Lowering Barriers into HPC through Open Education

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

ABSTRACT

In this paper, we describe a trilogy of Massive Open Online Courses (MOOCs) that together expose knowledge and skills of fundamental importance to HPC. *Linear Algebra: Foundations to Frontiers* (LAFF) covers topics found in an introductory undergraduate course on linear algebra. It links abstraction in mathematics to abstraction in programming, with many enrichments that connect to HPC. *LAFF-On Programming for Correctness* introduces how to systematically derive programs to be correct. Of importance to HPC is that this methodology yields families of algorithms so that the best one for a given situation can be chosen. *Programming for HPC* (working title) is in the design stage. We envision using a very simple example, matrix-matrix multiplication, to illustrate how to achieve performance on a single core, on multicore and many-core architectures, and on distributed memory computers. These materials lower barriers into HPC by presenting insights, supports, and challenges to novices and HPC experts while scaling access to the world.

CCS CONCEPTS

• **Mathematics of computing → Mathematical software performance**; • **Social and professional topics → Computing education**;

KEYWORDS

high-performance computing, linear algebra, open education

ACM Reference Format:
Robert A. van de Geijn, Jianyu Huang, Margaret E. Myers, Devangi N. Parikh, and Tyler M. Smith . 2017. Lowering Barriers into HPC through Open Education. In *Proceedings of Workshop on Education for High-Performance Computing (EduHPC-17) , Denver, CO, USA, November 2017 (EduHPC-17),* 7 pages.

EduHPC-17, November 2017, Denver, CO, USA

ACM ISBN .

1 INTRODUCTION

When distributed memory parallel computers first became the fastest supercomputers in the world, almost every effort to parallelize and optimize a computational application was a "one-off" customization. These were the early days of exploration to gain the insight that underlies what we call the *Science of High-Performance Computing*. Eventually, this exposed patterns and those patterns then led to conjectures about general principles. Once these principles were further tested, what was an art became a science. Thus, science organized knowledge into a systematic approach. By sharing the method behind the mystique, we can increase the accessibility of the field. Massive Open Online Courses (MOOCs) provide a vehicle to make the world our audience.

Our expertise is in the area of high-performance dense linear algebra software development. In the early 1990s, we made contributions to the distributed memory parallelization of individual operations in this domain. Often, others would approach us with an algorithm that needed to be parallelized and eventually patterns emerged behind the solutions that we found. Some of the key insights gained are:

- When presenting algorithms, abstract away from the details regarding data storage and mapping of data to processing nodes. This led to a new notation for expressing dense linear algebra algorithms that we call the *FLAME notation* [13, 20, 25], illustrated for LU factorization algorithms in Figure 1(left) and 2(left).
- When representing algorithms in code, use Application Programming Interfaces (APIs) – what we now would call domain specific languages, to implement libraries in an existing language. The API should closely mirror the notation with which algorithms are presented, hiding details of data storage and/or distribution, as illustrated in Figures 1(right) and 2(right).
- When optimizing and/or parallelizing, don't automatically start with the legacy solution, which may or may not suit the target situation well. Instead, derive families of algorithms from the specification of the operation and choose the "best" (by some measure) as your point of departure.
- When implementing your software, layer carefully. Leverage abstractions and isolate architecture-specific optimizations.
- When sharing gained insight, use the simplest example that illustrates it. This broadens the audience and helps draw novices into the field.

The first bullets help make knowledge systematic. The last is our guiding principle when sharing the resulting science.

By making HPC accessible, we can open the field so that its practice is not restricted to the "high priests of high performance." MOOCs help draw a broad audience into HPC.

2 THE FLAME NOTATION

Dense linear algebra algorithms are traditionally presented as operations performed on the elements of the arrays that store vectors and matrices. Mapping algorithms to distributed memory architectures added the need to describe the mapping of array elements and operations to processors. These details quickly get in the way of high level thinking, and stand in the way of democratizing HPC. The solution? An alternative notation that abstracts away from these details by closely mirroring how algorithms are naturally explained on, for example, a chalkboard.

Let us examine a concrete example. In a linear algebra course, a student will be taught Gaussian elimination and how it relates to LU factorization. This is typically achieved by linking systems of linear equations to matrices and then to the operations performed with individual entries in those matrices. In a numerical linear algebra course, algorithms for LU factorization are explained by starting with the specification: $A = LU$. Partitioning these matrices yields

$$\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) = \underbrace{\left(\begin{array}{c|c} 1 & 0 \\ \hline l_{21} & L_{22} \end{array}\right)\left(\begin{array}{c|c} \upsilon_{11} & u_{12}^T \\ \hline 0 & U_{22} \end{array}\right)}_{\left(\begin{array}{c|c} \upsilon_{11} & u_{12}^T \\ \hline \upsilon_{11} l_{21} & l_{21}u_{12}^T + L_{22}U_{22} \end{array}\right)}.$$

This then leads to the observation that

- υ_{11} and u_{12}^T respectively equal α_{11} and a_{12}^T;
- $l_{21} := a_{21}/\alpha_{11}$ can overwrite a_{21};
- A_{22} can be updated with the rank-1 update $A_{22} - l_{21}u_{12}^T$; and
- L_{22} and U_{22} are computed as the LU factorization of A_{22}.

The explanation is in terms of submatrices instead of individual entries. On a chalkboard this is accompanied by drawing a square that represents the matrix, and lines that partition this matrix into submatrices.

While this is how the algorithm is elegantly explained, a typical numerical linear algebra text then presents the algorithm by translating the submatrices into index ranges for the parts of the array where the submatrices are stored: $A_{22} - a_{21}a_{12}^T$ in "MATLAB notation" becomes

```
A( j+1:n, j+1:n ) - A( j+1:n, j ) * A( j, j+1:n ),
```

where `j` is the index of the "current column" in the matrix. We argue that there is a disconnect between the notation used to explain how to obtain the algorithm and the notation used to express the algorithm. This disconnect is much more pronounced for so-called *blocked* algorithms that can attain high performance.

The algorithm, expressed with the FLAME notation that was developed in our research and is used in our teaching, is given in Figure 1. This notation is meant to capture how the algorithm progresses through the matrix. We effectively used this notation to bring freshman with no background in linear algebra up to a level where they could get involved in our research on high-performance linear algebra software development. Thus, the notation is valuable when introducing more advanced topics to those who have experience with linear algebra *and* it can be used to first introduce the subject. It becomes core to how our MOOCs enable abstract thinking of value to HPC.

3 A TRILOGY OF MOOCS

We now describe a trilogy of MOOCs, two of which are currently offered on the edX platform and a third one that is in the planning stages. The courses are by no means a sequence: They can be taken individually and/or in any order. Someone with no or limited background in linear algebra would want to start with the first. What all three have in common is that they use linear algebra (matrix computations) as a means by which to illustrate topics of importance to HPC, thus providing an entry path into HPC.

3.1 Linear Algebra: Foundations to Frontiers

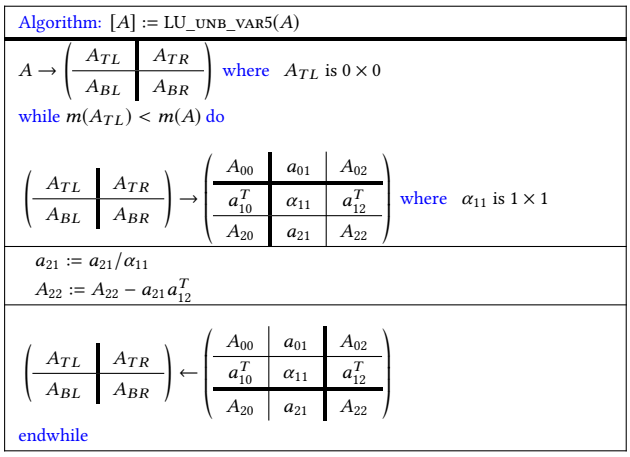
Linear Algebra - Foundations to Frontiers (LAFF) [21] is a MOOC that has been offered regularly on the edX platform starting in Spring 2014. It is a fifteen week course with a content and level that falls somewhere between a first course on matrix computations (what we would consider a "how to" course) and a first course on the theory of linear algebra (which is often a first exposure to proofs in a math department). What is different is that it links abstraction in mathematics (notation and proofs) to abstraction in programming (APIs). It includes enrichments related to HPC, which makes the course of interest to learners with varying levels of background, ranging form high school students to Ph.D.s in related fields. Many use it to review.

3.1.1 Basic idea. The class focuses on linking the mathematics with algorithms and their instantiation in code. Effective reasoning about algorithms requires abstraction. For this reason, the course is structured, and the materials scaffolded, to efficiently lead the learner from the concrete stage of learning to the more abstract stage of learning. Towards this purpose, the layering in the mathematical theory is linked to the layering in the implementation. The notation described in Section 2 supports this.

3.1.2 Links to HPC. Starting in the first week, the course stresses thinking of vectors and matrices in terms of subvectors and submatrices. This is accomplished by carefully scaffolding exercises to lead the learner from concrete examples to abstract notation and thinking. Operations with vector and matrix elements are linked to algorithms cast in terms of subvectors and submatrices, using the notation illustrated in Figure 1(left). This is further reinforced by having the learners implement various operations using the API illustrated in Figure 1(right), with MATLAB Online[1]. The rigid structure of the algorithms and their implementation allows a code skeleton to be generated from a webpage[2] menu illustrated in Figure 3. The code implemented by the learner, written in MATLAB, inherently does not attain high performance. However, the code is

[1] MathWorks gives learners access to a license to MATLAB Online for the duration of the course.

[2] http://www.cs.utexas.edu/users/flame/Spark/




```
function [ A_out ] = LU_unb_var5( A )

  [ ATL, ATR, ...
    ABL, ABR ] = FLA_Part_2x2( A, ...
                               0, 0, 'FLA_TL' );
  while ( size( ATL, 1 ) < size( A, 1 ) )
    [ A00,  a01,     A02,  ...
      a10t, alpha11, a12t, ...
      A20,  a21,     A22 ] = ...
      FLA_Repart_2x2_to_3x3( ATL, ATR, ...
                             ABL, ABR, 1, 1, 'FLA_BR' );
    %------------------------------------------------%
    laff_invscal( alpha11, a21 );
    laff_ger( 1.0, a21, a12t, A22 );
    %------------------------------------------------%
    [ ATL, ATR, ...
      ABL, ABR ] = ...
      FLA_Cont_with_3x3_to_2x2( A00,  a01,     A02,  ...
                                a10t, alpha11, a12t, ...
                                A20,  a21,     A22,  ...
                                'FLA_TL' );
  end
  A_out = [ ATL, ATR
            ABL, ABR ];
end
```

Figure 1: Left: "Right-looking" algorithm (unblocked Invariant 5) for overwriting A with its LU factorization. Right: Corresponding code using the FLAME@lab API for MATLAB.

structured as it would be coded in a higher performing language like C [3].

The students internalize these abstractions in the first five Weeks, during which they are exposed to the fundamental vector operations (scaling, dot product, "axpy"). In Week 2, linear transformations are introduced, which are then linked to linear combinations of vectors and ultimately to matrix-vector multiplication. Along the way, the layering in these mathematical operations is linked to the layering in the implementation: matrix-vector multiplication in terms of dot products of rows with the vector and in terms of axpy operations with the columns of the matrix.

Composition of linear transformations is linked to matrix-matrix multiplication. By Weeks 4 and 5, the learner discovers how this operations, $C = AB$ can be expressed as multiple matrix-vector multiplications (one per column of C and B) or a sequence of rank-1 updates. How this is linked to the Basic Linear Algebra Subprograms (BLAS) interface [9, 10, 17] and impacts performance is the topic of various enrichments in those weeks. This part of the course finishes with an important enrichment on the basic principles behind high-performance implementation of matrix-matrix multiplication, previewing a major topic of the third MOOC discussed in Section 3.3.

A typical introduction to linear algebra starts with solving linear systems. In this course, we reach this topic by Week 6. Vector spaces, low rank approximation and eigenvalue problems (diagonalization) are some of the topics that round out the course. Having instilled the ability to think in terms of submatrices and subvectors, the learner is quickly led from concrete examples to abstract thinking in terms of Gauss transforms, and eventually to LU factorization. Importantly, they are introduced to the algorithm in Figure 1(left) and implement it as in Figure 1(right). Notice again that the algorithm is expressed in terms of basic linear algebra operations: scaling of a vector ($a_{21} := a_{21}/\alpha_{11}$) and rank-1 update ($A_{22} := A_{22} - a_{21}a_{12}^T$. A discussion of a high-performance blocked version of the algorithm is in the enrichment of that week.

The point: within the first six weeks of a first course on linear algebra, the learner is already exposed to abstraction, implementation, and HPC issues.

3.2 LAFF-On: Programming for Correctness

Correctness is of fundamental importance to programming. Many who develop software approach correctness instinctually, building on experience and ironing out the kinks by "debugging". As Dijkstra pointed out in his Turing Award lecture [8]:

> "The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand."

This raises a number of questions: How does one prove a program correct and, more importantly, how does one derive a program hand in hand with its correctness?

For many HPC applications, including dense matrix computations, implementations of key operations are loop-based. Classic techniques developed in the 1960s by Dijkstra and his contemporaries use the concept of a *loop invariant* to prove a loop correct using the *Principle of Mathematical Induction* (PMI). "LAFF-On Programming for Correctness" [22] was developed to teach this and illustrate its importance to programming in general and HPC in particular.

3.2.1 Basic idea. The key to proving a loop-based program correct is the loop invariant. It is a condition that must hold before and

Algorithm: $[A] := \text{LU\_BLK\_VAR2}(A)$

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \quad \text{where} \quad A_{TL} \text{ is } 0 \times 0$$

while $m(A_{TL}) < m(A)$ do

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) \quad \text{where} \quad A_{11} \text{ is } b \times b$$

$$A_{01} := L_{00}^{-1} A_{01}$$
$$A_{11} := A_{11} - A_{10}A_{01}$$
$$A_{11} := \text{LU\_UNB\_VAR5}(A_{11})$$
$$A_{21} := A_{21} - A_{20}A_{01}$$
$$A_{21} := A_{21}U_{11}^{-1}$$

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

```
int LU_blk_var2( FLA_Obj A, int nb_alg )
{
  FLA_Obj ATL,   ATR,       A00, A01, A02,
          ABL,   ABR,       A10, A11, A12,
                            A20, A21, A22;
  int b;

  FLA_Part_2x2( A,     &ATL, &ATR,
                       &ABL, &ABR,      0, 0, FLA_TL );
  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ){
    b = min( FLA_Obj_length( ABR ), nb_alg );
    FLA_Repart_2x2_to_3x3(
        ATL, /**/ ATR,        &A00, /**/ &A01, &A02,
      /* ************* */   /* ****************** */
                              &A10, /**/ &A11, &A12,
        ABL, /**/ ABR,        &A20, /**/ &A21, &A22,
                                    b, b, FLA_BR );
    /*----------------------------------------------*/
    FLA_Trsm( FLA_SIDE_LEFT, FLA_LOWER_TRIANGULAR,
              FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
              FLA_ONE, A00, A01 );
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A10, A01, FLA_ONE, A11 );
    LU_unb_var5( A11 );
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A20, A01, FLA_ONE, A21 );
    FLA_Trsm( FLA_SIDE_LEFT, FLA_UPPER_TRIANGULAR,
              FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
              FLA_ONE, A11, A21 );
    /*----------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
        &ATL, /**/ &ATR,        A00, A01, /**/ A02,
                                A10, A11, /**/ A12,
      /* ************* */   /* ****************** */
        &ABL, /**/ &ABR,        A20, A21, /**/ A22,
                                      FLA_TL );
  }
  return FLA_SUCCESS;
}
```

Figure 2: Left: "Left-looking" blocked algorithm for LU factorization. This algorithm casts most computation in terms of matrix-matrix multiplication (for high performance). It is a good choice for out-of-core computation (where the data resides on disk or some other slow layer of memory.) Right: Its implementation using the FLAME/C API for the C programming language.

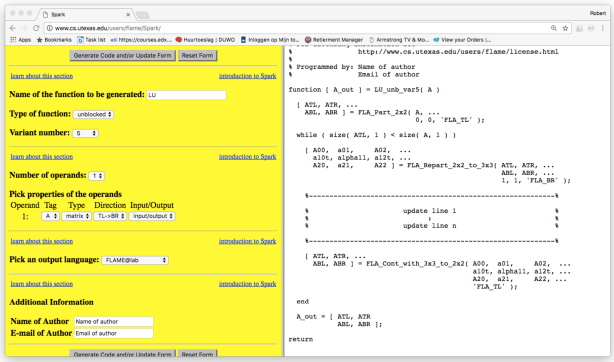


Figure 3: Spark webpage that generates code skeletons (on right) from a menu (on left).

after each iteration of the loop. The fact that it holds before the loop starts is the base case for mathematical induction. If the fact that it holds before any arbitrary iteration implies that it holds after that same iteration can be shown, this corresponds to the inductive in a proof by induction. The PMI then tells us it holds before and after all iterations of the loop. If the loop completes, the loop invariant holds after the loop. If this implies that the desired operation has been computed, the loop is correct.

In the example in Section 2, the loop invariant is given by

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} \{L\backslash U\}_{TL} & U_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} - L_{BL}U_{TR} \end{array}\right)$$
$$\wedge \left(\begin{array}{c|c} \widehat{A}_{TL} = L_{TL}U_{TL} & \widehat{A}_{TR} = L_{TL}U_{TR} \\ \hline \widehat{A}_{BL} = L_{BL}U_{TL} & \end{array}\right)$$

where the $\wedge$ indicates the logical and operator, $\{L\backslash U\}_{TL}$ indicates that unit lower triangular matrix L_{TL} and upper triangular matrix U_{TL} overwrite A_{TL}, and $\widehat{A}$ denotes the original content of A. With this, the correctness proof for algorithm in Figure 1(left) is given in Figure 4. Assertions about the state of the variables are given in the highlighted boxes. Notice that the loop invariant is true at all four places indicated by "Step 4" (details in [23]).

Step	Algorithm: $A := \text{LU\_UNB\_VAR5}(A)$
1a	$A = \widehat{A}$
4	$A \rightarrow \left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$ where A_{TL} is 0×0
2	$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \{L\backslash U\}_{TL} & U_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} - L_{BL}U_{TR} \end{array}\right)$ $\wedge \left(\begin{array}{c\|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL} & \end{array}\right)$
3	while $m(A_{TL}) < m(A)$ do
2,3	$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \{L\backslash U\}_{TL} & U_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} - L_{BL}U_{TR} \end{array}\right)$ $\wedge \left(\begin{array}{c\|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL} & \end{array}\right) \wedge m(A_{TL}) < m(A)$
5a	$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$ where α_{11} is 1×1
6	$\left(\begin{array}{c\|c\|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c\|c\|c} \{L\backslash U\}_{00} & u_{01} & U_{02} \\ \hline l_{10}^T & \widehat{\alpha}_{11} - l_{10}^T u_{01} & \widehat{a}_{12}^T - l_{10}^T U_{02} \\ \hline l_{20} & \widehat{a}_{21} - L_{20}u_{01} & \widehat{A}_{22} - L_{20}U_{02} \end{array}\right) \wedge \cdots$
8	$a_{21} := a_{21}/\alpha_{11}$ $A_{22} := A_{22} - a_{21}a_{12}^T$
5b	$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c\|c\|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$
7	$\left(\begin{array}{c\|c\|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c\|c\|c} \{L\backslash U\}_{00} & u_{01} & U_{02} \\ \hline l_{10}^T & \upsilon_{11} & u_{12}^T \\ \hline l_{20} & l_{21} & \widehat{A}_{22} - L_{20}U_{02} - l_{21}u_{12}^T \end{array}\right) \wedge \cdots$
2	$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \{L\backslash U\}_{TL} & U_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} - L_{BL}U_{TR} \end{array}\right)$ $\wedge \left(\begin{array}{c\|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL} & \end{array}\right)$
	endwhile
2,3	$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \{L\backslash U\}_{TL} & U_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} - L_{BL}U_{TR} \end{array}\right)$ $\wedge \left(\begin{array}{c\|c} L_{TL}U_{TL} = \widehat{A}_{TL} & L_{TL}U_{TR} = \widehat{A}_{TR} \\ \hline L_{BL}U_{TL} = \widehat{A}_{BL} & \end{array}\right) \wedge \neg(m(A_{TL}) < m(A))$
1b	$A = L\backslash U \wedge LU = \widehat{A}$

Figure 4: Derivation of unblocked Variant 5.

Now, the algorithm can be systematically derived from the loop invariant, filling in the "worksheet" in Figure 4 in the order indicated in the column labeled "Step". Better yet: the loop invariant can be systematically derived from the specification of the operation (details in [23]). Best yet: One can systematically derive a number of loop invariants for a given operation, which then yield a family of algorithms. For example, the loop invariant

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} \{L\backslash U\}_{TL} & \widehat{A}_{TR} \\ \hline L_{BL} & \widehat{A}_{BR} \end{array}\right) \wedge \left(\begin{array}{c|c} \widehat{A}_{TL} = L_{TL}U_{TL} & \\ \hline \widehat{A}_{BL} = L_{BL}U_{TL} & \end{array}\right)$$

yields the so-called left-looking blocked algorithm in Figure 2(left). Thus we achieve Dijkstra's goal of deriving the algorithm hand-in-hand with its proof of correctness.

As in LAFF, we use the API for MATLAB[3] illustrated in Figures 1(right) and Figure 2(right) so that the correctness of the algorithm implies the correctness of the implementation.

3.2.2 Links to HPC. Effective mapping of algorithms to architectures starts with choosing or discovering an algorithm that maps well to the given architecture. For dense linear algebra, how much data movement between memory layers is incurred by an algorithm and how much parallelism it exhibits are key to its ability to achieve high performance.

With this in mind, the key insights to which learners are brought to light by this MOOC are:

- For those who have not yet gained experience, it exposes the system behind algorithm development and programming.
- Experienced software developers for HPC often code based on experience (intuition) rather than formal training. The MOOC exposes such people to the formal thinking that underlies their intuition.
- Optimizing for a new architecture often starts with a given implementation. A search for an entirely new algorithm then only commenses when the given implementation cannot be easily "morphed" into one that maps well to the architecture. Formal derivation yields families of algorithms from which the most suited can be chosen. For example, a blocked version of the algorithm in Figure 1(left) maps well to distributed memory architectures [5, 24, 34]. The one given in Figure 2(left) is well-suited for out-of-core computation, where the data resides in a very slow layer of memory [14, 16, 28, 29, 32].
- Abstraction is the key to keeping algorithm development and coding manageable.
- While the course uses MATLAB to illustrate how algorithms can be translated into code, a similar style of coding can target the C programming language (as we do in practice for our libflame dense linear algebra library [3, 36, 37]), as is discussed in enrichments in the course.

3.3 Programming for HPC

Online resources for HPC are an attractive way of scaling up tutorials given by, for example, supercomputing centers. There are free online materials available, including articles [7, 11, 15, 38, 39, 41], wikis [18, 35], free electronic books [40], and MOOCs. Some target foundational topics on HPC [2, 19, 42] while others target HPC for specific domains [26].

We are in the process of designing a third MOOC that exposes the learner to issues that were crucial to our own success in HPC. We envision using a simple example, matrix-matrix multiplication, to illustrate fundamental concerns when optimizing and/or parallelizing scientific computing applications. In addition to the skills that are of importance to HPC that can be demonstrated through such a study, what we hope the learner will take away is a deep understanding of the fact that success in HPC depends on deriving

[3] MathWorks gives learners access to a license to MATLAB Online for the duration of this course as well.

an entire space of solutions for computing a given operation so that a member in that space that has desirable (performance and/or space and/or power) properties can be chosen. This is captured by a quote attributed to Dijkstra:

> "Always design your program as a member of a whole family of programs, including those that are likely to succeed it."

3.3.1 The basic idea. Optimization for HPC platforms starts with optimization for a single core and multicore processor. Multicore processors may be the end target, or they form the building blocks for nodes of a distributed memory architecture when that is the ultimate goal.

A vehicle that is often used to illustrate the basic ideas behind optimizing for a core and multiprocessor is matrix-matrix multiplication (GEMM) [4, 7, 41]. On the other end of the spectrum, parallelizing for a distributed memory architecture is also often illustrated with matrix-matrix multiplication. At all levels, it is how to reduce data movement between memory layers and between nodes that is key to performance.

3.3.2 What we envision. We envision three parts for the proposed MOOC.

- The first part will build on our research on refactoring the GotoBLAS approach to optimizing `gemm` (published in [11], itself a paper that is often used in the classroom), which has yielded the BLAS-like Library Instantiation Software (BLIS) framework [38, 39]. The learner will gain insight into how to optimize for a single core and how to parallelize such an implementation with OpenMP [30]. Related to this, we already support two pedagogical exercises: the "how-to-optimize-gemm" wiki [35]and a sandbox for optimizing GEMM that we call BLISlab [15]. Others have build similar exercises upon the same insights [18]
- Practical distributed memory parallel implementations of GEMM tend to be variants on the Scalable University Matrix Multiplication Algorithm (SUMMA) [1, 33]. It casts communication between nodes in terms of collective communication operations and local computation in terms of a multicore GEMM operation. The second part of the MOOC introduces concepts related to communication on distributed memory architectures: message passing; collective communication operations, algorithms, and cost; the Message-Passing Interface [12, 31] and how to implement collective communication in terms of individual messages. Here we build upon the paper "Collective communication: theory, practice, and experience" [6].
- With how to optimize for an individual node and how to communication between nodes as tools, how to map a SUMMA-like algorithm to a distributed memory architecture is the topic of the third part of the MOOC. That part builds on a recent paper "Parallel Matrix Multiplication: A Systematic Journey" [27].

It is our experience that a remarkable number of issues related to HPC can be demonstrated with this material. Because the example is simple, it is accessible to a broad audience. Importantly, this simple example gives rise to a surprisingly large design space of solutions.

3.3.3 Links to HPC. The links to HPC are self-evident.

3.3.4 Soliciting your input. We have described one possible vision. However, as of the writing of this paper, we are still in the design stage and plan to solicit input from the community regarding both the overall focus of the course as well as specifics.

4 CONCLUSION

As science embraces computation as a third pillar of research, joining theory and experimentation, the democratization of HPC becomes more urgent. There must be materials for professional development that target domain experts who need to acquire deeper knowledge and skills for computing as well as self-taught scientific software developers who have experience but may lack some of the formal foundations. By making materials accessible and interesting at multiple levels, we can target not only that demographic, but also entice novices into the field. MOOCs are an attractive means by which to achieve this because they are flexible (the learner can choose the breadth and depth of knowledge he/she desires to acquire) and they scale to the world. They are a means by which HPC scholars can share his/her experiences with a broad audience.

Acknowledgments

This work was sponsored in part by the National Science Foundation under grant numbers ACI-1148125, ACI-1550493, CCF-1714091, a grant from The University of Texas System, and a gift from MathWorks. edX is a non-profit founded by Harvard University and the Massachusetts Institute of Technology. We thank the rest of the SHPC team (http://shpc.ices.utexas.edu) for their support.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] R. C. Agarwal, F. Gustavson, and M. Zubair. 1994. A high-performance matrix multiplication algorithm on a distributed memory parallel computer using overlapped communication. *IBM Journal of Research and Development* 38, 6 (1994).

[2] Ahmed Shamsul Arefin. 2017. Learn to Use HPC Systems and Supercomputers (Complete Guide). https://www.udemy.com/learn-to-use-hpc-systems-and-supercomputers. (2017). Massive Open Online Course on Udemy.

[3] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. 2005. Representing Linear Algebra Algorithms in Code: The FLAME Application Programming Interfaces. *ACM Trans. Math. Soft.* 31, 1 (March 2005), 27–59. http://doi.acm.org/10.1145/1055531.1055533

[4] Jeff Bilmes, Krste Asanović, Chee whye Chin, and Jim Demmel. 1997. Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology. In *Proceedings of International Conference on Supercomputing*. Vienna, Austria.

[5] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. 1997. *ScaLAPACK Users' Guide*. SIAM.

[6] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. 2007. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience* 19, 13 (2007), 1749–1783.

[7] James Demmel. 2016. U.C. Berkeley CS267: Applications of Parallel Computers. https://people.eecs.berkeley.edu/~demmel/cs267_Spr16. (2016).

[8] Edsger W. Dijkstra. 1972. The humble programmer. (1972). http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF Turing Award lecture.

[9] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. 1990. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.* 16, 1 (March 1990), 1–17.

[10] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. 1988. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.* 14, 1 (March 1988), 1–17.

[11] Kazushige Goto and Robert van de Geijn. 2008a. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Soft.* 34, 3: Article 12, 25 pages (May 2008a).

[12] W. Gropp, E. Lusk, and A. Skjellum. 1994. *Using MPI.* The MIT Press.

[13] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. 2001. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Soft.* 27, 4 (December 2001), 422–455. http://doi.acm.org/10.1145/504210.504213

[14] Brian C. Gunter, Wesley C. Reiley, and Robert A. van de Geijn. 2001. Parallel Out-of-Core Cholesky and QR Factorizations with POOCLAPACK. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society.

[15] Jianyu Huang and Robert A. van de Geijn. 2016. *BLISlab: A Sandbox for Optimizing GEMM.* FLAME Working Note #80, TR-16-13. The University of Texas at Austin, Department of Computer Science. http://arxiv.org/pdf/1609.00076v1.pdf

[16] Ken Klimkowski and Robert van de Geijn. 1995. Anatomy of an out-of-core dense linear solver. In *Proceedings of the International Conference on Parallel Processing 1995*, Vol. III - Algorithms and Applications. 29–33.

[17] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Soft.* 5, 3 (Sept. 1979), 308–323.

[18] Michael Lehn. 2014. GEMM: From Pure C to SSE Optimized Micro Kernels. http://apfel.mathematik.uni-ulm.de/~lehn/sghpc/gemm/index.html. (2014).

[19] Randall J. LeVeque. 2017. High Performance Scientific Computing. http://faculty.washington.edu/rjl/uwhpsc-coursera/. (2017). University of Washington AMath 483/583, Massive Open Online Course on Coursera.

[20] Margaret E. Myers, Pierce M. van de Geijn, and Robert A. van de Geijn. 2015. *Linear Algebra: Foundations to Frontiers - Notes to LAFF With.* ulaff.net.

[21] Margaret E. Myers and Robert A. van de Geijn. 2014. Linear Algebra: Foundations to Frontiers (LAFF). https://www.edx.org/course/linear-algebra-foundations-frontiers-utaustinx-ut-5-05x-0. (2014). Massive Open Online Course on edX.

[22] Margaret E. Myers and Robert A. van de Geijn. 2017. LAFF-On Programming for Correctness. https://www.edx.org/course/laff-programming-correctness-utaustinx-ut-p4c-14-01x. (2017). Massive Open Online Course on edX.

[23] Margaret E. Myers and Robert A. van de Geijn. 2017. *LAFF-On Programming for Correctness.* ulaff.net.

[24] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. 2013. Elemental: A New Framework for Distributed Memory Dense Matrix Computations. *ACM Trans. Math. Softw.* 39, 2, Article 13 (February 2013), 24 pages. https://doi.org/10.1145/2427023.2427030

[25] Enrique S. Quintana, Gregorio Quintana, Xiaobai Sun, and Robert van de Geijn. 2001. A Note On Parallel Matrix Inversion. *SIAM J. Sci. Comput.* 22, 5 (2001), 1762–1771.

[26] Vincent Carey Rafael Irizarry and Michael Love. 2017. High-performance Computing for Reproducible Genomics. https://www.edx.org/course/high-performance-computing-reproducible-harvardx-ph525-6x-0. (2017). Harvard University PH 525, Massive Open Online Course on edX.

[27] Martin D. Schatz, Robert A. van de Geijn, and Jack Poulson. 2016. Parallel Matrix Multiplication: A Systematic Journey. *SIAM J. Sci. Comput.* 38 (2016), C748–C781. Issue 6.

[28] David S. Scott. 1992. Out of core dense solvers on Intel parallel supercomputers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*. 484–487.

[29] David S. Scott. 1993. Parallel I/O and solving out-of-core systems of linear equations. In *Proceedings of the 1993 DAGS/PC Symposium*. Dartmouth Institute for Advanced Graduate Studies, Hanover, NH, 123–130.

[30] Tyler M Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R Hammond, and Field G Van Zee. 2014. Anatomy of high-performance many-threaded matrix multiplication. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 1049–1059.

[31] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. 1996. *MPI: The Complete Reference.* The MIT Press.

[32] Sivan Toledo and Fred G. Gustavson. 1996. The Design and Implementation of SOLAR, a Portable Library for Scalable Out-of-Core Linear Algebra Computation. In *Proceedings of IOPADS '96.*

[33] Robert van de Geijn and Jerrell Watts. 1997. SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience* 9, 4 (April 1997), 255–274.

[34] Robert A. van de Geijn. 1997. *Using PLAPACK: Parallel Linear Algebra Package.* The MIT Press.

[35] Robert A. van de Geijn. 2016. How to Optimize Matrix Multiplication. https://github.com/flame/how-to-optimize-gemm/wiki. (2016).

[36] Field G. Van Zee. 2009. `libflame: The Complete Reference.` `www.lulu.com`.

[37] Field G. Van Zee, Ernie Chan, Robert van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. 2009. The libflame Library for Dense Matrix Computations. *IEEE Computation in Science & Engineering* 11, 6 (2009), 56–62.

[38] Field G. Van Zee, Tyler M. Smith, Bryan Marker, Tze Meng Low, Robert A. van de Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John A. Gunnels, and Lee Killough. 2016. The BLIS Framework: Experiments in Portability. *ACM Trans. Math. Softw.* 42, 2, Article 12 (June 2016), 19 pages. https://doi.org/10.1145/2755561

[39] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* 41, 3, Article 14 (June 2015), 33 pages. https://doi.org/10.1145/2764454

[40] Victor Eijkhout with Robert van de Geijn and Edmond Chow. 2011. *Introduction to High Performance Scientific Computing.* lulu.com. http://www.tacc.utexas.edu/~eijkhout/istc/istc.html.

[41] V. Volkov and J. W. Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.

[42] Rich Vuduc and Catherine Gamboa. 2017. High Performance Computing. https://www.udacity.com/course/high-performance-computing--ud281. (2017). Georgia Tech CS 6220, Massive Open Online Course on Udacity.