Pushing the Bounds for Matrix-Matrix Multiplication

FLAME Working Note #83

Tyler Michael Smith and Robert A. van de Geijn

Department of Computer Science,
The University of Texas at Austin

February 3, 2017

Abstract

A tight lower bound for required I/O when computing a matrix-matrix multiplication on a processor with two layers of memory is established. Prior work obtained weaker lower bounds by reasoning about the number of *phases* needed to perform $C := AB$, where each phase is a series of operations involving S reads and writes to and from fast memory, and S is the size of fast memory. A lower bound on the number of phases was then determined by obtaining an upper bound on the number of scalar multiplications performed per phase. This paper follows the same high level approach, but improves the lower bound by considering $C := AB + C$ instead of $C := AB$, and obtains the maximum number of scalar fused multiply-adds (FMAs) per phase instead of scalar additions. Key to obtaining the new result is the decoupling of the per-phase I/O from the size of fast memory. The new lower bound is $2mnk/\sqrt{S} - 2S$. The constant for the leading term is an improvement of a factor $4\sqrt{2}$. A theoretical algorithm that attains the lower bound is given, and how the state-of-the-art Goto's algorithm also in some sense meets the lower bound is discussed.

1 Introduction

Matrix-matrix multiplication[1] (MMM) is an important practical operation from which many applications demand high performance. A limiting factor on what fraction of theoretical peak this operation can attain is the input-output (I/O) operations, data movements between memory layers, that are incurred. For this reason, lower bounds on the I/O requirements are of great interest, especially as the ratio between the speed of I/O and floating point computation continues to deteriorate.

Deriving lower bounds starts with the assumption that a processor has two layers of memory hierarchy, a small *fast* memory and large *slow* memory. The fast and slow memory could represent the cache(s) and main memory of a processor, respectively, or main memory and disk. Practical implementations minimize the movement of data between these (and more) layers.

Hong and Kung [4] introduced what they called the red-blue pebble game model for a machine with two layers of memory. A limited number of blue pebbles represented fast memory while an unlimited number of red pebbles represented slow memory. By reasoning how at different times blue and red pebbles could

[1] In this paper, we only consider *conventional* matrix-matrix multiplication where $C := AB$ with matrices C, A, and B of respective sizes $m \times n$, $m \times k$, and $k \times n$, require mnk scalar multiplies and mnk scalar adds.

be associated with subsets of data, a lower bound of $\Omega(mnk/\sqrt{S})$ for MMM was obtained, where S is the size of the fast memory (in matrix elements). In 1995, Savage [9] extended the red-blue pebble game to an arbitrary number of layers, and showed that the lower bound from [4] applies at every layer of the memory hierarchy.

In 2004, Irony et al. [5] use a very similar technique to extend the lower bounds to communication between nodes of a distributed memory parallel computer. Importantly, they provide a constant for the leading term of the I/O lower bound, $mnk/(2\sqrt{2}\sqrt{S})$. More recently, in 2014, Ballard et al. [1] generalized the techniques in [4] and [5] so that they can be applied to many operations in numerical linear algebra, not only matrix multiplication. Since the publication of the first result in 1981, the holy grail has been to obtain the best (greatest) constant for the leading term in the bound.

This paper provides the first tight constant for the leading term of the lower bound for this problem. While prior papers focused on the set of multiplications that need to be performed as part of a MMM, this paper observes that in practice *fused multiply add* (FMA) operations are employed, and that in practice $C := AB + C$ (matrix-matrix multiplication and accumulation or MMA) is more representative of how matrix-matrix operations are implemented in high-performance libraries. It shows that by targeting MMA instead of MMM, a superior lower bound of $2mnk/\sqrt{S} - 2S$ can be obtained. The constant on the leading term is $4\sqrt{2}$ times greater than that of the previous lower bound. It shows how the new result for $C := AB + C$ can be translated into a lower bound of $2mnk/\sqrt{S} - 2S - \mathcal{O}(mn)$ for MMM ($C := AB$). It discusses algorithms that essentially attain this lower bound, showing that **the leading term on the lower bound is sharp**. It shows that the current best practical algorithm by Goto and van de Geijn [2] essentially attains the new lower bound. Together, this advances the understanding of the limits on performance for this operation.

2 Problem Definition

In this section, we give a formal description of MMM for which we will derive I/O lower bounds. The computation that must be performed can be described as follows: Consider $C := AB$ and let $\gamma_{i,j}$, $\alpha_{i,j}$, and $\beta_{i,j}$ equal the (i, j) elements of the respective matrices.
Then $\gamma_{i,j} := \sum_{p=0}^{k-1} \alpha_{i,p}\beta_{p,j}$. This requires mnk scalar multiplications and mnk scalar additions. Alternatively, it requires mnk scalar fused multiply-adds (FMAs).

2.1 Prior approaches

Previous work, including both Hong and Kung [4] and Irony et al. [5], focused on MMM and described computation in terms of a *directed a-cyclic graph* (DAG). In each of those papers, the DAGs have input vertices corresponding to the elements of the matrices A and B, output vertices corresponding to the elements of the result matrix C, and computation vertices corresponding to the the mnk elementary multiplications $\alpha_{i,p}\beta_{p,j}$. Each computation vertex has as inputs an element of A and an element of B, and as as an output a scalar that must be summed with others to form an element of C. These DAGs are elegant and easy to work with but importantly do not expose the costs associated with reading elements of C from slow memory.

2.2 Our approach

To achieve tight lower bounds, our problem definition does expose the costs of reading elements of C. To achieve this, we model computation in terms of FMAs: In a conventional MMA ($C := AB + C$), there are mnk scalar multiplications and mnk scalar additions. We assume that each scalar multiplication $\alpha_{i,p}\beta_{p,j}$

is paired with the scalar addition with which the result is accumulated in a single FMA that has three inputs (a variable in which contributions to $\gamma_{i,j}$ are accumulated, and the elements $\alpha_{i,p}$ and $\beta_{p,j}$) and one output (the variable in which contributions to $\gamma_{i,j}$ are accumulated).

When describing a DAG for an MMA that casts computation in terms of FMAs, each computation vertex would depend on another vertex that contributes to the same element of C. Thus the definition of such a DAG would impose a partial ordering on the computation. We wish to avoid such an ordering, which leads us to *not* describe computation in terms of a DAG.

A conventional MMA must execute mnk FMAs, and every FMA has three inputs: An element each of A, B, and C that must all be in fast memory in order for the FMA to be executed. Our proofs only reason about the input costs, and so our problem definition does not need to say anything about outputs. With a DAG problem definition, dependencies are described by arcs between vertices. Without a DAG, we instead describe the FMAs that must take place. It is sufficient to say that each FMA must be of the form $\tau_{i,j} = \delta_{i,j} + \alpha_{i,p}\beta_{p,j}$ where $\tau_{i,j}$ and $\delta_{i,j}$ each are a partial accumulations of $\gamma_{i,j}$. For simplicity, the rest of this paper uses either $\gamma_{i,j}$ or the phrase "an element of C" as a shorthand to refer to any partial result to be accumulated into that element of C, except where a distinction between $\tau_{i,j}$, $\delta_{i,j}$, and $\gamma_{i,j}$ needs to be made.

Our assumptions. In our approach we have made the additional assumption that computation is cast in terms of elementary FMAs. We will now discuss to what extent this assumption restricts the generality of our proof. For our proof, it is not necessary that the computation is performed in terms of actual FMA instructions. What is assumed is that the result of any $\alpha_{i,p}\beta_{p,j}$ is immediately added to some $\delta_{i,j}$. This is the case whenever FMA instructions are used. It is also the case whenever there is no workspace used for C. This is because if the result of $\alpha_{i,p}\beta_{p,j}$ is not immediately added to some $\delta_{i,j}$, then it must be stored somewhere for a later scalar addition, and this requires workspace.

3 Lower Bound Proof

In this section we prove that an MMA must have an I/O cost of at least $2mnk/\sqrt{S} - 2S$.

3.1 High-level strategy

In order to obtain I/O lower bounds, we will think of computation as being divided into phases, where there are exactly M loads and stores during each phase (except for the last phase). That is to say, each phase has an I/O cost of at most M. If one can prove that there must be at least N phases for any algorithm, then it follows that the algorithm must have a total I/O cost of at least $(N-1)M$. The $N-1$ comes from the fact that the last phase may have less than M loads and stores.

How many phases must there be? Since it is a conventional MMA, we know that mnk FMAs must be executed. Let F be an upper bound on the number of FMAs that can occur during a single phase. Then there must be at least $(mnk)/F$ phases. This gives an overall I/O lower bound of $((mnk)/F - 1)M$.

How do we find F? We know that the size of fast memory is S, and there are at most M loads during a single phase. This means that there are $S + M$ elements of A, B, and C that can be used as inputs to FMAs during a phase. Thus placing an upper bound on the number of FMAs that can be computed using $S + M$ elements gives an upper bound on F.

This is nearly the same strategy that was used by [4] and [5]. The important difference is that in the previous papers, the number of loads and stores to and from fast memory per phase is always equal to the

size of fast memory, S. We will show that one can achieve greater lower bounds by allowing M to be a value other than S.

3.2 Employing the Loomis-Whitney inequality

The Loomis-Whitney inequality [7] was used in [5] to determine how many elementary operations involved in an MMM can be executed with some number of elements.

Theorem 3.1 (Loomis-Whitney)**.** *Let m be the measure of an open subset $\mathcal{O}$ of Euclidean n-space, and let $m_1, ..., m_n$ be the $(n-1)$-dimensional measures of the projections of $\mathcal{O}$ on the coordinate hyper planes. Then $m^{n-1} \leq m_1 m_2 \cdots m_n$.*

To apply this theorem to our situation, let $\mathcal{O}$ represent a three dimensional set of some FMAs that occur during an MMA. Each FMA has a coordinate (i, j, p) in the m, n, and k dimensions: $\gamma_{i,j} + := \alpha_{i,p}\beta_{p,j}$. The projection of $\mathcal{O}$ in each of the m, n, and k dimensions respectively corresponds to the elements of B, A, and C that are inputs to the FMAs in $\mathcal{O}$. If F is the number of FMAs that occur during an MMA using x elements of A, y elements of B, and z elements of C then the Loomis-Whitney inequality tells us that $F^2 \leq xyz$ and hence $F \leq \sqrt{xyz}$.

There are at most $S + M$ elements that can be used as inputs to computation during a phase, because there are at most S elements of A, B, and C in fast memory at the start of the phase and at most M elements read from slow memory during the phase. Similarly there can be at most $S + M$ elements that are outputs of computation during a phase, because there are at most S elements in fast memory at the end of the phase and at most M elements written to slow memory during the phase.

The authors of [5] reason separately about x, y, and z, showing that since there can be at most $S + M$ inputs to computation during phase, $x \leq S + M$ and $y \leq S + M$. Similarly, since there can be at most $S + M$ elements written during a phase, $z \leq S + M$. Working with FMAs, we do better because we know that the x elements of A, the y elements of B, and the z elements of C must all be inputs to the same phase. Therefore, we can reason about x, y, and z all together: $x + y + z \leq S + M$.

The above can be formulated as a constrained maximization problem,

$$\text{maximize } F \text{ under the constraints } \begin{cases} F \leq \sqrt{xyz} \\ 0 \leq x, y, z \\ x + y + z \leq S + M. \end{cases}$$

Application of standard Langrange multiplier methods, detailed in Appendix A, tells us that the global maximum occurs when

$$x = y = z = \frac{S + M}{3} \quad \text{so that} \quad F = \frac{(S + M)\sqrt{S + M}}{3\sqrt{3}}.$$

3.3 A lower bound for $C := AB + C$

The upper bound on F gives us the following lower bound for the I/O cost of MMA:

$$\left(\frac{mnk}{F} - 1\right) M = \left(\frac{3\sqrt{3}}{(S + M)\sqrt{S + M}} mnk - 1\right) M.$$

In this lower bound, M is a free variable meaning that different choices for M yield different lower bounds. In [4], [5], and [1], M is always equal to S. If we also make this choice, we obtain the lower bound:

$$\frac{3\sqrt{3}}{2\sqrt{2}} \frac{mnk}{\sqrt{S}} - S.$$

which is already an improvement over previous lower bounds.

It is possible to do better still. In order to find the greatest lower bound for any M, our goal is to find the M that maximizes:

$$\max_{M>0}\left(\frac{3\sqrt{3}Mmnk}{(S+M)\sqrt{S+M}}-1\right)\approx\max_{M>0}\left(\frac{3\sqrt{3}Mmnk}{(S+M)\sqrt{S+M}}\right)$$

when m, n, and k are large. Standard maximization techniques from calculus yield that the global maximum is attained when $M=2S$ so that the I/O lower bound (when m, n, and k are large so that the $-M$ term can be ignored) becomes:

$$\left(\frac{3\sqrt{3}mnk}{(S+M)\sqrt{S+M}}-1\right)M=\left(\frac{3\sqrt{3}mnk}{3S\sqrt{3S}}-1\right)(2S)=\frac{2mnk}{\sqrt{S}}-2S.$$

3.4 Improving the lower bound for $C := AB$

Above, we found a lower bound for a different problem than was considered in previous work [4, 5], which studied $C := AB$. We will now use the lower bound for $C := AB + C$ to obtain a new lower bound for the operation $C := AB$.

Consider the MMA $C := AB + C$. It can be implemented by the sequence of operations:

$$D := AB; \quad C := D + C.$$

Let Q_{AB} be the I/O time for the first operation and Q_{D+C} be the I/O time for the second. Then since these operations implement an MMA we find that $Q_{AB}+Q_{D+C}\geq 2mnk/\sqrt{S}-2S$. It is easy to show that there exists an algorithm for $D + C$ such that its I/O cost is $\mathcal{O}(mn)$. Then:

$$Q_{AB}+\mathcal{O}(mn)\geq\frac{2mnk}{\sqrt{S}}-2S \quad \text{or, equivalently,} \quad Q_{AB}\geq\frac{2mnk}{\sqrt{S}}-2S-\mathcal{O}(mn).$$

Thus, our new lower bound for $C := AB + C$ yields a new lower bound on the I/O cost of $C := AB$.

4 Attaining the Lower Bound

In this section, we first develop intuition for how to attain near-optimal I/O cost for matrix-matrix multiplication algorithms. This motivates an algorithm that attains the lower bound for a processor with two layers of memory, a fast and a slow memory. Finally, we discuss how the currently most practical high-performance algorithm by Kazushige Goto [2].

4.1 Blocked algorithms

It has been well-known since the arrival of hierarchical memories in the 1980s that high-performance implementations of many dense linear algebra algorithms, including matrix-matrix multiplication, are facilitated by so-called blocked algorithms. The reason is simple: Multiplying $C := AB + C$ when $m = n = k = n_b$ and all matrices fit into fast memory allows $\mathcal{O}(n_b^2)$ I/O operations to be amortized over $2n_b^3$ scalar floating point operations (flops). When m, n, k are larger than n_b, the operands can be blocked into $n_b \times n_b$ submatrices and staged as a sequence of MMA operations with these submatrices. The question is how to optimally break the operands into submatrices.

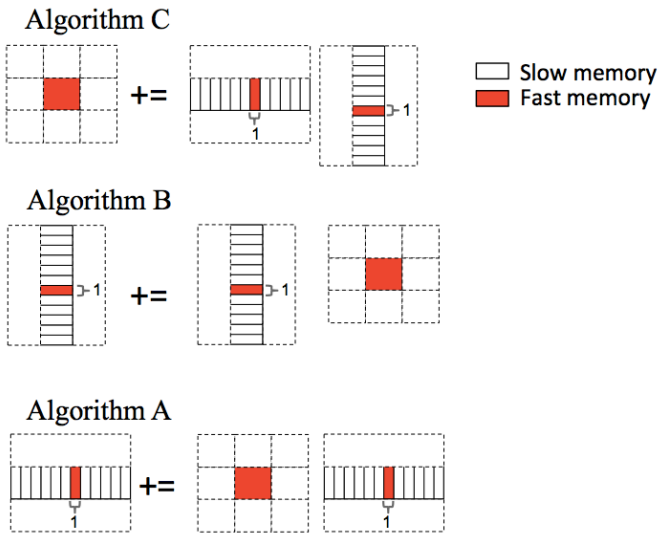

Figure 1: Illustration of Algorithms C, B, and A.

4.2 Insights

From the I/O lower bound $Q_{AB+C} \geq 2mnk/\sqrt{S} - 2S$, and from the fact that a matrix multiplication requires mnk FMAs, we know that if an optimal algorithm exists, it must perform $\sqrt{S}/2$ FMAs per I/O operation for large matrices. Rewriting the ratio as S FMAs per $2\sqrt{S}$ I/Os, the following points towards an algorithm: Assuming a $\sqrt{S} \times \sqrt{S}$ block of C is already in fast memory, perform a rank-1 update of that block of C memory using the appropriate part of a column of A with $\sqrt{S}$ elements and the appropriate part of a row of B with $\sqrt{S}$ elements. This requires $2\sqrt{S}$ I/O ops for S FMAs.

Fast memory cannot hold all of these elements at once. However it is possible to do so with a $(\sqrt{S} - 1) \times (\sqrt{S} - 1)$ block of C instead, leaving enough room for $\sqrt{S} - 1$ elements of A and $\sqrt{S} - 1$ elements of B, thus achieving close to the desired I/O lower bound. By performing the update of the block of C as a sequence of rank-1 updates, bringing that block of C into fast memory can be amortized over many flops, thus essentially achieving the assumption that the block is already in fast memory.

4.3 An asymptotically optimal blocked algorithm

Algorithm C: We are now ready to give an optimal algorithm (in the sense of asymptotically attaining the lower bound). Consider $C := AB + C$. Partition:

$$C \to \left(\begin{array}{c|c|c} C_{0,0} & \cdots & C_{0,N-1} \\ \hline \vdots & & \vdots \\ \hline C_{M-1,0} & \cdots & C_{M-1,N-1} \end{array}\right), \quad A \to \left(\begin{array}{c} A_0 \\ \hline \vdots \\ \hline A_{M-1} \end{array}\right), \quad B \to \left(\begin{array}{c|c|c} B_0 & \cdots & B_{N-1} \end{array}\right),$$

where $C_{i,j}$ is $(\sqrt{S} - 1) \times (\sqrt{S} - 1)$, A_i is $(\sqrt{S} - 1) \times k$, and B_j is $k \times (\sqrt{S} - 1)$. Then a simple loop over all the blocks of C, computing $C_{i,j} = A_i B_j + C_{i,j}$ via rank-1 updates, requires

$$\frac{m}{(\sqrt{S}-1)} \frac{n}{(\sqrt{S}-1)} \left(2(\sqrt{S}-1)k + (\sqrt{S}-1)^2\right) = 2\frac{mnk}{\sqrt{S}-1} + mn \approx 2\frac{mnk}{\sqrt{S}} + mn$$

reads and mn writes. When mnk is large, this algorithm (illustrated in Figure 1) approaches the lower bound.

4.4 Read-optimal and write-hidden algorithms

Algorithm B: An early occurrence of an algorithm that we can now realize is optimal in terms of the number of reads was presented and analyzed in [6]. Partition:

$$C \to \left(\begin{array}{c|c|c} C_0 & \cdots & C_{N-1} \end{array}\right), \quad A \to \left(\begin{array}{c|c|c} A_0 & \cdots & A_{N-1} \end{array}\right), \quad B \to \left(\begin{array}{c|c|c} B_{0,0} & \cdots & B_{0,N-1} \\ \hline \vdots & & \vdots \\ \hline B_{M-1,0} & \cdots & B_{M-1,N-1} \end{array}\right),$$

where C_j and A_p are $m \times (\sqrt{S}-1)$, and $B_{p,j}$ is $(\sqrt{S}-1) \times (\sqrt{S}-1)$. Then a simple loop over all blocks of B, keeping $B_{p,j}$ in fast memory and streaming rows of C_j and A_p while computing $C_j := A_p B_{p,j} + C_j$, requires

$$\frac{k}{(\sqrt{S}-1)} \frac{n}{(\sqrt{S}-1)} \left(2m(\sqrt{S}-1) + (\sqrt{S}-1)^2\right) \approx 2\frac{mnk}{\sqrt{S}} + nk$$

reads and

$$\frac{k}{(\sqrt{S}-1)} \frac{n}{(\sqrt{S}-1)} m(\sqrt{S}-1) \approx \frac{mnk}{\sqrt{S}}$$

writes.

The input cost of this algorithm, illustrated in Figure 1, is essentially equal to the I/O lower bound, but it requires many writes to slow memory and so cannot be considered I/O optimal. On the other hand, processors often have full-duplex memory bandwidth (meaning that the bandwidth available for reads is separate from the bandwidth available for writes), so the output cost may not be visible if it is less than or equal to than the input cost and if the reads and writes can be overlapped. Since that is the case for this algorithm, it may execute just as efficiently as the algorithm presented in Section 4.2. Thus, we can say that this algorithm is read-optimal and write-hidden. This becomes important when we later discuss practical implementations.

Algorithm A: We now present an algorithm that is in some sense the mirror image to Algorithm B, keeping a square block of A in fast memory instead instead of a square block of B. Partition:

$$C \to \left(\begin{array}{c} C_0 \\ \hline \vdots \\ \hline C_{M-1} \end{array}\right), \quad A \to \left(\begin{array}{c|c|c} A_{0,0} & \cdots & A_{0,K-1} \\ \hline \vdots & & \vdots \\ \hline A_{M-1,0} & \cdots & A_{K-1,N-1} \end{array}\right), \quad B \to \left(\begin{array}{c} B_0 \\ \hline \vdots \\ \hline B_{K-1} \end{array}\right),$$

where C_i and B_p are $(\sqrt{S}-1) \times n$, and $A_{i,p}$ is $(\sqrt{S}-1) \times (\sqrt{S}-1)$. Then a simple loop over all blocks of A, keeping $A_{i,p}$ in fast memory and streaming columns of C_i and B_p while computing $C_i := A_{i,p} B_p + C_i$ yields an algorithm with I/O cost of requires

$$\frac{k}{(\sqrt{S}-1)} \frac{n}{(\sqrt{S}-1)} \left(2m(\sqrt{S}-1) + (\sqrt{S}-1)^2\right) \approx 2\frac{mnk}{\sqrt{S}} + mk$$

reads and

$$\frac{k}{(\sqrt{S}-1)} \frac{n}{(\sqrt{S}-1)} m(\sqrt{S}-1) \approx \frac{mnk}{\sqrt{S}}$$

writes. The algorithm is also illustrated in Figure 1.

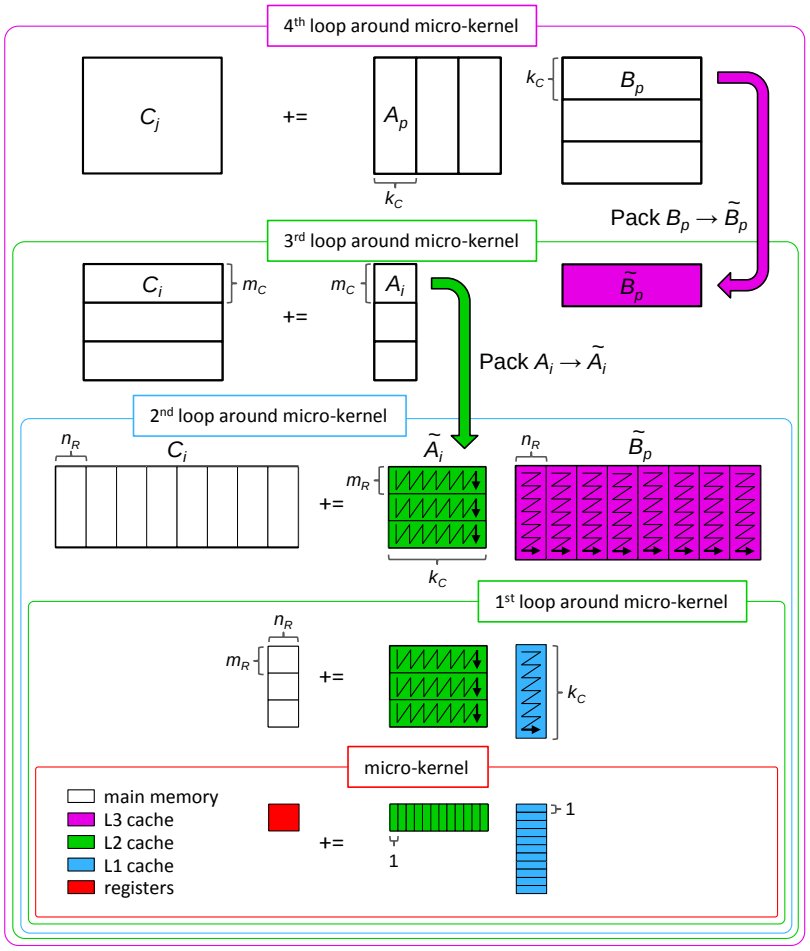

Figure 2: Illustration of the BLIS implementation of Goto's algorithm. This illustration was first designed for use in [10]. Used with permission.

4.5 Practical implications

We now discuss how current practical algorithms measure up against the theoretical lower bound.

4.5.1 Goto's algorithm

The algorithm used in the state-of-the-art GotoBLAS [2] and implemented in the BLAS-like Library Instantiation Software (BLIS) [12, 11] is illustrated in Figure 2. Those who are interested in the practical implementation of matrix multiplication should be familiar with these various papers, and hence we do not go into detail here. Goto's algorithm is a practical algorithm that targets multiple layers of cache. Details of how this algorithm is parameterized based on machine constants can be found in [8]. We will show that the results for this theoretical paper targeting a single layer of cache can be used to analyze in what sense Goto's algorithm does and in what sense it does not achieve optimality with respect to I/O [2].

[2] In our analysis we ignore an extra I/O cost of copying submatrices into contiguous buffers.

Blocking for the L3 cache. First, the algorithm partitions the matrices in the n dimension with a blocksize of n_c. Then, it partitions the k dimension with a blocksize of k_c, where k_c is determined by further subsequent blocking for the L2 cache. This bounds the size of a short and very wide panel of B that fill "most of" the L3 cache. Subsequent loops encourage the cache replacement policy to retain that panel of B in the L3 cache. The number of reads into the L3 cache from main memory is given by $\frac{mnk}{n_c} + \frac{mnk}{k_c} + nk$. This is far from optimal because typically $n_c \gg k_c$ and hence $k_c \ll \sqrt{S_3}$, where S_3 is the size of the L3 cache.

Blocking for the L2 cache. The algorithm next partitions the m dimension with a blocksize of m_c. This creates a block of A that will reside in the L2 cache, similar to Algorithm A. If we assume that n_c is very large, then we can approximate the number of reads into the L2 cache from slower layer of memory by $\frac{mnk}{m_c} + \frac{mnk}{k_c} + mk$. This is close to optimal because typically the block of A in the L2 cache is roughly square and it occupies nearly the entire L2 cache: In this case $m_c \approx k_c \approx \sqrt{S}$, so when m, n, and k are large we can ignore the mk term and, $\frac{mnk}{m_c} + \frac{mnk}{k_c} \approx \frac{2mnk}{\sqrt{S}}$.

Subsequent layers of cache. For the L1 cache, the algorithm partitions the n dimension with blocksize n_r. This creates a block of B that will reside in the L1 cache, however it is far from square, and so the number of reads into the L1 cache is suboptimal for similar reasons that the number of reads into the L3 cache is suboptimal.

The final step that we will consider is that algorithm then partitions in the m dimension with blocksize m_r. This creates a roughly square block of C that will reside in registers, and updated by a series of rank-1 updates, similar to the optimal Algorithm C.

Summary. We conclude that Goto's algorithm is optimal in the sense that it optimizes for the number of L2 cache misses, but is suboptimal in terms of L3 cache misses. Goto's algorithm is a practical one, and matrix multiplication on modern machines does not need to optimize for L3 cache misses because there is sufficient bandwidth available from main memory. However in the future it is possible that even matrix multiplication may become bandwidth limited, and Goto's algorithm may need to be tweaked so that it places a square matrix block in the L3 cache.

4.5.2 A family of algorithms

Gunnels, et al. [3] presented a family of algorithms for implementing MMMA. It discussed three shapes of MMMA that correspond to Algorithms A, B, and C and hence are read-optimal and write-hidden. These algorithms result from asking the question of how to optimally block between two adjacent layers of memory and applying this to multiple layers in the hierarchy. The locally optimal solution is that when one of the three algorithms is used for some layer of memory, one of the other two algorithms should be used at the next faster layer. While this yields the discussed read optimal and write hidden algorithms, it resulted from fixing the family of algorithms under consideration and therefore was not known to be optimal until the result in the present paper was obtained.

5 Conclusion

In this paper, we improved the I/O lower bound for $C := AB + C$ to $2mnk/\sqrt{S} - 2S$ and for $C := AB$ to: $2mnk/\sqrt{S} - 2S - \mathcal{O}(mn)$. We showed that these lower bounds are sharp with respect to the constant on the highest ordered term by analyzing known algorithms. We also analyzed the state-of-the-art Goto's

algorithm and noted its strengths and weaknesses in light of the MMMA lower bound. These lower bounds are not only of interest as a theoretical result but also to help gain fundamental insight into how MMM must be implemented.

We believe that the proof techniques presented in this paper can apply to algorithms outside of matrix multiplication. In particular, in the domain of linear algebra, we believe they can be combined with the techniques introduced by Ballard et al. [1] in order to find lower bounds with sharp constants for other matrix operations such as the LU, QR, and Cholesky factorizations.

While the proof in this paper only applies to algorithms that do not use workspace for the matrix C, we suspect that the lower bound applies in this case. Future work is to prove this to be true. We do not think that modifying the proof technique of breaking the problem into phases is useful for doing so, because the lower bound in this paper requires three inputs to one unit of computation, the multiply-add. If the scalar multiplication occurs during a different phase from a corresponding scalar addition, then it appaers difficult to apply this proof technique. One option would be to prove that a matrix-matrix multiplication with workspace always requires more I/O than one without workspace. Another option would of course be to disprove our suspicions and find an algorithm that uses workspace for C to have an I/O cost less than our lower bound.

Acknowledgements

This work is supported by the National Science Foundation, under grant Award ACI-1550493 and the Science of High-Performance Computing group's Intel Parallel Computing Center.

References

[1] Grey Ballard, E Carson, J Demmel, M Hoemmen, Nicholas Knight, and Oded Schwartz. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica*, 23:1–155, 2014.

[2] Kazushige Goto and Robert van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3), May 2008.

[3] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A family of high-performance matrix multiplication algorithms. In *ICCS '01*, 2001.

[4] Jia-Wei Hong and Hsiang-Tsung Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333. ACM, 1981.

[5] Dror Irony, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

[6] Monica D Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 63–74. ACM, 1991.

[7] Lynn H. Loomis and Hassler Whitney. An inequality related to the isoperimetric inequality. *Bulletin of the American Mathematical Society*, 55:961–962, 1949.

[8] Tze Meng Low, Francisco D Igual, Tyler M Smith, and Enrique S Quintana-Orti. Analytical modeling is enough for high-performance BLIS. *ACM Transactions on Mathematical Software*, 43(2):12, 2016.

[9] John E Savage. Extending the hong-kung model to memory hierarchies. In *Computing and Combinatorics*, pages 270–281. Springer, 1995.

[10] Field G Van Zee and Tyler M Smith. Inducing complex matrix multiplication via the 3M and 4M methods FLAME Working Note# 81. 2016. Submitted to ACM Trans. Math. Softw.

[11] Field G. Van Zee, Tyler M. Smith, Bryan Marker Bryan, Tze Meng Low, Robert van de Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John Gunnels, and Lee Killough. The BLIS framework: Experiments in portability. *ACM Transactions on Mathematical Software*, 42(2):12, 2016.

[12] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 41(3), 2015.

A Constrained Global Maximum of $\sqrt{xyz}$

In this appendix, we give details on how the optimal F is determined. The problem to be solved is

$$\text{maximize } F \text{ under the constraints } \begin{cases} F \leq \sqrt{xyz} \\ 0 \leq x, y, z \\ x + y + z \leq S + M \end{cases}.$$

We first observe that if any of x, y, or z is zero, then so is F and hence will only consider the case where $0 < x, y, z$. Also, if $x + y + z$ are strictly less than $S + M$, then one of x, y, or z can be increased, thereby increasing F, and hence we only need to consider $x + y + z = S + M$. Finally, given these constraints we can optimize $F = \sqrt{xyz}$, as long as we check that the result is a maximum. The constrained problem thus becomes

$$\text{maximize } F = \sqrt{xyz} \text{ under the constraints } \begin{cases} 0 < x, y, z \\ x + y + z = S + M \end{cases}.$$

We can use the Lagrange Multiplier method to solve $\nabla F = \lambda \nabla (x + y + z - (S + M))$ for x, y, z. Hence

$$\frac{yz}{2\sqrt{xyz}} = \lambda, \quad \frac{xy}{2\sqrt{xyz}} = \lambda, \quad \frac{xz}{2\sqrt{xyz}} = \lambda, \quad \text{and } S + M = x + y + z.$$

Since then $yz = xy = xz$ and we know that x, y and z are nonzero, we deduce that $x = y = z$ and hence $S + M = 3x$. As a result, the solution is $x = y = z = (S + M)/3$. To show that this is a global maximum, we can find the second derivative of F at this point, or we can evaluate F at this point and any point on the boundary of our region to show that any value on the boundary is smaller.

We conclude that the global maximum of F is:

$$F = \frac{S + M}{3} \sqrt{\frac{S + M}{3}} = \frac{(S + M)\sqrt{S + M}}{3\sqrt{3}}$$