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

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

February 8, 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 $C := \beta C + \alpha AB$ for matrices $A$, $B$, and $C$ and scalars $\alpha$ and $\beta$, 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 $D := \beta D + \alpha ABC$ 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. 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 `gemm3()`, it is important to review the techniques for the operation $C \coloneqq \alpha AB + \beta C$, that is, `gemm()`. For simplicity, we'll present the operation as $C \mathrel{+}= AB$ for simplicity. A naive implementation would proceed as follows (where $A$ is $m$ by $k$, $B$ is $k$ by $n$, and $c$ is $m$ by $n$) This

**Algorithm 1** Naive implementation of `gemm()`

1: **procedure** GEMM($A, B, C$)
2:     **for** $i \leftarrow 0$ up to $m$ **do**
3:         **for** $j \leftarrow 0$ up to $n$ **do**
4:             **for** $c \leftarrow 0$ up to $k$ **do**
5:                 $C_{i,j} \leftarrow C_{i,j} + A_{i,c}B_{c,j}$

algorithm has serious performance issues in that it accesses the memory of one of the operands ($A$ for row-major storage and $B$ for column-major) at a stride of $k$, which is almost always a number that makes it impossible for the processor to stream both matrices' values into memory through prefectching or to vectorize the memory accesses, which would allow multiple elements of $C$ to be computed simultaneously on the same CPU core. Therefore, it is effectively never used in practice except as a verification tool for more efficient algorithms.

Many of the high-performance `gemm()` algorithms in use today are based on the approach of Goto**TODO cite** These algorithms massively improves performance by taking advantage of the multi-level cache present on modern CPU architectures. They operate by reducing the `gemm()` to a series of sub-problems that are sized such that their inputs and/or outputs fit into the levels of the system's cache, and additionally by rearranging the inputs to those subploblems into a form that can be streamed from cache by the *microkernel*, a highly-optimized inner loop.

One commonly-used algorithm of this type is the BLIS algorithm,