be arranged so that the microkernel's memory reads will operate contiguously. This is generally 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.

To fill the cache, several such panels from A and B are placed contiguously in memory in the process of *packing*. Since caches have a fixed size, the amount of data that can be packed at any given time is limited. Specifically, we can only pack an $m_C \times K_C$ block of A and a $k_C \times n_C$ block of B, where m_C, k_C and n_C are constants that depend on the cache structure of the CPU, as well as m_R and n_R.

We will use $\tilde{A}$ and $\tilde{B}$ to represent the matrices formed by packing data from regions of A and B, respectively, and C' to represent the memory region updated by $\tilde{A}\tilde{B}$ at any given time. 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**TODO cite him?**. One such implementation(**algorithm?**) is BLIS**TODO cite**, which is shown as Algorithm 2.. This algorithm brinks elements of B into the L_3 (and later L_1) cache, while storing elements of A in L_2.

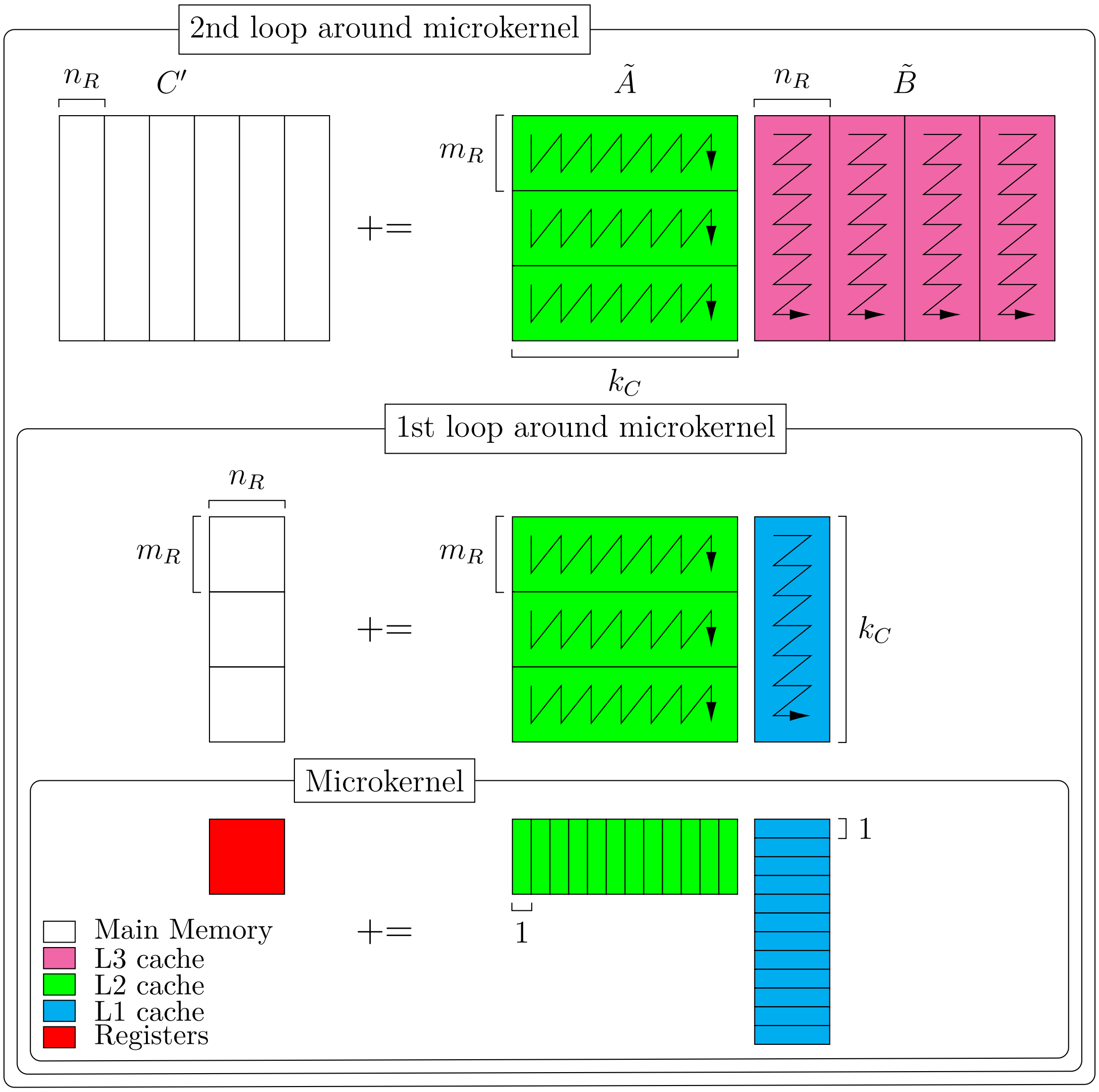
2.2 Cache-efficient families of gemm() algorithms

The BLIS algorithm will, during its operation, re-use the same panel of $\tilde{A}$ multiple times while executing a loop over panels of $\tilde{B}$. However, the packed date in $\tilde{B}$ is only used once. This both results in a suboptimal algorithm and prevents us from using the BLIS algorithm for GEMM()3, as it will not be possible to reuse any cache-sized results that are large enough to be useful (that is, those that fit into L_3).

However, a family of efficient GEMM() algorithms that supports a larger amount of cache reuse has been identified[1, 3]. These algorithms insert an additional loop around the macrokernel to facilitate reuse both in the L_3 and L_2 caches. The algorithms in this family differ in which matrices are resident in each level of cache.

Two of the algorithms in this family are relevant to our work. The first is known as B_3A_2, (Algorithm 3) which keeps elements of B and A resident in L_3 and L_2 cache, respectively.

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



procedure MACROKERNEL($\tilde{A}, \tilde{B}, C'$)

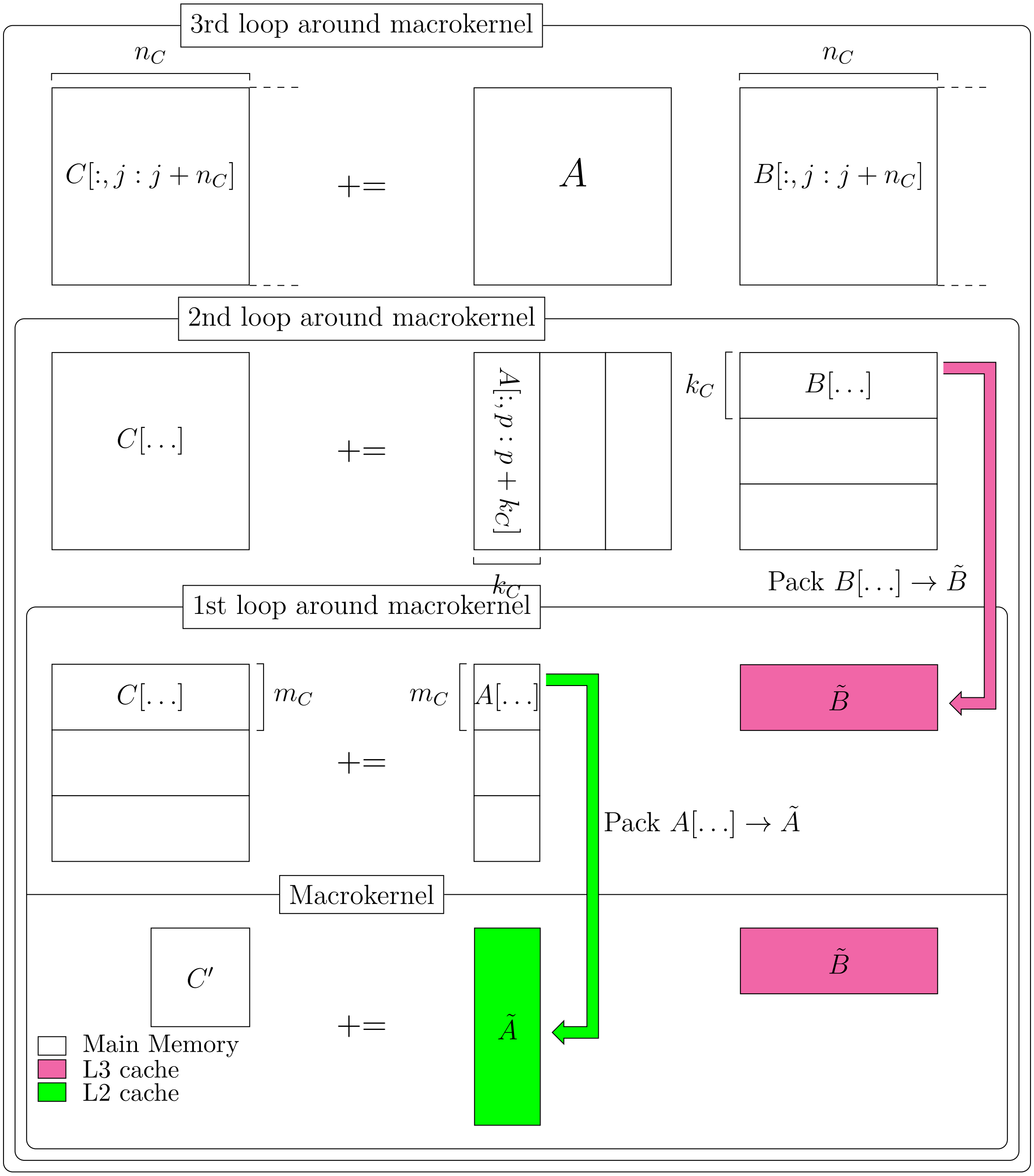
  for $j \leftarrow 0, n_R, \ldots$ **to** n_C **do**

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

The second, which meets a theoretical lower bound for I/O cost when the output has size $\sqrt{S_3} \times \sqrt{S_3}$, where S_3 is the size of the L_3 cache[4], is C_3A_2 (Algorithm 4). **TODO confirm I've got the reuse properties on that second one right**

Even though the constants in these algorithms share names with those in the BLIS algorithm, their values may be different.

Algorithm 3 B_3A_2 algorithm for GEMM()

```
procedure B3A2_GEMM(A, B, C)
    for j ← 0, n_C, ... to n do
        for p ← 0, k_C, ... to k do
            pack B[p : p + k_C, j : j + n_C] → B̃
            for i ← 0, m_C, ... to m do
                for p′ ← p, p + k_C2, ... to p + k_C do
                    pack A[m : m + m_C, p′ : p′ + k_C2] → Ã
                    MACROKERNEL(Ã, B̃, C[i : i + m_C, j : j + n_C])
```

Algorithm 4 C_3A_2 algorithm for GEMM()

```
procedure C3A2_GEMM(A, B, C)
    for j ← 0, n_C, ... to n do
        for i ← 0, m_C, ... to m do
            for p ← 0, k_C, ... to k do
                pack B[p : p + k_C, j : j + n_C] → B̃
                for i′ ← i, i + m_C2, ... to i + m_C do
                    pack A[i′ : i′ + m_C2, p : p + k_C] → Ã
                    MACROKERNEL(Ã, B̃, C[i′ : i′ + m_C2, j : j + n_C])
```

3 Algorithm and Method

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 $l \times n$.

Our algorithm fuses together the B_3A_2 and C_3A_2 algorithms. To do this, we replace the "pack into $\tilde{B}$" step of Algorithm 3 (the "outer" algorithm) with a call to a variant of Algorithm 4 (the "inner" algorithm). For our implementation, the inner algorithm has had its outer two loops removed, and has had its macrokernel adjusted to output in the packed format needed by the outer algorithm. The only change made to the outer algorithm was to adjust n_C and k_C so the inner algorithm would have an approximation to $\sqrt{S_3}$ by $\sqrt{S_3}$ for its output. The complete algorithm is Algorithm 5.

The outer and inner algorithm were chosen over other elements of their family because, if they are run in the fused configuration, the output of the inner algorithm will be where the outer algorithm expects it — in L_3 cache.

Algorithm 5 Algorithm for GEMM()3

TODO picture

```
procedure GEMM3(A, B, C, D)
    for j_o ← 0, n_Ci, ... to n do
        for p_o ← 0, m_Ci, ... to k do
            for p_i ← 0, k_Ci, ... to l do
                pack C[p_i : p_i + k_Ci, j_o : j_o + n_Ci] → B̃_i
                for i_i ← p_0, p_0 + m_Ci2, ... to p_0 + m_Ci do
                    pack B[i_i : i_i + m_Ci2, p_i : p_i + k_Ci] → Ã_i
                    MACROKERNEL(Ã_i, B̃_i, B̃_o)                ▷ writes in packed form
            for i ← 0, m_Co, ... to m do
                for p′ ← p, p + k_C2, ... to p + k_C do
                    pack A[m : m + m_C, p′ : p′ + k_C2] → Ã_o
                    MACROKERNEL(Ã_o, B̃_o, D[i : i + m_C, j : j + n_C])
```

It should be noted that the given algorithm computes $D \mathrel{+}= A(BC)$. Our attempt to derive an algorithm directly for $D \mathrel{+}= (AB)C$ using an algorithm that keeps A resident in L_3 yielded poor results. However, we can observe that, to compute $D \mathrel{+}= (AB)C$, we can compute $D^T = C^T(B^T A^T)$, and that interpreting column-major matrices as row-major (or vice-versa) would be equivalent to those transposes. This means that we can use the same algorithm for both possible parenthizations of $D \mathrel{+}= ABC$ without significant performance costs.

3.1 Constants

TODO get unblocked on this

3.2 Testing methodology

We implemented this algorithm in the **TODO expand acronym** (MOMMS) framework, which was developed to rapidly implement multiple algorithms from the family discussed in Subsection 2.2[3]. This framework, which is implemented in the Rust programming language, was first used to implement the BLIS algorithm as a baseline for comparisons. The MOMMS implementation of the BLIS algorithm had comprable performance to the original.

(Things to discuss)

- Paralellization
- Choice of machine(s)
- Haswell and KNL (if we do a KNL thing)
- Probably some other stuff

4 Results

TODO fix the experiments first

5 Discussion

TODO have better results first

References

[1] John A. Gunnels et al. "A family of high-performance matrix multiplication algorithms". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 3732 LNCS (2006), pp. 256–265. ISSN: 03029743. DOI: `10.1007/11558958_30`.

[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.

[4] Tyler Michael Smith and Robert A. van de Geijn. "Pushing the Bounds for Matrix-Matrix Multiplication". In: (2017), pp. 1–11. arXiv: 1702.02017. URL: http://arxiv.org/abs/1702.02017.