# Learning from Optimizing Matrix-Matrix Multiplication

Devangi N. Parikh, Jianyu Huang, Margaret E. Myers, Robert A. van de Geijn
*The University of Texas at Austin*
*Austin, Texas, USA*
{*dnp, jianyu, myers, rvdg*}*@cs.utexas.edu*

***Abstract*—We describe a learning process that uses one of the simplest examples, matrix-matrix multiplication, to illustrate issues that underlie parallel high-performance computing. It is accessible at multiple levels: simple enough to use early in a curriculum yet rich enough to benefit a more advanced software developer. A carefully designed and scaffolded set of exercises leads the learner from a naive implementation towards one that extracts parallelism at multiple levels, ranging from instruction level parallelism to multithreaded parallelism via OpenMP to distributed memory parallelism using MPI. The importance of effectively leveraging the memory hierarchy within and across nodes is exposed, as do the GotoBLAS and SUMMA algorithms. These materials will become part of a Massive Open Online Course (MOOC) to be offered in the future.**

***Keywords*-parallel computing; high performance computing; matrix-matrix multiplication; computing education; open education**

## I. Introduction

A purpose of exploiting parallelism is to compute an answer in less time. To achieve this, parallelism can and should be extracted at multiple levels: at the single core level via instruction level parallelism, at the node level via multithreading, and between nodes via distributed memory parallelism. Vital to achieving near perfect speedup, high performance, and scalability is the choice of algorithm, load balance, and amortization of the cost of data movement. These are principles the mastery of which is now of importance to any programmer of applications that are time-critical or low power.

For decades, there has been a stated desire to teach parallel computing early in the (undergraduate) curriculum [1], [2], [3], [4]. A key is a set of examples/activities that are simple, yet illustrate a reasonable subset of issues. Ideally, activities are rich enough to interest and benefit both novice and advanced software developers. Our experience over several decades of teaching the subject is that matrix-matrix multiplication (MMM) is one such example/activity.

### A. *Why matrix-matrix multiplication?*

There are a multitude of reasons:

**It is easy to define.** Given matrices $C$, $A$, and $B$ of sizes $m\times n$, $m\times k$, and $k\times n$, updating $C$ with the result of multiplying $A$ times $B$ is given by $C := AB + C$, which means each element $\gamma_{i,j}$ of $C$ is updated with $\gamma_{i,j} := \sum_{p=0}^{k-1} \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$, where $\alpha_{i,p}$ and $\beta_{p,j}$ equal the $i,p$ and $p,j$ elements of $A$ and $B$, respectively.

**It is taught early in the curriculum.** MMM is often already introduced in high school and most undergraduate programs in computer science and other sciences require linear algebra early in their core curriculum.

**Performance matters.** Matrix-matrix multiplication is at the core of many scientific applications and, more recently machine learning algorithms.

**Choice of algorithm matters.** What we will see later in this paper is that high-performance (parallel) implementations employ all in a family of algorithms.

**Parallelism is exploited at all levels.** High-performance requires instruction level, multi-threaded, and distributed memory parallelism.

**Data movement matters.** Key to high performance is the careful amortization of movement of data between memory layers, not only between nodes of a distributed memory architecture, but also between local memory, local caches, and the registers of a core.

**Data decomposition matters.** A simplistic distribution of data between nodes of a distributed memory architecture will inherently prevent even so-called weak scalability.

**It extends.** Contemporary operations, such as the computation of the K-Nearest Neighbor [5] and tensor contraction [6], [7], [8] are variations on MMM as are Strassen-like fast MMM algorithms [9], [10].

**It satisfies the need for speed.** Just like some become addicted to tinkering on race cars, driving them to push the limits of performance, the same is true for high-performance implementation of MMM.

MMM is simple yet complex enough to introduce many of the issues with which one grapples when confronted with the complexities of a modern sequential or parallel computer.

### B. *Related work*

MMM has often been used as an exercise when teaching about optimization. A quick Google search yielded numerous lecture notes and/or homework exercises that utilize this operation [11], [12]. What these materials have in common is that they cite a number of insightful papers [13], [14],

[15], [16], [17], [18]. We ourselves created the "how-to-optimize-gemm" wiki [19] and a sandbox that we call BLISlab [20] that build upon our BLAS-like Library Instantiation Software [21], [22] refactoring of the GotoBLAS approach [13] to implementing MMM. Others have created similar exercises [23].

Similarly, there are many course materials (e.g., [24], [25]) that build upon the SUMMA algorithm [26], [27] for distributed memory parallel MMM, variants of which are used in practical libraries like ScaLAPACK [28], PLAPACK [29], and Elemental [30].

The materials we describe are an attempt to provide an updated experience similar to these prior efforts that is carefully structured and integrated. When launched as a Massive Open Online Course (MOOC), it will scale.

## C. This paper

We narrate a set of exercises that will be part of a MOOC to be offered in the future. It is the third in a loosely-coupled set of MOOCs [31] that have been developed at UT-Austin. The first two are already offered on the edX platform and expose learners to HPC through enrichments.

The described exercises have been used in an on-campus upper-division computer science course at UT-Austin, titled "Programming for Correctness and Performance." At the writing of this paper, it is the exercises related to parallelism within a core and node that have been developed. The exercises related to the distributed memory parallelization are described, but are still under construction.

The notes that are being written for this course, as well as the related activities, can be found at http://www.ulaff.net.

## II. Naive Implementations

We start the journey towards optimization with a simplest implementation of MMM. It is described how matrices are mapped to memory, and by playing with the ordering of the triple-nested loop, the learner discovers that there is a performance benefit that comes from accessing contiguous memory (spacial locality). The learner also finds out that these simple implementations are grossly suboptimal, relative to the theoretical peak of a processor and to a high-performance reference implementation.

### A. A simple algorithm

The learner starts with a C implementation for computing MMM. This results in a triple-nested loop,

```
for $i := 0, \ldots, m-1$
  for $j := 0, \ldots, n-1$
    for $p := 0, \ldots, k-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end
  end
end
```

This loop ordering, IJP, casts the computations in terms of dot products of the rows of $A$ and columns of $B$. Learners are introduced to different ways the elements of a matrix are stored in memory, and how this code strides through the memory to access the various elements of each matrix when column-major ordering is used.

With this exercise, learners are also introduced to measuring the performance of the implementation of the code, in terms of timing information and rate of computation (GFLOPS) as a function of problem size and execution time. They calculate the theoretical peak performance based on the specification of their processor. This gives the learner an idea of the target performance of this compute bound operation.

Figure 1 (left) is representative of the graph that the learners create from the exercise. Experiencing that a naive implementation only achieves a tiny fraction of the theoretical peak of a processor and a high-performance reference implementation is a real eye-opener.

### B. Ordering the loops

Experienced programmers know that computing with data that are contiguous in memory ("stride one access") is better than computing with data that are accessed with a larger stride. The learners experiment with this by changing the loop ordering to see its effect on the performance.

They learn that

- For the IJP and JIP orderings, the inner-most loop casts the computations in terms of dot products (DOT) of the appropriate row and column of $A$ and $B$ respectively. It is the order in which the elements of $C$ are visited that are determined by the ordering of the outer two loops.
- The IPJ and PIJ orderings cast the inner-most computations in terms of an AXPY operation (scalar times vector added to a vector) with row vectors, while the outer two loops visit each element of the matrix $A$.
- The JPI and PJI orderings cast the inner-most computations in terms of an AXPY operation with column vectors, while the outer two loops visit each element in the matrix $B$.

Since we adopt column-major storage in our exercises, the JPI ordering has the most favorable access pattern. This is observed in practice, as illustrated in Figure 1 (right).

### C. Simple layering of the operations

The exercises so far already introduce the learner to the idea that MMM can be implemented in terms of DOT and AXPY operations. To prepare for the many optimizations that follow, the learners go through a number of exercises that illustrate how MMM can also be implemented as a single loop around a matrix-vector or rank-1 update operation, each of which itself can be implemented in terms of DOT or AXPY operations. This introduces them to the functionality supported by the Basic Linear Algebra Subprograms (BLAS) [33], [34], [35] and that there are families of algorithms for MMM.

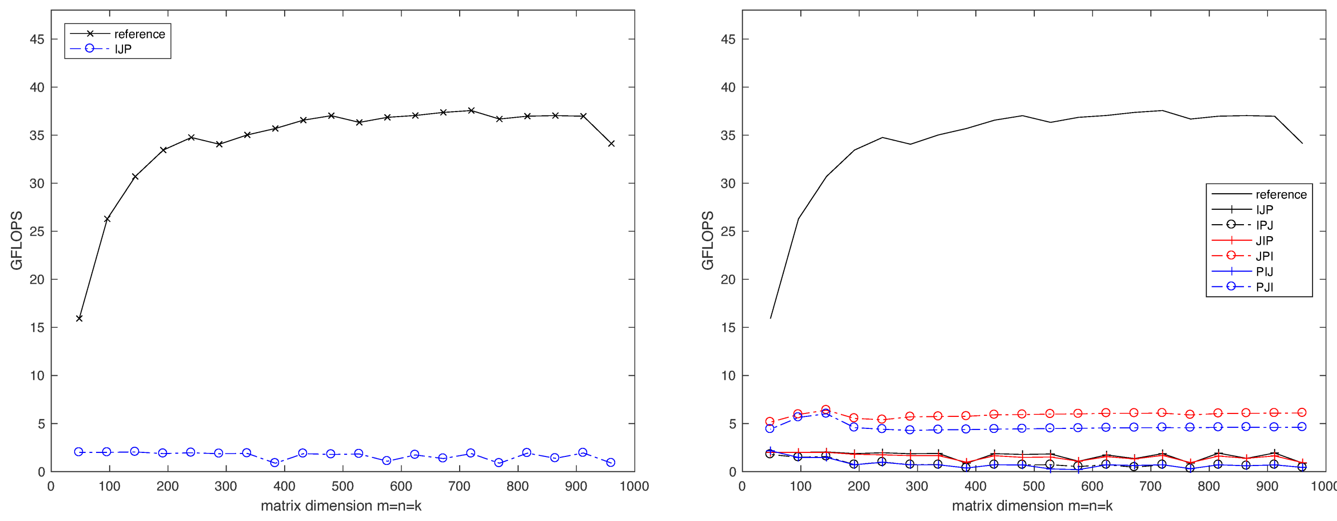

Figure 1. Left: Performance of a simple triple-nested loop, compared to a high-performance reference implementation (our BLIS library [21], [32]). Right: Performance of all orderings of the loops. The top of the graphs represents the theoretical peak of the processor.

## III. Optimizing for a Single Core

The previous section set the stage: learners are now fully aware of the inadequacy of simple implementations and the limitations of optimizing compilers. The next step is to extract instruction level parallelism through calls to single-instruction, multiple data (SIMD) operations, which execute identical floating point operations simultaneously on different data. Using fast operations is not enough: careful attention must be paid to the memory hierarchy and the (near) optimal amortization of data movement over useful computation.

### A. Blocked MMM

In order to take advantage of the memory hierarchy, matrices are blocked into submatrices

$$X = \begin{pmatrix} X_{0,0} & X_{0,1}, \dots \\ X_{1,0} & X_{1,1}, \dots \\ \vdots & \vdots \end{pmatrix} \quad \text{for } X \in \{C, A, B\} \qquad (1)$$

after which $C := AB + C$ can be computed by blocks: $C_{i,j} := A_{i,0}B_{0,j} + \cdots + A_{i,K-1}B_{K-1,j} + C_{i,j}$. Throughout the remaining exercises, a family of blocked versions of MMM improve temporal and spacial locality.

### B. Vector registers and instructions

In order to access the full potential of the CPU, computation must be cast in terms of vector operations with data that are stored in vector registers. After introducing these concepts and the corresponding intrinsic commands via the C language, the learner completes a routine that implements $C_{i,j} := A_iB_j + C_{i,j}$, where $C_{i,j}$ is a $4 \times 4$ matrix. Initially, the size of $C_{i,j}$ is arbitrary since the focus is on how to cast computation in terms of vector operations in the SIMD instruction set of the target processor.

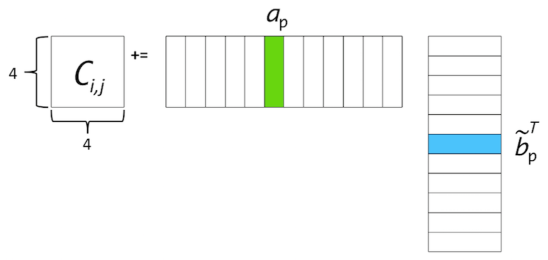

Figure 2. Computing $C_{i,j} := A_iB_j + C_{i,j}$ where $C_{i,j}$ is $4 \times 4$ in terms of a sequence of rank-1 updates $a_p\widetilde{b}_p^T$.

The computation is orchestrated as a loop around rank-1 updates, where for each rank-1 update a vector register is loaded with column $a_p$ of $A_i$, and another one is loaded with elements in row $\widetilde{b}_p$ of $B_j$, as illustrated in Figure 2. We refer to this sub-operation as the micro-kernel for MMM. Through its implementation the learner also gains an understanding of instruction latency and data dependency issues to avoid stalls in the instruction pipeline.

To then compute $C := AB + C$, where $C$ is partitioned as in (1), and

$$A = \begin{pmatrix} A_0 \\ A_1 \\ \vdots \end{pmatrix} \text{ and } B = \begin{pmatrix} B_0 & B_1 & \cdots \end{pmatrix},$$

and a routine that implements the double nested loop around the kernel that computes $C_{i,j} = A_iB_j + C_{i,j}$ is employed.

Figure 3 (left) illustrates the performance benefits that the learner observes.

From the implementation of the $4 \times 4$ kernel for double-precision floating point on a CPU with vector length of 256 bits, learners make the following observation:

- The submatrix $C_{i,j}$ requires 4 vector registers. The values of $C_{i,j}$ are loaded into the vector registers, updated, and then written to memory only once for an arbitrary value of $k$, and is hence reused $k$ times.
- In general, the kernel updates the values of the submatrix $C_{i,j}$ that can be of size $m_R \times n_R$. If $r_C$ is the number of doubles the vector registers can hold then $m_R \times n_R \leq r_C$.
- For each iteration of a kernel of size $m_R \times n_R$, loading $m_R$ doubles from the matrix $A$ and $n_R$ doubles from the matrix $B$ is amortized over $2m_R n_R$ flops. Thus, the flops/load ratio is $\frac{2m_R n_R}{m_R+n_R}$

With this understanding, learners experiment with varying the values of $m_R$ and $n_R$. They analyze how many vector registers each implementation requires and observe the overall performance of the resulting implementations. From these experiments, the learner empirically determines the optimal value of $m_R$ and $n_R$. Figure 3 (right) illustrates the performance for various micro-kernels.

Under a simplified model, the optimal values of $m_R$ and $n_R$ can be analytically determined by maximizing the flops/load ratio given the constraint of the registers available. Learners note that under these constraints the block of $C_{i,j}$ that resides in the registers must be squarish. The interested learner is pointed to further reading [36] that refines the model and better predicts optimal choices.

These exercises link what was learned about casting MMM in terms of rank-1 update to the reuse of data that is stored in registers to blocked MMM. Better performance is achieved, but there is still considerable room for improvement.

## C. Blocking for multiple levels of cache memory

The exercises so far assume a simple memory model of main memory and registers. Learners note from Figure 3 that the performance of their MMM implementation based on the vectorized micro-kernel is fast for smaller matrix sizes, but they observe a performance drop for larger matrices. This drop in performance is a segue to teach the learners about memory hierarchy in modern computers. After an introduction to multi-level caches, the learners understand that the drop in performance is because elements of the matrix are not in cache.

A naive way to block for caches is to look at Figure 3 (right) and notice that good performance is achieved for matrices with $m = n = k = 48$. By partitioning all matrices into submatrices of that size, a larger MMM is then cast in terms of subproblems of that size. Following this approach, the learner creates an implementation that yields the performance reported in Figure 4 (left), labeled `GemmFiveLoops_12x4`.

## D. Near-optimal blocking with packing

The question becomes how to optimally block for all caches. Through an additional sequence of steps, the learner is introduced to a near-optimal approach pioneered by Kazushige Goto [13] that is illustrated in Figure 5. It blocks for all three levels of cache and packs data for further locality. With appropriately chosen block sizes, the learner's implementation reaches the performance reported in Figure 4 (right). This same approach underlies the implementation labeled "reference" from our BLIS library.

There are a few final techniques that are suggested to the learner, such as prefetching, loop unrolling, and vector instructions that force alignment, which help further close the gap with the "reference" implementation.

## E. Summary

What has been described is a set of carefully scaffolded exercises that quickly take the learner from the naive implementation that achieves only a fraction of the CPU peak performance to a state-of-the-art implementation, while introducing the learner to important concepts in computer architecture that enable high-performance implementations on a single core.

## IV. Multi Core Implementation

Once equipped with the tools to write a high-performance MMM code for a single core, the learners are now ready to parallelize across multiple cores that have shared memory. In the process, they are exposed to multi-threading with OpenMP [38].

## A. Opportunities for Parallelism

Blocking for multiple levels of cache results in an implementation that has five loops around the kernel as shown in Figure 5. These various loops give us multiple opportunities for parallelism [39]. Learners use OpenMP to experiment with parallelizing different loops. Which loop is chosen for parallelization influences how much parallelism exists, the granularity of the tasks that are concurrently executed, and the efficiency with which caches are used depending on how they are shared by cores.

For example, for a machine with a shared L3 cache, but private L2 caches, parallelizing the third loop around the micro-kernel has some advantage over parallelizing the second loop, since the L2 cache is more efficiently used. This is the case because each thread computes with its own block $\widetilde{A}$, and shared panel $\widetilde{B}$. However, parallelizing the third loop yields a coarser granularity of tasks, which can negatively affect load balance. In other words: there is a tension between concerns. A representative graph of performance is given in Figure 6.

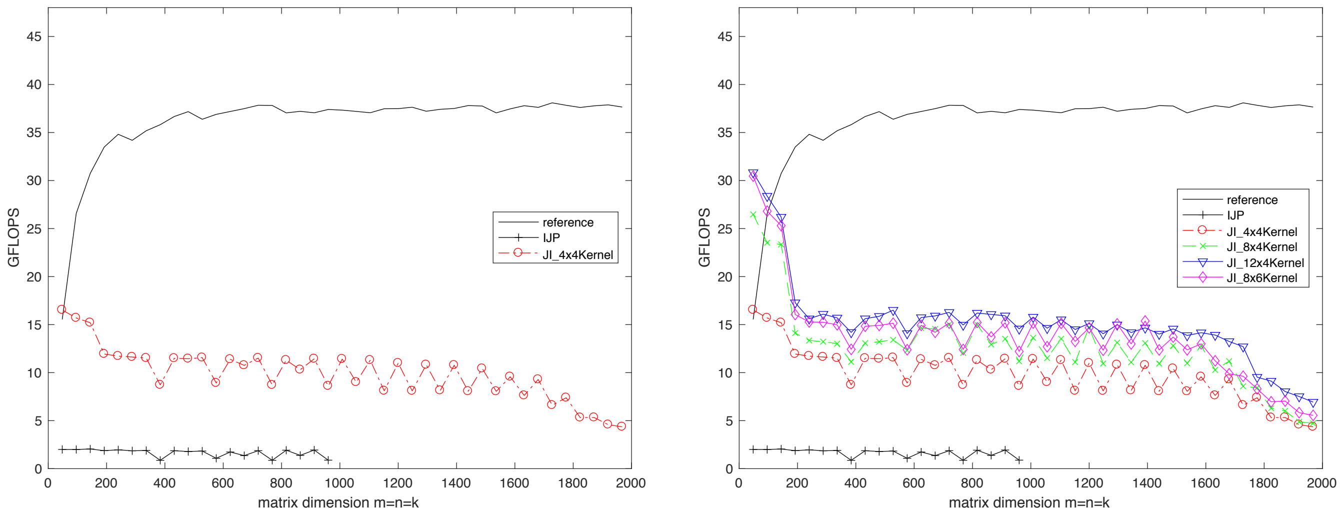

Figure 3. Left: Performance MMM cast in terms of $4 \times 4$ kernel. Right: Performance when the choices of $m_R$ and $n_R$ are varied.

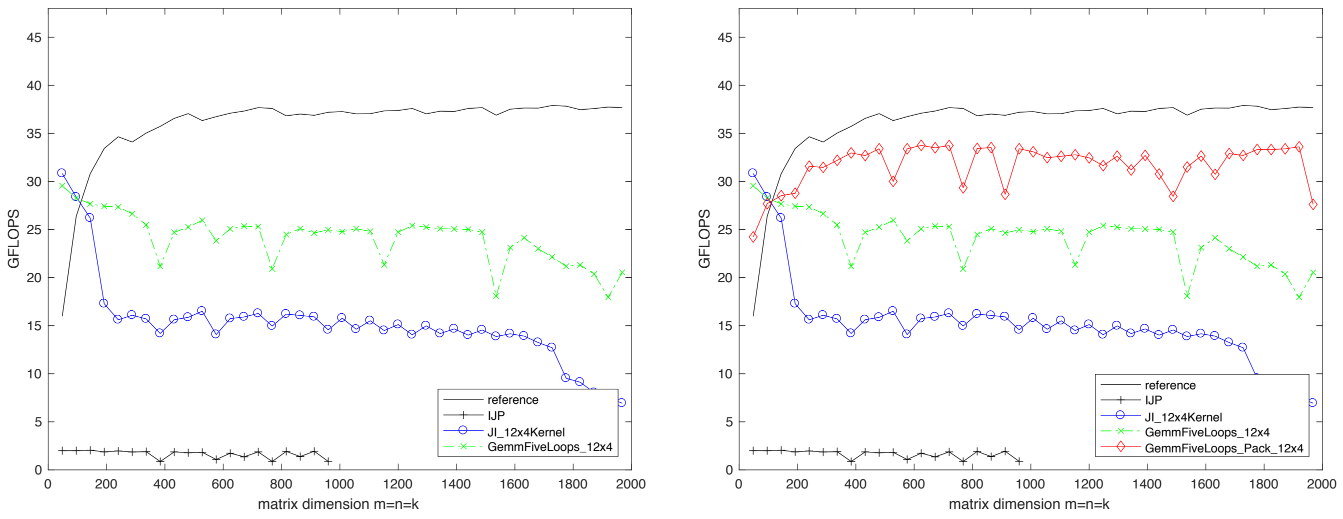

Figure 4. Left: Performance blocked MMM where all three matrices are blocked into $48 \times 48$ submatrices, and for each $C_{i,j} := A_{i,p}B_{p,j} + C_{i,j}$ the implementation from Section III is used. Right: Implementation of Figure 5 with packing and near-optimal parameter choices.

The code with which they start is written so that the learners have to struggle with the concepts of race conditions, cache coherency, and data sharing issues.

## B. *Performance Metrics*

When writing multi-threaded code, it is important to understand how well the code performs. Learners time their multi-threaded code, then calculate the speedup and efficiency that is attained. The limitations imposed by Amdahl's law are an important concept for understanding what parts of the code need further optimization.

## C. *Other Insights on Multicore Parallelization*

Equipped with the knowledge gained so far, learners are able to identify other possible ways they can further improve performance. For example, they observe that the packing routines can also be parallelized across the threads to distribute that workload among the cores as well. If this is not done, then the packing is a limiting factor, per Amdahl's law.

## D. *Summary*

The learner walks away understanding some core ideas:

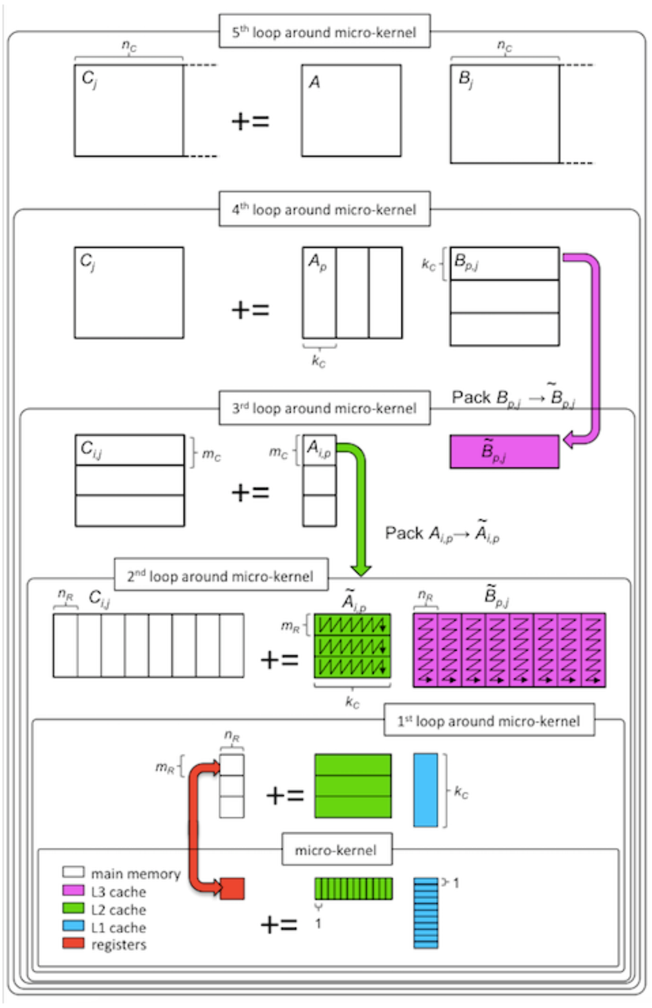
5th loop around micro-kernel
4th loop around micro-kernel
3rd loop around micro-kernel
2nd loop around micro-kernel
1st loop around micro-kernel
micro-kernel
Pack $B_{p,j} \rightarrow \tilde{B}_{p,j}$
Pack $A_{i,p} \rightarrow \tilde{A}_{i,p}$
main memory
L3 cache
L2 cache
L1 cache
registers

Figure 5. Blocking for multiple levels of cache, with packing. Picture adapted from [37]

- Parallelization should only be started once the optimization on a single core is mostly completed.
- Optimization on a single core is all about temporal and spacial locality. Techniques for achieving this often prepare an implementation for parallelization.
- How to extract parallelism is often machine dependent.
- All major components of an implementation need to be examined for possible parallelization in order to avoid the pitfalls dictated by Amdahl's law.
- OpenMP is a powerful mechanism for expressing parallelism.

## V. Distributed Memory Implementation

The system that has been targeted up to this point is what become a node in a distributed memory architecture. MMM now becomes the example that demonstrates that the effective use of a distributed memory computer applies the same principles: amortize the cost of data movement over useful computation.

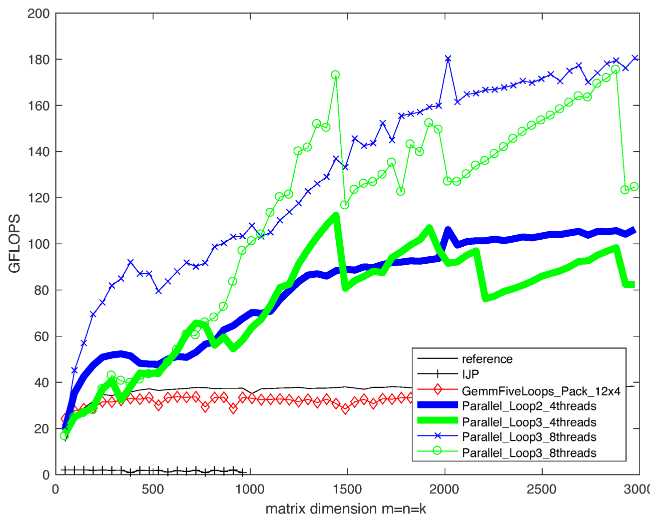

Figure 6. Parallelization of the second and third loop around the microkernel.

### A. *Programming distributed memory architectures*

Modern supercomputers consist of thousands of nodes, each an individual off-the-shelf computer. The accepted programming paradigm for such a system is known as Single Program Multiple Data (SPMD) programming, where a single program is executed on all the nodes available. Each node has a unique identifier. Programs take a different execution path and data is shared as needed, based on the identifier of the node on which it is executed. Typically, the *Message Passing Interface* (MPI) [40] is used for communication. Learners are first introduced to programming on a distributed memory system by writing a simple "Hello World" program using MPI routines.

### B. *Collective communications*

For nodes to come together to collectively solve a problem, input data needed and/or the answer a node produces must be shared with other nodes. This is the data movement that must be amortized over useful computation.

Learners are introduced to collective communications (collectives), the data redistribution and consolidation operations encountered in practical parallel algorithms for MMM [26], [27]. Given a simple model of the cost of point-to-point communication, they calculate lower bounds on the cost of these collectives. Various algorithms are discovered and their costs analyzed. Importantly, learners are exposed to how collectives can be composed to implement other collectives [41], as illustrated in Figure 7. This "calculus" of collectives later simplifies the development of a family of algorithms for MMM on distributed memory architectures.

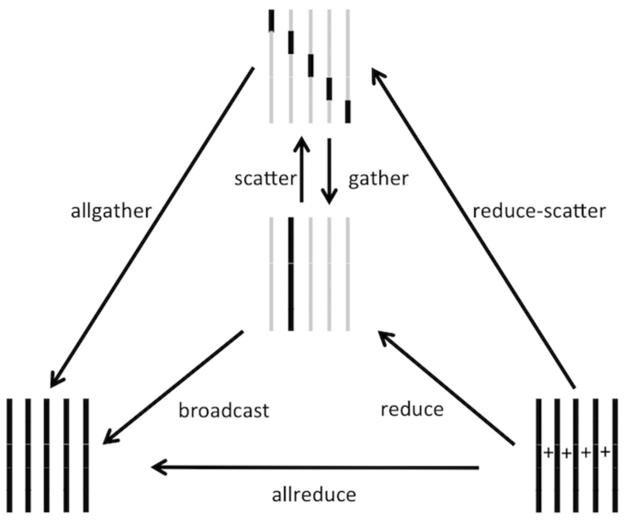

Figure 7. A calculus of collective communications.

A benefit of casting algorithms in terms of computation interleaved with collectives is that this is an example of programming with the Block Synchronous Parallel (BSP) model, which makes cost analyses simpler.

### *C. Distributed Memory MMM*

Once equipped with an understanding of the collectives, learners are now ready to implement distributed memory MMM. Starting with a parallel matrix-vector implementation on a 1D compute grid, they analyze the cost of their implementation. As for the multi-threaded code, learners calculate the speedup and efficiency of their implementation, as well as its scalability. They find that 1D algorithms are inherently not weakly scalable. Through further exercises they find that a two-dimensional data distribution is required for weak scalability to be achieved. Learners apply this to the parallelization of the rank-1 update operation next.

Learners already know how to cast MMM in terms of rank-1 updates. They analyze that a distributed memory MMM in terms of rank-1 updates is weakly scalable. However, it is not high performing since the underlying computations on each node (local rank-1 updates) do not attain high performance. Learners are able to modify their implementation such that the computations on each node are a local MMM instead of a rank-1 update, and achieve high performance that is weakly scalable.

Through simple examples, learners gain insights that have been published in [26], [27]. What they also observe is that even for an operation as simple as MMM, coding in terms of MPI calls and local calls to the BLAS is cumbersome and error prone. To alleviate this, they discover how to abstract away from details of distribution, much like PLAPACK [29] and Elemental [30] do. In our experience, this is a skill that needs mastering if one is to become a practitioner of HPC targeting large distributed memory systems.

### *D. Summary*

This part of the course is still very much in the design phase. Developing and testing the implementations can be done on a typical laptop or desktop computer. Gathering performance data requires access to an actual distributed memory system. A major question is how to give those learners who lack access to high performance distributed memory parallel systems the ability to experience those architectures.

## VI. Conclusion

We have described a set of learning experiences centered around MMM. Together, they introduce the learner to parallelization at different levels of the architecture: within a core, within a processor, and across a distributed memory system. Along the way, the learner experiences the importance of amortization of data movement over useful computations (GFLOPS). The learner is thus exposed to vector instructions, OpenMP, and MPI, as well as concepts like efficiency, rate of computation, speedup, and scalability.

The materials provide the learners just enough information, just in time. Enrichments provide an opportunity to further deepen their knowledge. This makes it suitable for an audience ranging from the novice with some experience with the C programming language to a domain scientist who realizes high performance is needed to solve problems in their domain. In addition to it reaching the life-long learners who gravitate towards MOOCs, exercises might be used in a traditional classroom and other settings.

For some, success lies not with achieving a higher level of proficiency in HPC: The realization that optimization is best left to experts and that performance can be achieved through use of high-performance software libraries is valuable as well.

## Acknowledgment

This work was sponsored in part by NSF under grant numbers ACI-1148125, ACI-1550493, CCF-1714091, a grant from the University of Texas System, and a gift from MathWorks. We thank the rest of the SHPC team (http://shpc.ices.utexas.edu) for their support.

## References

[1] M. J. Meredith, "Introducing parallel computing into the undergraduate computer science curriculum: A progress report," in *SIGCSE*, 1992.

[2] D. Johnson *et al.*, "Teaching parallel computing to freshmen," in *Conference on Parallel Computing for Undergraduates. Colgate University*, 1994.

[3] B. L. Kurtz *et al.*, "Parallel computing in the undergraduate curriculum," *ACM SIGCSE Bulletin*, vol. 30, no. 1, 1998.

[4] S. K. Prasad *et al.*, "NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing: core topics for undergraduates," in *SIGCSE*, vol. 11, 2011.

[5] C. D. Yu *et al.*, "Performance optimization for the K-Nearest Neighbors kernel on x86 architectures," in *SC 15*, 2015.

[6] D. A. Matthews, "High-performance tensor contraction without transposition," *SIAM J. Sci. Comput.*, 2018.

[7] P. Springer and P. Bientinesi, "Design of a high-performance gemm-like tensor-tensor multiplication," *ACM Trans. Math. Softw.*, 2018.

[8] J. Huang *et al.*, "Strassen's algorithm for tensor contraction," FLAME Working Note #84,, 2017. [Online]. Available: https://arxiv.org/pdf/1704.03092.pdf

[9] J. Huang *et al.*, "Strassen's algorithm reloaded," in *SC 16*, 2016.

[10] J. Huang *et al.*, "Generating families of practical fast matrix multiplication algorithms," in *IPDPS*, 2017.

[11] J. Demmel, "CS267 Lecture 2a: High performance programming on a single processor: Memory hierarchies matrix multiplication automatic performance tuning," 2006. [Online]. Available: https://people.eecs.berkeley.edu/~demmel/cs267_Spr05/Lectures/Lecture02/lecture_02a_AutomaticTuning_jd05.ppt

[12] S. Baden, "CSE260 - Assignment 1: High performance matrix multiplication on a CPU," 2015. [Online]. Available: https://cseweb.ucsd.edu/classes/fa15/cse260-a/static/HW/A1/

[13] K. Goto and R. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Soft.*, 2008.

[14] S. Chellappa *et al.*, *How to Write Fast Numerical Code: A Small Introduction*, 2008. [Online]. Available: https://doi.org/10.1007/978-3-540-88643-3_5

[15] M. Pueschel, "How to write fast numerical code," 2015. [Online]. Available: https://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring15/course.html

[16] J. Bilmes *et al.*, "Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology," in *ICS*, 1997.

[17] M. Lam *et al.*, "The cache performance and optimization of blocked algorithms," in *ASPLOS4*, April 1991.

[18] R. C. Whaley and J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of SC'98*, 1998.

[19] R. A. van de Geijn, "How to optimize matrix multiplication," https://github.com/flame/how-to-optimize-gemm/wiki, 2016.

[20] J. Huang and R. van de Geijn, "BLISlab: A sandbox for optimizing GEMM," FLAME Working Note #80,, 2016. [Online]. Available: http://arxiv.org/pdf/1609.00076v1.pdf

[21] F. Van Zee *et al.*, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Trans. Math. Softw.*, 2015.

[22] F. Van Zee *et al.*, "The BLIS framework: Experiments in portability," *ACM Trans. Math. Softw.*, 2016.

[23] M. Lehn, "GEMM: From pure C to SSE optimized micro kernels," http://apfel.mathematik.uni-ulm.de/~lehn/sghpc/gemm/index.html, 2014.

[24] J. Gilbert, "CS 140 Homework 3: SUMMA matrix multiplication," 2009. [Online]. Available: http://www.cs.ucsb.edu/~gilbert/cs140/old/cs140Win2009/assignments/hw3.pdf

[25] R. Ge, "MSCS6060 Homework 3: SUMMA matrix multiplication," 2014. [Online]. Available: http://www.mscs.mu.edu/~rge/mscs6060/homework-s2014/mpi_matmul/summa.pdf

[26] R. van de Geijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, 1997.

[27] M. Schatz *et al.*, "Parallel matrix multiplication: A systematic journey," *SIAM J. Sci. Comput.*, 2016.

[28] L. Blackford *et al.*, *ScaLAPACK Users' Guide*. SIAM, 1997.

[29] R. A. van de Geijn, *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.

[30] J. Poulson *et al.*, "Elemental: A new framework for distributed memory dense matrix computations," *ACM Trans. Math. Softw.*, 2013.

[31] R. van de Geijn *et al.*, "Lowering barriers into HPC through open education," in *EduHPC 2017*, 2017.

[32] BLAS-like library instantiation software framework (BLIS). `https://github.com/flame/blis`.

[33] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for Fortran usage," *ACM Trans. Math. Soft.*, vol. 5, no. 3, Sept. 1979.

[34] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of FORTRAN basic linear algebra subprograms," *ACM Trans. Math. Soft.*, vol. 14, no. 1, March 1988.

[35] J. Dongarra *et al.*, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Soft.*, 1990.

[36] T. M. Low *et al.*, "Analytical modeling is enough for high-performance BLIS," *ACM Trans. Math. Softw.*, 2016.

[37] F. Van Zee *et al.*, "Implementing high-performance complex matrix multiplication via the 3M and 4M methods," *ACM Trans. Math. Softw.*, 2017.

[38] L. Dagum and R. Menon, "OpenMP: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, 1998.

[39] T. Smith *et al.*, "Anatomy of high-performance many-threaded matrix multiplication," in *IPDPS'2014*, 2014.

[40] W. Gropp *et al.*, *Using MPI*, 1994.

[41] E. Chan *et al.*, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*.