# Developing Linear Algebra Algorithms: A Collection of Class Projects

**STATUS AT END OF SEMESTER**

FLAME Working Note #3

John A. Gunnels
Robert A. van de Geijn

Department of Computer Sciences
The University of Texas
Austin, TX 78712
{gunnels,rvdg}@cs.utexas.edu

May 31, 2001

## Abstract

In this document we present a new approach to developing sequential and parallel dense linear algebra libraries. Given a linear algebra operation, we demonstrate how formal techniques can be used to derive a family of algorithms. Due to the systematic approach used, correctness of the algorithms can be asserted. Next, we introduce a library of routines that hides the manipulation of indices and allows the code to mirror the algorithms as they are naturally presented. The idea is that by having the code mirror the algorithm, the opportunity for introducing indexing errors is minimized. Thus, the correctness assertions regarding the algorithms carry over to the implementations.

The philosophy behind the approach is that one should start by systematically deriving the algorithms. The recipe for derivation is given in Chapter 2. Moreover, this derivation should be carefully documented. To facilitate this, we provide a set of LaTeX macros, given in an appendix. Once one or more algorithms have been developed, they are translated to implementations using a library of C routines, as part of the Formal Linear Algebra Methods Environment (FLAME). This library allows the code to look much like the algorithms as written using LaTeX. For all examples in the report we demonstrate that high performance can be attained on an Intel Pentium (R) III processor.

We illustrate the techniques with a large number of case studies, most of which were carried out by teams of computer science undergraduate students as part of a class taught in Spring 2001 at UT-Austin titled *High-Performance Parallel Algorithms.* The names of the members of the teams are given as the authors of the chapter on the operation assigned to that team. Thus we show that the approach makes the development and implementation of high-performance sequential and parallel algorithms for dense linear algebra operations accessible to novices.

It is important to realize that this document is meant to capture the progress of the project during a single semester. Thus, the document is incomplete in many ways. For example, the review of the literature is sparse at best. Many sections and chapters are missing or incomplete. Typographical errors are scattered throughout. For each operation, only a few algorithms are derived and implemented. The performance results are limited to a single architecture. While high-performance parallel implementations were also created using our Parallel Linear Algebra Package (PLAPACK), the discussion of these implementations did not make it into the document. It is our hope that there is value in this document despite these shortcomings.

# Contents

# Chapter 1

# Introduction

When considering the unmanageable complexity of computer systems, Dijkstra recently made the following observations [10]:

(i) When exhaustive testing is impossible –i.e., almost always– our trust can only be based on proof (be it mechanized or not).
(ii) A program for which it is not clear why we should trust it, is of dubious value.
(iii) A program should be structured in such a way that the argument for its correctness is feasible and not unnecessarily laborious.
(iv) Given the proof, deriving a program justified by it, is much easier than, given the program, constructing a proof justifying it.

Our Formal Linear Algebra Methods Environment (FLAME) is an attempt to address these concerns when coding linear algebra libraries [18].

The core undergraduate curriculum in computer science department often includes at least one course that focuses on the formal derivation and verification of algorithms [13]. Many of us in scientific computing may have, at some point in time, hastily dismissed this approach, arguing that this is all very nice for small, simple algorithms, but an academic exercise hardly applicable in "our world." Since it is often the case that our work involves libraries comprised of hundreds of thousands or even millions of lines of code, the knee-jerk reaction that this approach is much too cumbersome to take seriously is understandable. Furthermore, the momentum of established practices and "traditional wisdom" do little if anything to dissuade one from this line of reasoning. Yet, as the result of our search for superior methods for designing and constructing high-performance parallel linear algebra libraries, we have come to the conclusion that it is *only* through the systematic approach offered by formal methods that we will be able to deliver reliable, maintainable, flexible, yet highly efficient matrix libraries even in the relatively well-understood area of (sequential and parallel) dense linear algebra.

While some would immediately draw the conclusion that a change to a more modern programming language like C++ is at least highly desirable, if not a necessary precursor to writing elegant code, the fact is that most applications that call packages like the Linear Algebra PACKage (LAPACK) [4] and the Scalable Linear Algebra PACKage ScaLAPACK [8, 7] are still written in Fortran and/or C. Interfacing such an application with a library written in C++ presents certain complications. However, during the mid-nineties, the Message-Passing Interface (MPI) introduced to the scientific computing community a programming model, object-based programming, that possesses many of the advantages typically associated with the intelligent use of an object-oriented language [33]. Using objects (e.g. communicators in MPI) to encapsulate data structures and hide complexity, a much cleaner approach to coding can be achieved. Our own work on the Parallel Linear Algebra PACKage (PLAPACK) borrowed from this approach in order to hide details of data distribution and data mapping in the realm of parallel linear algebra libraries [3, 5, 16, 29, 30, 35, 37]. The primary concept also germane to this paper is that PLAPACK raises the level of abstraction at which one programs so that indexing is essentially removed from the code, allowing the routine to reflect the algorithm as it is naturally presented in a classroom setting. Since our initial work on PLAPACK, we have experimented with similar interfaces in such seemingly disparate contexts as (parallel) out-of-core linear algebra packages [19, 31, 32] and a low-level implementation of the sequential Basic Linear Algebra Subprograms (BLAS) [11, 12, 17, 28].

Our Formal Linear Algebra Methods Environment (FLAME) is the latest step in the evolution of these systems. It facilitates the use of a programming style that is equally applicable to everything from out-of-core, parallel systems to single-processor systems where cache-management is of paramount concern.

Over the last seven or eight years it has become apparent that what makes our task of library development more manageable is this systematic approach to deriving algorithms coupled with the abstractions we use to make our code reflect the algorithms thus produced. Further, from these experiences we can confidently state that this approach to programming greatly reduces the complexity of the resultant code and does not sacrifice high performance in order to do so.

Indeed, the formal techniques that we may have dismissed as merely academic or impractical make this possible, as we attempt to illustrate in this document.

## 1.1 The case for formal derivation

Ideally, an implementation should clearly reflect the algorithm as it is presented in a classroom setting. Additionally, some of the derivation of the algorithm should be apparent in the code and different variants of an algorithm should be recognizable as slight perturbations to an algorithmic "skeleton" or base code. indexbase code If the code is just a mechanically-realizable, straightforward translation of this algorithmic expression, there should be no opportunity for the introduction of logical errors or coding bugs. (Note: while we will frequently refer to translations from algorithms to code as being mechanical or automatic, this process is currently performed by hand.) Presumably, it should be possible to prove the algorithms correct, thus ensuring that the code is correct.

Typically, it is difficult to prove code correct precisely because one is not certain that the code truly mirrors the algorithm. With our approach, the chasm is largely bridged by the simple yet crucial fact that some very simple syntactic rewrite rules can produce the code from an algorithm expressed as one might in a classroom, using mathematical formulas and stylized matrix depictions. Since we can prove the correctness of the algorithm we wish to employ (the proof is generally constructive in nature, but this is of little consequence) and because the correctness of the translation from algorithm to code is at least as reliable as compiler technology, the complexity of the task at hand is greatly ameliorated. By assuming that components adhere to explicit "contractual obligations" [2], the algorithmic proof requires little alteration in order to be applicable to the code. In the case of a library constructed entirely through the methodology presented here, these components would be composed in like manner so as to make this task manageable. This is largely due to the fact that the approach presented here leads to a software architecture layered in such a way so as to require one to rely on the correctness of a very small number of base-level modules. Since those units are small, their correctness can be established through the application of standard formal methods. It is true that, in practice, one must accept that an application will need to interface with other libraries (for example, the vendor-supplied BLAS) that are not built in a "proof-friendly" format. However, it may still be possible to establish the correctness of a program if one is careful to impose minimal obligations on these, presumably time-tested and well-documented, pieces of code.

It should be noted that the "correctness" discussed so far does not address issues of numerical stability. We make no claim regarding the stability of the resulting algorithm.

Having said this, we will clarify through a simple example in Sectionsec:example. But first, we review commonly used matrix and vector notation. For those for whom linear algebra is not second nature, the most basic of operations are reviewed in A

## 1.2 Notation

A *(column) vector*, $x$, is the $n$-tuple of real or complex numbers

$$x = \begin{pmatrix} \chi_1 \\ \chi_2 \\ \vdots \\ \chi_n \end{pmatrix}$$

Here $\chi_i$ are called the *components* of vector $x$. We will denote the set of all vectors with real components $\mathbf{R}^n$ and with complex components $\mathbf{C}^n$.

A *row* vector, $x^T$, is the $n$-tuple of real or complex numbers

$$x^T = \left( \begin{array}{c|c|c|c} \chi_1 & \chi_2 & \cdots & \chi_n \end{array} \right)$$

Here $x^T$ indicates a transposed (column) vector. (We will always assume vectors are column vectors, unless transposed like this, or explicitly noted.) More about transposition next.

An $m \times n$ matrix, $A$, is the array

$$A = \begin{pmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1n} \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2n} \\ \vdots & \vdots & & \vdots \\ \alpha_{m1} & \alpha_{m2} & \cdots & \alpha_{mn} \end{pmatrix}$$

with $m$ rows and $n$ columns. The $(i, j)$ component, or *element*, of $A$ refers to $\alpha_{ij}$, which may be real or complex. The numbers $m$ and $n$ are the dimensions of $A$. If $m = n$ then the matrix is said to be **square**. Otherwise, it is said to be **rectangular**.

Notice that we use the convention introduced in [34] of using Greek letters for real or complex numbers, lowercase italicized letters for vectors, and uppercase italicized letters for matrices.

Frequently, we will wish to partition a matrix into blocks. For example, if $A$ is an $m \times n$ matrix, it can be partitioned into a $M \times N$ matrix of submatrices like

$$A = \left( \begin{array}{c|c|c|c} A_{11} & A_{12} & \cdots & A_{1N} \\ \hline A_{21} & A_{22} & \cdots & A_{2N} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{M1} & A_{M2} & \cdots & A_{MN} \end{array} \right)$$

where $A_{ij}$ is an $m_i \times n_j$ matrix, with $\sum_{i=1}^{N} n_i = n$ and $\sum_{i=1}^{M} m_i = m$.

Similarly, a vector can be partitioned into subvectors. For example, if $x$ is a vector of length $n$, we may wish to partition like

$$x = \left( \begin{array}{c} x_1 \\ \hline x_2 \\ \hline \vdots \\ \hline x_N \end{array} \right)$$

where $x_i$ is an vector of length $n_i$, with $\sum_{i=1}^{N} n_i = n$.

Additional notation and basic linear algebra operations are reviewed in Appendix A.

## 1.3 A motivating example: LU factorization

We illustrate our approach by considering LU factorization without pivoting. Given an $n \times n$ matrix $A$ we wish to compute an $n \times n$ lower triangular matrix $L$ with unit main diagonal and an $n \times n$ upper triangular matrix $U$ so that $A = LU$. The original matrix $A$ is overwritten by $L$ and $U$ in the process.

The usual derivation of an algorithm for the LU factorization proceeds as follows: Partition

$$A = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right), \quad L = \left(\begin{array}{c|c} 1 & 0 \\ \hline l_{21} & L_{22} \end{array}\right), \quad \text{and} \quad U = \left(\begin{array}{c|c} \upsilon & u_{12}^T \\ \hline 0 & U_{22} \end{array}\right)$$

Now $A = LU$ translates to

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

so the following equalities hold:

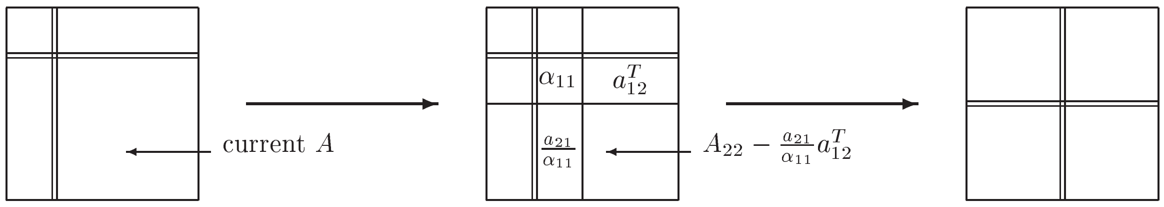
$$\left(\begin{array}{c|c} \alpha_{11} = \upsilon & a_{12}^T = u_{12}^T \\ \hline a_{21} = \upsilon_{11} l_{21} & a_{22} = l_{21}u_{12}^T + L_{22}U_{22} \end{array}\right)$$

Finally, we arrive at the following algorithm

- Overwrite $\alpha_{11}$ and $a_{12}^T$ with $\upsilon_{11}$ and $u_{12}^T$, respectively (no-op).
- Update $a_{21} \leftarrow l_{21} = a_{21}/\upsilon_{11}$.
- Update $A_{22} \leftarrow A_{22} - l_{21}u_{12}^T$.
- Recursively factor $A_{22} \rightarrow L_{22}U_{22}$.

While the algorithm is formulated as tail-recursive, it is usually implemented as a loop.

When presented in a classroom setting, the explanation is typically accompanied by the following progression of pictures:


current $A$
$\alpha_{11}$
$a_{12}^T$
$\frac{a_{21}}{\alpha_{11}}$
$A_{22} - \frac{a_{21}}{\alpha_{11}} a_{12}^T$

with an indication that at a given stage the current active part of the matrix resides in the lower-right quadrant of the left picture. Next, the different parts to be updated are identified and the updates given (middle picture). Finally, the boundary indicating how far the computation has progressed is moved forward (right picture).

It is this progression depicted in the pictures that we try to capture both in the derivation and the implementation of the algorithm. We claim that the discussed algorithm for LU factorization is naturally given by the algorithm in Fig. 1.1. A code for implementing the algorithm using FLAME is given in Fig. 1.2.

The code can be obtained from the algorithm essentially via textual substitution. Notice that the calls to `FLA_Inv_scal` and `FLA_Ger` implement division of a vector by a scalar and rank-1 update of a matrix, respectively. The formatting of the code is a deliberate attempt to capture the partitioning and repartitioning in the algorithm.

## 1.4 Performance experiments

For each matrix operation discussed in this document, we report performance on an Intel Pentium III (650 MHz) processor with 16 Kbyte L1 data cache and a 256 Kbyte L2 cache running RedHat 7.1 Linux. All floating point calculations were performed in double precision (64-bit) arithmetic.

The FLAME routines that perform level 1 BLAS (vector-vector operations) and level 2 BLAS (matrix-vector operations) interface to a standard BLAS library. For performance experiments, the ATLAS library implementation was used [36]. In particular, the prebuilt version in

```
partition $A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$ where $A_{TL}$ is $0 \times 0$
do until $A_{BR}$ is $0 \times 0$
  repartition $\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & a_{01} & A_{02} \\ \hline\hline a_{10}^T & \alpha_{11} & a_{02}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$ where $\alpha_{11}$ is a scalar
  $\alpha_{11} \leftarrow \upsilon_{11} = \alpha_{11}$
  $a_{12}^T \leftarrow u_{12}^T = a_{12}^T$
  $a_{21} \leftarrow l_{21} = a_{21}/\alpha_{11}$
  $A_{22} \leftarrow A_{22} - l_{21} u_{12}^T$
  continue with $\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{02}^T \\ \hline\hline A_{20} & a_{21} & A_{22} \end{array}\right)$
enddo
```

Figure 1.1: An unblocked (rank-1 update based) eager LU factorization algorithm (without pivoting)

```
#include "FLAME.h"

void FLA_LU_nopivot_eager_level2( FLA_Obj A )
{
  FLA_Obj     ATL, ATR,   A00,  a01,     A02,
              ABL, ABR,   a10t, alpha11, a12t,
                          A20,  a21,     A22;

  FLA_Part_2x2( A,  &ATL, /**/ &ATR,
                   /* ************** */
                     &ABL, /**/ &ABR,
            /* with */ 0, /* by */ 0, /* submatrix */ FLA_TL );

  while ( min( FLA_Obj_length( ABR ), FLA_Obj_width( ABR ) ) != 0 ){

    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,       &A00,  /**/ &a01,     &A02,
                        /* ************* */   /* ************************* */
                                /**/            &a10t, /**/ &alpha11, &a12t,
                           ABL, /**/ ABR,       &A20,  /**/ &a21,     &A22,
            /* with */ 1, /* by */ 1, /* alpha11 split from */ FLA_BR );

    /* ********************************************************************* */

    FLA_Inv_scal( alpha11, a21 );             /* a21 <- a21 / alpha11        */

    FLA_Ger( MINUS_ONE, a21, a12t, A22 );     /* A22 <- A22 - a21 * a12t     */

    /* ********************************************************************* */

    FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,      A00,  a01,     /**/ A02,
                                     /**/            a10t, alpha11, /**/ a12t,
                            /* ************* */ /* *********************** */
                              &ABL, /**/ &ABR,      A20,  a21,     /**/ A22,
            /* with alpha11 added to submatrix */ FLA_TL );
  }
}
```

Figure 1.2: An unblocked (rank-1 update based) eager LU factorization implementation (without pivoting) using FLAME.

`atlas3.2.0_Linux_SSE1256.tgz`

available from

`http://www.netlib.org/atlas/archives/linux/`

was used. In many of the performance graphs for level 3 BLAS (matrix-matrix) operations, we report performance by a reference implementation. This reference implementation is the one provided by ATLAS.

Notice that our implementations heavily rely on a high-performance matrix-matrix multiplication kernel. For our performance experiments, the `DGEMM` level 3 BLAS kernel for matrix-matrix multiplication was used. We use two different implementations:

- the implementation provided by ATLAS, and
- our own ITXGEMM implementation [17].

The latter implementation, which yields better performance for most matrix sizes, is further explained in Chapter 4.

We report performance as the rate at which the computation was performed in MFLOPS/sec. (millions of floating point operations per second). More precisely, if $C$ equals the number of floating point operations required to complete the computation, and $t$ equals the time required, the rate in MFLOPS/sec. is given by

$$\frac{C}{t} \times 10^{-6}$$

## 1.5 Overview

Our methodology for developing high-performance linear algebra algorithms is introduced in Chapter 2. The application programming interface (API) is introduced in Chapter 3 as a set of C routines. In Chapter 4, we discuss how high performance can be attained by a matrix-matrix multiplication kernel. In Chapters 5–16 we report a large number of case studies that demonstrate the methodology for developing algorithms. They also show how blocked algorithms can be used to formulate the algorithms in terms of matrix-matrix multiplications, which allows the high performance attained by this kernel to be exploited. It should be noted that the idea of implementing this particular set of operations in terms of matrix-matrix multiplication has been studied extensively elsewhere [8, 9, 20, 21, 24, 25, 26, 36]. In particular, one may recognize them as special cases of the level 3 BLAS.

## 1.6 Availability

Information related to FLAME is available at

`http://www.cs.utexas.edu/users/flame/`

Codes discussed in this document can be found at

`http://www.cs.utexas.edu/users/flame/materials/`

The LaTeXcommands and environments used to typeset many of the formulas and algorithms in this document, which are described in Appendix B, can be found at

`http://www.cs.utexas.edu/users/flame/LaTeX/`

# Chapter 2

# Formal Derivation

In this chapter, we briefly review general techniques for the derivation of algorithms. We relate these techniques to the iterative algorithms encountered in subsequent chapters. Finally, we give a systematic recipe for deriving linear algebra algorithms. The recipe is illustrated for the triangular matrix-matrix multiplication $B \leftarrow LB$.

## 2.1 The correctness of loops

In a standard text [14] used to teach discrete mathematics to undergraduates in computer science we find the following material:

> We prefer to write a while loop using the syntax
>
> **do** $B \rightarrow S$ **od**
>
> where Boolean expression $B$ is called the *guard* and statement $S$ is called the *repetend*.
>
> [The l]oop is executed as follows: If $B$ is *false*, then execution of the loop terminates; otherwise $S$ is executed and the process is repeated.
>
> Each execution of repetend $S$ is called an *iteration*. Thus, if $B$ is initially *false*, then 0 iterations occur.
>
> [$\cdots$]
>
> We now state and prove the fundamental invariance theorem for loops. This theorem refers to an assertion $P$ that holds before and after each iteration (provided it holds before the first). Such a predicate is called a *loop invariant*.
>
> (12.43) **Fundamental invariance theorem.** Suppose
>
> - $\{P \wedge B\}S\{P\}$ holds – i.e. execution of $S$ begun in a state in which $P$ and $B$ are *true* terminates with $P$ *true* – and
> - $\{P\}$ **do** $B \rightarrow S$ **od** *true* – i.e. execution of the loop begun in a state in which $P$ is *true* terminates.
>
> Then $\{P\}$ **do** $B \rightarrow S$ **od** $\{P \wedge \neg B\}$ holds. [In other words, if the loop is entered in a state where $P$ is *true*, it will complete in a state where $P$ is *true* and guard $B$ is *false*.]

The text proceeds to prove this theorem using mathematical induction.

Let us translate the above into our setting, which will accommodate linear algebra algorithms. Consider the loop

```
do until $\neg B$
  $S$
enddo
```

where $B$ is some condition and $S$ is the body of the loop. The above theorem says that *if*

- The loop is entered in a state where some condition $P$ holds, and
- for each iteration, $P$ holds at the top of the loop, and
- the body of the loop $S$ has the property that if it is executed starting in a state where $P$ holds it completes in a state where $P$ holds.

*then* the loop will complete in a state where conditions $P$ and $\neg B$ both hold.

A method that formally derives a loop (i.e., iterative implementation) approaches the problem of determining the body of the loop as follows:

- First, one must determine conditions $B$ and $P$.
- Next, the body $S$ should be developed so that it maintains condition $P$ while making progress towards completing the iterative process (eventually $B$ should become *false*).

As a consequence of the Fundamental Invariance Theorem, this approach implies correctness of the loop.

What we show in the remainder of this chapter, and the subsequent case studies in the remainder of this book, is that for a large class of dense linear algebra algorithms

- There is a systematic way of determining different conditions $P$ that allow us develop loops to compute the result of a given linear algebra operation.
- This in turn yields different algorithms for computing the operation.

## 2.2 A recipe for deriving linear algebra algorithms

### 2.2.1 A typical operation

A typical linear algebra matrix-matrix operation involves up to three operands: matrices $A$, $B$, and $C$:

$$C \leftarrow \text{op}(A, B, C)$$

**Example (LTRMM)** A lower-triangular matrix-matrix multiplication (LTRMM) can be expressed as

$$B \leftarrow LB$$

where $B$ is an $m \times n$ matrix and $L$ is a $m \times m$ lower triangular matrix. Notice that the purpose of the game is to overwrite $B$ with the results without requiring a work array in which to compute $LB$.

### 2.2.2 determining possible loop invariants

1. **Temporarily replace $C$ by $D$.** Since $C$ appears both on the left and the right of the operation to be performed, we temporarily replace one of the instances by a new operand, $D$.

   **Example (LTRMM)** $B \leftarrow LB$ is replaced by $D = LB$. Notice that the arrow indicates that the operand is overwritten with the result. We now replace this by an equality, since the purpose of the game will be to determine equalities that must hold as the computation unfolds.

2. **Pick an Operand and Partition** The first step is to pick an operand and partition[1] it in a meaningful way:

[1]Note: we provide some useful LaTeXmacros for partitioning matrices in Appendix C.

- If the operand that is picked has a triangular storage structure, the partitioning should be into four quadrants, so that the quadrant that contains the block that is not used for storing the matrix can be identified.

**Example (LTRMM)** Pick $L$ for partitioning:

$$L \rightarrow \left( \begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right)$$

where $L_{TL}$ is a square block, say of size $k \times k$.

- If the operand has no special structure, it is typically partitioned into two submatrices, either horizontally or vertically.

**Example (LTRMM)** Pick $B$ for partitioning:

$$B \rightarrow \left( \begin{array}{c||c} B_L & B_R \end{array} \right)$$

where $B_L$ has $k$ columns. Note: this second partitioning does not necessarily lead to the same set of loop invariants as the partition of $L$ above. Indeed, it doesn't.

3. **Partition the other operands conformal to the first one:** Given that the first operand has been partitioned, the other operands should be partitioned *conformally* to ensure that blocked multiplication of the submatrices makes sense.

**Example (LTRMM)** Let us concentrate of the case where $L$ has been partitioned like

$$L \rightarrow \left( \begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right)$$

When considering $D = LB$, we notice that $D$ and $B$ must be partitioned by rows:

$$D \rightarrow \left( \begin{array}{c} D_T \\ \hline\hline D_B \end{array} \right), \quad \text{and} \quad B \rightarrow \left( \begin{array}{c} B_T \\ \hline\hline B_B \end{array} \right)$$

where $D_T$ and $B_T$ have $k$ rows.

4. **Rewrite the operation using the partitionings:** Next, plug the partitioned matrices into the operation $D = \text{op}(A, B, C)$.

**Example (LTRMM)** In the example $D = LB$, where $D$, $L$, and $B$ have been partitioned as described, this yields

$$\left( \begin{array}{c} D_T \\ \hline\hline D_B \end{array} \right) = \left( \begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \left( \begin{array}{c} B_T \\ \hline\hline B_B \end{array} \right)$$

5. **Perform blocked matrix-matrix multiplications and additions:** Now that the operation has been expressed with blocked matrices, we perform the given operations.

**Example (LTRMM)** In our lower triangular matrix-matrix multiplication example we get

$$\left( \begin{array}{c} D_T \\ \hline\hline D_B \end{array} \right) = \left( \begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \left( \begin{array}{c} B_T \\ \hline\hline B_B \end{array} \right) = \left( \begin{array}{c} L_{TL}B_T \\ \hline\hline L_{BL}B_T + L_{BR}B_B \end{array} \right)$$

6. **Determine equalities that must hold:** Once the operations have been performed with the submatrices, equalities that must hold can be determined.

**Example (LTRMM)** In our lower triangular matrix-matrix multiplication example we determined that

$$\left(\frac{D_T}{\overline{D_B}}\right) = \left(\frac{L_{TL}B_T}{\overline{L_{BL}B_T + L_{BR}B_B}}\right)$$

Thus, we conclude that the following equalities must hold:

$$\begin{aligned} D_T &= L_{TL}B_T \\ D_B &= L_{BL}B_T + L_{BR}B_B \end{aligned}$$

7. **Determine possible contents of $D$:** Next, we determine possible contents of $D$ under the assumption that some of the computations that appear in the equalities have occurred. One approach to this is to enumerate all the major operations that must be formed. At a given stage of the computation, each of these computations either has or has not occurred. By considering all possible combinations, one can enumerate essentially all possible conditions.

**Example (LTRMM)** See Fig. 2.1.

8. **Eliminate unreasonable conditions:** Once possible contents of $D$ have been enumerated, some of the conditions can typically be eliminated since they do not lead to reasonable loops.

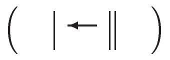
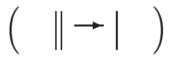
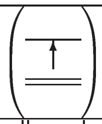
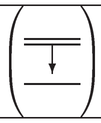
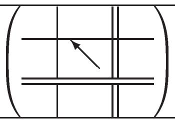
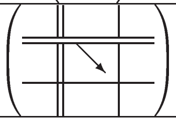
   There are two reasons for rejecting a condition:

   (a) We want to derive a loop that maintains the condition while making progress towards the desired result. We need to be able to achieve that condition before we enter the loop. Furthermore, upon leaving the loop, the desired result should have been computed. If the condition doesn't allow this, then the condition is rejected.

   (b) The result overwrites part of one of the input operands before that part of the input operand is no longer needed.

   We will illustrate how the first approach can be used to reject possible conditions for the symmetric matrix multiply in Chapter 6. In Fig. 2.2, we use the second approach to reject a number of conditions for the triangular matrix-matrix multiplication example.

9. **Determine the direction of the computation:** The condition that determines the contents of $D$ indicates a direction in which the computation naturally proceeds. Frequently encountered directions are

| | | | | | |
|---|---|---|---|---|---|
| left |  | right |  | up |  |
| down |  | left-up |  | right-down |  |

**Example (LTRMM)** Consider the condition that currently $D$ contains $\left(\frac{\star}{\overline{L_{BL}B_T + L_{BR}B_B}}\right)$. In order to move the boundary that indicates how far the computation has proceeded, that boundary must be moved up. Thus, this algorithm naturally moves through matrices $D$ and $B$ in the "up" direction.

**Example (LTRMM)** In our lower triangular matrix-matrix multiplication example we have determined that

$$\begin{aligned} D_T &= L_{TL}B_T \\ D_B &= L_{BL}B_T + L_{BR}B_B \end{aligned}$$

The major operations to be performed are $L_{TL}B_T$, $L_{BL}B_T$, and $L_{BR}B_B$. Each of these either has or has not already been computed, leading to the $2^3 = 8$ possible conditions tabulated below. In the table, a $\star$ indicates the indicated part of $D$ has yet to be computed.

| Computed? | | | $D$ contains |
|---|---|---|---|
| $L_{TL}B_T$ | $L_{BL}B_T$ | $L_{BR}B_B$ | |
| NO | NO | NO | $\left(\begin{array}{c} \star \\ \hline\hline \star \end{array}\right)$ |
| YES | NO | NO | $\left(\begin{array}{c} L_{TL}B_T \\ \hline\hline \star \end{array}\right)$ |
| NO | YES | NO | $\left(\begin{array}{c} \star \\ \hline\hline L_{BL}B_T \end{array}\right)$ |
| YES | YES | NO | $\left(\begin{array}{c} L_{TL}B_T \\ \hline\hline L_{BL}B_T \end{array}\right)$ |
| NO | NO | YES | $\left(\begin{array}{c} \star \\ \hline\hline L_{BR}B_B \end{array}\right)$ |
| YES | NO | YES | $\left(\begin{array}{c} L_{TL}B_T \\ \hline\hline L_{BR}B_B \end{array}\right)$ |
| NO | YES | YES | $\left(\begin{array}{c} \star \\ \hline\hline L_{BL}B_T + L_{BR}B_B \end{array}\right)$ |
| YES | YES | YES | $\left(\begin{array}{c} L_{TL}B_T \\ \hline\hline L_{BL}B_T + L_{BR}B_B \end{array}\right)$ |

Figure 2.1: Step 7 for the LTRMM example.

**Example (LTRMM)** In the following table, we again list possible contents of $D$. This time, we comment on each possibility.

| $D$ contains | Comments | Viable? |
|---|---|---|
| $\left(\frac{\star}{\star}\right)$ | This condition indicates no progress has been made. | NO |
| $\left(\frac{L_{TL}B_T}{\star}\right)$ | Since $B_T$ is to be overwritten by $D_T$, this condition is not feasible since $B_T$ is still needed for the computation $L_{BL}B_T$. | NO |
| $\left(\frac{\star}{L_{BL}B_T}\right)$ | Since $B_T$ is to be overwritten by $D_T$, this condition is not feasible since $B_T$ is still needed for the computation $L_{BL}B_T$. | NO |
| $\left(\frac{L_{TL}B_T}{L_{BL}B_T}\right)$ | Since $B_B$ is to be overwritten by $D_B$, this condition is not feasible since $B_B$ is still needed for the computation $L_{BR}B_B$. | NO |
| $\left(\frac{\star}{L_{BR}B_B}\right)$ | | YES |
| $\left(\frac{L_{TL}B_T}{L_{BR}B_B}\right)$ | Since $B_T$ is to be overwritten by $D_T$, this condition is not feasible since $B_T$ is still needed for the computation $L_{BL}B_T$. | NO |
| $\left(\frac{\star}{L_{BL}B_T + L_{BR}B_B}\right)$ | | YES |
| $\left(\frac{L_{TL}B_T}{L_{BL}B_T + L_{BR}B_B}\right)$ | This condition indicates that the computation has completed. | NO |

Considering the comments, only two viable conditions are left, the ones for which there are no comments.

Figure 2.2: Step 8 for the LTRMM example.

10. **Repartition the matrices:** In order to expose what elements must be updated to make progress, $D$ is repartitioned. Similarly, the other matrices must be repartitioned conformally to expose submatrices needed to update $D$.

> **Example (LTRMM)** Since the computation moves through $D$ in the "up" direction, we repartition $D$ like
>
> $$\textbf{repartition } \left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline d_1^T \\ \hline\hline D_2 \end{array}\right) \textbf{ where } d_1^T \textbf{ is a row}$$
>
> Similarly, we must repartition the other matrices:
>
> $$\textbf{repartition } \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline\hline B_2 \end{array}\right) \textbf{ where } b_1^T \textbf{ is a row}$$
>
> $$\textbf{repartition } \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22} \end{array}\right) \textbf{ where } \lambda_{11} \textbf{ is a scalar}$$
>
> Notice that the double lines have meaning:
>
> $$\begin{array}{l} D_T = \left(\begin{array}{c} D_0 \\ \hline d_1^T \end{array}\right) \\ \hline D_B = \quad D_2 \end{array} \qquad \begin{array}{l} B_T = \left(\begin{array}{c} B_0 \\ \hline b_1^T \end{array}\right) \\ \hline B_B = \quad B_2 \end{array} \qquad \begin{array}{l||l} L_{TL} = \left(\begin{array}{c|c} L_{00} & 0 \\ \hline l_{10}^T & \lambda_{11} \end{array}\right) & \\ \hline L_{BL} = \left(\begin{array}{c|c} L_{20} & l_{21} \end{array}\right) & L_{BR} = L_{22} \end{array}$$

11. **Determine what is currently in the matrix:** Given the repartitionings, we determine what is currently in matrix D.

> **Example (LTRMM)** $D$ currently contains
>
> $$\begin{aligned} \left(\begin{array}{c} \star \\ \hline\hline L_{BL}B_T + L_{BR}B_B \end{array}\right) &= \left(\begin{array}{c} \star \\ \hline\hline \left(\begin{array}{c|c} L_{20} & l_{21} \end{array}\right) \left(\begin{array}{c} B_0 \\ \hline b_1^T \end{array}\right) + L_{22}B_2 \end{array}\right) \\ &= \left(\begin{array}{c} \star \\ \hline\hline L_{20}B_0 + l_{21}b_1^T + L_{22}B_2 \end{array}\right) \end{aligned}$$

12. **Determine what needs to be in the matrix after the boundary shifts:** Notice that once the boundary shifts, the partitionings of the matrices indicate different submatrices of those matrices. To maintain the condition, an update to the contents of $D$ is required.

> **Example (LTRMM)** After the boundaries shift
>
> $$\begin{array}{l} D_T = \quad D_0 \\ \hline D_B = \left(\begin{array}{c} d_1^T \\ \hline D_2 \end{array}\right) \end{array} \qquad \begin{array}{l} B_T = \quad B_0 \\ \hline B_B = \left(\begin{array}{c} b_1^T \\ \hline B_2 \end{array}\right) \end{array} \qquad \begin{array}{l||l} L_{TL} = \quad L_{00} & \\ \hline L_{BL} = \left(\begin{array}{c} l_{10}^T \\ \hline L_{20} \end{array}\right) & L_{BR} = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right) \end{array}$$
>
> and thus $D$ must hold
>
> $$\begin{aligned} \left(\begin{array}{c} \star \\ \hline\hline L_{BL}B_T + L_{BR}B_B \end{array}\right) &= \left(\begin{array}{c} \star \\ \hline\hline \left(\begin{array}{c} l_{10}^T \\ \hline L_{20} \end{array}\right) B_0 + \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right) \left(\begin{array}{c} b_1^T \\ \hline B_2 \end{array}\right) \end{array}\right) \\ &= \left(\begin{array}{c} \star \\ \hline\hline \left(\begin{array}{c} l_{10}^T B_0 + \lambda_{11} b_1^T \\ \hline L_{20}B_0 + l_{21}b_1^T + L_{22}B_2 \end{array}\right) \end{array}\right) \end{aligned}$$

13. **Determine what update must occur:** The known contents of $D$ before the shift of the boundaries and the desired contents of $D$ after the shift of the boundaries determines the update that must occur.

**Example (LTRMM)** The contents of $D$ must change as follows

$$\left(\begin{array}{c} \left(\begin{array}{c} \star \\ \hline \star \end{array}\right) \\ \hline\hline L_{20}B_0 + l_{21}b_1^T + L_{22}B_2 \end{array}\right) \rightarrow \left(\begin{array}{c} \star \\ \hline\hline \left(\begin{array}{c} l_{10}^T B_0 + \lambda_{11} b_1^T \\ \hline L_{20}B_0 + l_{21}b_1^T + L_{22}B_2 \end{array}\right) \end{array}\right)$$

14. **State the algorithm:** At this point, the partitioning and repartition of the matrices have been derived, as have the steps required to maintain the desired condition. Thus, the algorithm can be given.

**Example (LTRMM)** See Fig. 2.3.

**Example (LTRMM)** One algorithm for the lower triangular matrix-matrix multiplication. Notice that we overwrite $B$ with the result.

```
partition $B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$ where $B_B$ has 0 rows
partition $L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$ where $L_{BR}$ is $0 \times 0$
do until $L_{TL}$ is $0 \times 0$
  repartition $\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline\hline B_2 \end{array}\right)$ where $b_1^T$ is a row
  repartition $\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22} \end{array}\right)$ where $\lambda_{11}$ is a scalar
  ------------------------------------------------
  $b_1^T \leftarrow \lambda_{11} b_1^T$
  $b_1^T \leftarrow l_{10}^T B_0 + b_1^T$
  ------------------------------------------------
  continue with $\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right)$
  continue with $\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline\hline b_1^T \\ \hline B_2 \end{array}\right)$
enddo
```

Figure 2.3: Step 14 for the LTRMM example.

15. **Classify the algorithm:** We classify algorithms by

- the direction in which they move, and
- how aggressively they use and/or update data in the matrices.

While intuitively the classification is consistent, the different categories have slightly different meaning depending on whether we categorize with respect to an input or an output parameter. In particular:

Let $X$ be the operand with respect to which we will categorize the algorithm. The following table explains the different categories:

**right/left-moving algorithms:** Consider the current partitioning on operand in a right- or left-moving algorithm w.r.t. $X$: $\left( X_L \,\Big\|\, X_R \right)$. Then we will use the following categorization of algorithms w.r.t. $X$:

| right-moving algorithm | |
|---|---|
| Lazy | The entries in $X_L$ have been completely used and/or updated. The entries in $X_R$ has not been touched (used or updated). |
| Eager | The entries in $X_L$ have been completely used and/or updated. The entries in $X_R$ have been updated as much as is possible without completing another column of $X$. |

| left-moving algorithm | |
|---|---|
| Lazy | The entries in $X_R$ have been completely used and/or updated. The entries in $X_L$ has not been touched (used or updated). |
| Eager | The entries in $X_R$ have been completely used and/or updated. The entries in $X_L$ have been updated as much as is possible without completing another column of $X$. |

**down/up-moving algorithms:** Consider the current partitioning on operand in a down- or up-moving algorithm w.r.t. $X$: $\left( \frac{X_T}{X_B} \right)$. Then we will use the following categorization of algorithms w.r.t. $X$:

| down-moving algorithm | |
|---|---|
| Lazy | The entries in $X_T$ have been completely used and/or updated. The entries in $X_B$ has not been touched (used or updated). |
| Eager | The entries in $X_T$ have been completely used and/or updated. The entries in $X_B$ have been updated as much as is possible without completing another row of $X$. |

| up-moving algorithm | |
|---|---|
| Lazy | The entries in $X_B$ have been completely used and/or updated. The entries in $X_T$ has not been touched (used or updated). |
| Eager | The entries in $X_B$ have been completely used and/or updated. The entries in $X_T$ have been updated as much as is possible without completing another row of $X$. |

**down-right/up-left-moving algorithms:** Consider the current partitioning of an operand in a right-down- or left-up-moving algorithm w.r.t. $X$: $\left( \begin{array}{c||c} X_{TL} & X_{TR} \\ \hline\hline X_{BL} & X_{BR} \end{array} \right)$. Then we will use the following categorization of algorithms w.r.t. $X$:

| right-down-moving algorithm | |
|---|---|
| Lazy | The entries in $X_{TL}$ have been completely used and/or updated. The entries in $X_{TR}$, $X_{BL}$, and $X_{BR}$ have not been touched (used or updated). |
| Row-lazy | The entries in $X_{TL}$ and $X_{TR}$ have been completely used and/or updated. The entries in $X_{BL}$ and $X_{BR}$ have not been touched (used or updated). |
| Column-lazy | The entries in $X_{TL}$ and $X_{BL}$ have been completely used and/or updated. The entries in $X_{TR}$ and $X_{BR}$ have not been touched (used or updated). |
| Both-lazy | The entries in $X_{TL}$, $X_{TR}$, and $X_{BL}$ have been completely used and/or updated. The entries in $X_{BR}$ have not been touched (used or updated). |
| Eager | The entries in $X_{TL}$, $X_{TR}$, and $X_{BL}$ have been completely used and/or updated. The entries in $X_{BR}$ have been updated as much as is possible without completing another row and column of $X$. |

| left-up-moving algorithm | |
|---|---|
| Lazy | The entries in $X_{BR}$ have been completely used and/or updated. The entries in $X_{TR}$, $X_{BL}$, and $X_{TL}$ have not been touched (used or updated). |
| Row-lazy | The entries in $X_{BL}$ and $X_{BR}$ have been completely used and/or updated. The entries in $X_{TL}$ and $X_{TR}$ have not been touched (used or updated). |
| Column-lazy | The entries in $X_{TR}$ and $X_{BR}$ have been completely used and/or updated. The entries in $X_{TL}$ and $X_{BL}$ have not been touched (used or updated). |
| Both-lazy | The entries in $X_{BR}$, $X_{TR}$, and $X_{BL}$ have been completely used and/or updated. The entries in $X_{TL}$ have not been touched (used or updated). |
| Eager | The entries in $X_{BR}$, $X_{TR}$, and $X_{BL}$ have been completely used and/or updated. The entries in $X_{TL}$ have been updated as much as is possible without completing another row and column of $X$. |

**Example (LTRMM)** Consider the example that we have been using throughout this chapter. Using our categorization, we see that the algorithm that corresponds to the condition that currently $D$ contains $\left(\begin{array}{c} \star \\ \hline\hline L_{BL}B_T + L_{BR}B_B \end{array}\right)$ is up-moving and row-lazy with respect to matrix $L$ since data in $L_{BL}$ and $L_{BR}$ will not be required for further computation. It is also lazy with respect to matrix $D$, since $D_L$ has been completely computed.

The algorithm that corresponds to the the condition that currently $D$ contains $\left(\begin{array}{c} \star \\ \hline\hline L_{BR}B_B \end{array}\right)$ is also up-moving. However, it is lazy with respect to matrix $L$ since only the data in $L_{BR}$ will not be needed for further computation. It is also lazy with respect to matrix $B$, since $B_B$ will not be needed for further computation.

# Chapter 3

# Coding Linear Algebra Algorithms

In this chapter we introduce a set of library routines that will allow us to capture in code linear algebra algorithms as they are naturally presented, for example in a classroom setting. The idea is that by making the code look as much like the algorithm in Fig. 1.1 the opportunity for the introduction of bugs is minimized. Readers familiar with MPI [15, 33] and/or our own PLAPACK [35] will recognize the programming style as being very similar to that used by those interfaces.

## 3.1 initializing and finalizing FLAME

Before using the FLAME environment one must initialize with a call to

```
void FLA_Init( )
```
**Purpose:** Initialize FLAME.

If no more FLAME calls are to be made, the environment is exited by calling

```
void FLA_Finalize ( )
```
**Purpose:** Finalize FLAME.

Since an application may wish to query whether the environment has already been initialized, we provide the inquiry routine

```
int FLA_Initialized ( )
```
**Purpose:** Check if FLAME is initialized.

return value `TRUE` if FLAME is already initialized
`FALSE` otherwise

## 3.2 Creating an object

Notice that there the following attributed describe a matrix as it is stored in the memory of a computer:

- the datatype of the entries in the matrix, e.g., `double` or `float`,
- $m$ and $n$, the row and column dimensions of the matrix,
- the address where the data is stored, and
- the mapping that described how the two dimensional array is mapped to memory.

For now, we will assume that a matrix is stored using column-major ordering. Thus, the mapping to memory is described by a leading dimension that indicates the number of units through which one must stride in memory to get from one element in a row of the matrix to the next element in that row. The following call creates an object that describes a matrix and creates space to store the entries in the matrix:

```
void FLA_Obj_create ( int datatype, int m, int n, FLA_Obj *matrix )
```

**Purpose:** Create an object that describes an $m \times n$ matrix as well as associated storage.

| | |
|---|---|
| `datatype` | datatype of matrix |
| `m, n` | row dimensions of matrix |
| matrix | object that describes the matrix |

Notice that the leading dimension of the array that is used to store the actual matrix is itself determined inside of this call.

Valid datatype value include

`FLA_INT`, `FLA_DOUBLE`, `FLA_FLOAT`, `FLA_DOUBLE_COMPLEX`, and `FLA_COMPLEX`

for the obvious datatypes that are commonly encountered. Additional datatype may be added at a future stage.

Sometimes it will be handy to create an object without storage attached. This allows a matrix that has already been stored in a conventional two-dimensional array to be attached to an object. The following call creates such an object:

```
void FLA_Obj_create_without_buffer
          ( int datatype, int m, int n, FLA_Obj *matrix )
```

**Purpose:** Create an object that describes an $m \times n$ matrix without associated storage.

| | |
|---|---|
| `datatype` | datatype of matrix |
| `m, n` | dimensions of matrix |
| `matrix` | address of object that will describe the matrix |

If an object has been created without storage attached, an existing two-dimensional array can be attached by calling

```
void FLA_Obj_attach_buffer ( void *buff, int ldim, FLA_Obj matrix )
```

**Purpose:** Attach an existing buffer that holds a matrix stored in column-major order with leading dimension `ldim` to the object `matrix`.

| | |
|---|---|
| `buff` | address of where buffer exists |
| `ldim` | leading dimension of array |
| `matrix` | object that describes the matrix |

FLAME treats vectors as special cases of matrices, either as a $n \times 1$ matrix or an $1 \times n$ matrix. Thus, to create an object for a vector $x$ of length $n$ either of the following calls will suffice:

`FLA_Obj_create( FLA_DOUBLE, n, 1, &x )`, or
`FLA_Obj_create( FLA_DOUBLE, 1, n, &x )`,

where `x` has been declared as a `FLA_Obj` and `n` is an integer variable with value $n$.

Similarly, FLAME treats scalars as a $1 \times 1$ matrix. Thus, to create a object for a scalar $\alpha$ the following call is made:

`FLA_Obj_create( FLA_DOUBLE, 1, 1, &alpha )`

where `alpha` has been declared as a `FLA_Obj`. A number of scalars occur frequently and are therefore predefined by FLAME: `MINUS_ONE`, `ZERO`, and `ONE`. .

Often it is useful to create a matrix that has the same datatype and dimensions as a given matrix. For this we provide the call

```
void FLA_Obj_create_conf_to ( int trans, FLA_Obj old, FLA_Obj *matrix )
```

**Purpose:** Like `FLA_Obj_create` except that it creates an object with same datatype and dimensions as `old`, transposing if desired.

| | |
|---|---|
| `trans` | indicates whether to transpose |
| `old` | original object |
| `matrix` | new object |

Valid values for `trans` include `FLA_NO_TRANSPOSE` , `FLA_TRANSPOSE`, and `FLA_CONJUGATE_TRANSPOSE`. If `trans` equals `FLA_NO_TRANSPOSE`, the new object has the same dimensions as `old`. Otherwise, it has the same dimensions as the transpose of `old`.

### 3.2.1 Object destruction

If an object was created with `FLA_Obj_create` or `FLA_Obj_create_conf_to` a call to b `FLA_Obj_free` is required to ensure that all space associated with the object is properly released:

```
void FLA_Obj_free ( FLA_Obj *obj )
```

**Purpose:** Free all space allocated to store data associated with `obj`.

| | |
|---|---|
| `obj` | object that describes the object |

### 3.2.2 Inquiry routines

In order to be able to work with the raw data, a number of inquiry routines can be used to access information about a matrix (or vector or scalar). To extract the datatype and row and column dimensions of the matrix FLAME provides:

```
int FLA_Obj_datatype ( FLA_Obj matrix )
int FLA_Obj_length   ( FLA_Obj matrix )
int FLA_Obj_width    ( FLA_Obj matrix )
```

**Purpose:** Extract datatype, row, or column dimension of matrix, respectively.

| | |
|---|---|
| `matrix` | object that describes the matrix |
| return value | datatype, row, or column dimension of matrix, respectively |

To extract the address of the array that stores the matrix and the leading dimension of that array FLAME provides:

```
void *FLA_Obj_buffer ( FLA_Obj matrix )
int FLA_Obj_ldim  ( FLA_Obj matrix )
```

**Purpose:** Extract the address and leading dimension of the matrix, respectively.

| | |
|---|---|
| `matrix` | object that describes the matrix |
| return value | address and leading dimension of matrix, respectively |

An example of how to use this information to implement a simple matrix-vector multiplication is given in Fig. 3.1. To understand the code one must understand that element $\alpha_{ij}$ of matrix $A$ is stored in `buff_A[j*ldim_A+i]`, which conforms to column-major order. Similarly, $\chi_j$, the $j$th element of $x$, and $\eta_i$, the $i$th element of $y$, are stored in `buff_x[j*inc_x]` and `buff_y[i*inc_y]`, respectively. (Here indexing starts at zero.)

```
#include "FLA.h"

void FLA_simple_mv_mult( FLA_Obj A, FLA_Obj x, FLA_Obj y )
{
  int
    datatype_A,    m_A, n_A, ldim_A,    m_x, n_y, inc_x,    m_y, n_y, inc_y;

  datatype_A = FLA_Obj_datatype( A );
  m_A        = FLA_Obj_length( A );
  n_A        = FLA_Obj_width ( A );
  ldim_A     = FLA_Obj_ldim  ( A );

  m_x        = FLA_Obj_length( x );
  n_x        = FLA_Obj_width ( x );
  m_y        = FLA_Obj_length( y );
  n_y        = FLA_Obj_width ( y );

  if ( m_x == 1 ) {
    m_x  = n_x;
    inc_x = FLA_Obj_ldim( x );
  }
  else inc_x = 1;

  if ( m_y == 1 ) {
    m_y  = n_y;
    inc_y = FLA_Obj_ldim( y );
  }
  else inc_y = 1;

  if ( datatype_A == FLA_DOUBLE ){
    double
      *buff_A, *buff_x, *buff_y;

    buff_A = ( double * ) FLA_Obj_datatype( A );
    buff_x = ( double * ) FLA_Obj_datatype( x );
    buff_y = ( double * ) FLA_Obj_datatype( y );

    for ( i=0; i<m_A; i++ ) buff_y[ i*inc_y ] = 0;

    for ( j=0; j<n_A; j++ )
      for ( i=0; i<m_A; i++ )
        buff_y[ i*inc_y ] += buff_A[ j*ldim_A+i ] * buff_x[ j*inc_x ];
  }
  else FLA_Abort( "datatype not yet supported", __LINE__, __FILE__ );
}
```

Figure 3.1: A simple matrix-vector multiplication routine.

### 3.2.3 Setting and extracting the contents

```
void FLA_Obj_set_contents     ( int trans, int m, int n, void *A, int ldA,+ &\\
                                FLA_Obj matrix )
void FLA_Obj_axpy_to_contents ( int trans, void *alpha, int m, int n, void *A,
                                int ldA, FLA_Obj matrix )
```

**Purpose:** Set the contents of the matrix object to those in `m` × `n` matrix `A` with leading dimension `ldA`. For the second call, $\alpha$`A` is added to the current contents of `matrix`.

| | |
|---|---|
| `trans` | indicates whether to transpose data |
| `alpha` | scaling factor |
| `m, n` | dimensions of `A` |
| `A` | array with data to be entered in matrix |
| `ldim` | leading dimension of `A` |
| `matrix` | object that describes the matrix |

Here consistent means that the datatype of `ALPHA` and `A` must match that of the object `MATRIX`. Valid values for `trans` include `FLA_NO_TRANSPOSE`, `FLA_TRANSPOSE`, and `FLA_CONJUGATE_TRANSPOSE`. If `trans` equals `FLA_NO_TRANSPOSE`, $m$ and $n$ must equal the dimensions of `matrix`, respectively. Otherwise, they must equal the dimensions of the transpose of `matrix` and the data is transposed as it is entered in or added to `matrix`.

Similarly FLAME provides the following calls to extract the contents of an object:

```
void FLA_Obj_get_contents       ( int trans, FLA_Obj matrix,
                                  int m, int n, void A, int ldA )
void FLA_Obj_axpy_from_contents ( int trans, void *alpha, FLA_Obj matrix,
                                  int m, int n, void A, int ldA )
```

**Purpose:** Get the contents from the matrix object and store in `m` × `n` matrix `A` with leading dimension `ldA`. For the second call, $\alpha$ times the contents of `matrix` are added to the current contents of `A`.

| | |
|---|---|
| `trans` | indicates whether data is to be transposed |
| `alpha` | scaling factor |
| `matrix` | object that describes the matrix |
| `m, n` | dimensions of `A` |
| `A` | array with data to be entered in matrix |
| `ldim` | leading dimension of `A` |

## 3.3 A simple driver: matrix-vector multiplication

In Figure 3.2 we show a sample main program that uses most of the calls discussed so far.

**line 1** FLAME program files start by including the `FLAME.h` header file.

**line 5–6** FLAME objects `A`, `x`, and `y`, which will hold matrix $A$ and vectors $x$ and $y$, are declared to be of type `FLA_Obj`.

**line 10** Before any calls to FLAME routines can be made, the environment must be initialized by a call to `FLA_Init`.

**line 12–13** In our example, the user inputs the row and column dimension of matrix $A$.

**line 15–17** Descriptors are created for $A$, $x$, and $y$.

**line 19–20** A routine to be described next is used to fill $A$ and $x$ with values.

**line 22** Compute $y = Ax$ using the FLAME matrix-vector multiply routine `FLA_Gemv` to be described later.

**line 24–26** Print out the contents of $A$, $x$, and $y$. For each element the C print format `"%lf "` is used to print the contents as long floating point numbers. The calling sequence for `FLA_Obj_show` is given later in this chapter.

```
 1  #include "FLAME.h"
 2
 3  main()
 4  {
 5    FLA_Obj
 6      A, x, y;
 7    int
 8      m, n;
 9
10    FLA_Init( );
11
12    printf( "enter matrix dimensions m and n:" );
13    scanf( "%d%d", &m, &n );
14
15    FLA_Obj_create( FLA_DOUBLE, m, n, &A );
16    FLA_Obj_create( FLA_DOUBLE, m, 1, &y );
17    FLA_Obj_create( FLA_DOUBLE, n, 1, &x );
18
19    fill_matrix( A );
20    fill_matrix( x );
21
22    FLA_Gemv( FLA_NO_TRANSPOSE, ONE, A, x, ZERO, y );
23
24    FLA_Obj_show( "A = [", A, "%lf ", "]" );
25    FLA_Obj_show( "x = [", x, "%lf ", "]" );
26    FLA_Obj_show( "y = [", y, "%lf ", "]" );
27
28    FLA_Obj_free( &A );
29    FLA_Obj_free( &y );
30    FLA_Obj_free( &x );
31
32    FLA_Finalize( );
33  }
```

Figure 3.2: A simple C driver for matrix-vector multiplication.

**line 28** After the FLAME environment has finished it is finalized by a call to `FLA_Finalize`.

A sample routine for filling `A` and `x` with data is given in Fig. 3.3.

## 3.4 Views

Notice that in Fig. 2.3 became obvious that in stating a linear algorithm one frequently must partition a matrix, $A$, like

$$\textbf{partition} A \rightarrow \left( \begin{array}{c||c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right) \textbf{ where } \; A_{TL} \textbf{ is } m_b \times n_b$$

The primary mechanism used by our coding approach to hide complicated indexing is the notion of a *view*, which is simply a reference into an existing matrix or vector. Given that $A$ is a descriptor of a matrix, the following call creates descriptors of the four quadrants:

```
void FLA_Part_2x2 ( FLA_Obj A, FLA_Obj *ATL, FLA_Obj *ATR,
                               FLA_Obj *ABL, FLA_Obj *ABR,
                         int mb, int nb, int quadrant )
```

**Purpose:** Partition matrix $A$ into four quadrants where the quadrant indicated by `quadrant` is `mb` $\times$ `nb`

| | |
|---|---|
| `A` | matrix to be partitioned |
| `mb, nb` | row and column dimensions of quadrant indicated by `quadrant` |
| `quadrant` | quadrant for which dimensions are given in `mb` and `nb` |
| ATL-ABR | views of TL, TR, BL, and BR quadrants |

```
#include "FLAME.h"

#define BUFFER( i, j ) buff[ (j)*lda + (i) ]

void fill_matrix( FLA_Obj A )
{
  int datatype, m, n, lda;

  datatype = FLA_Obj_datatype( A );
  m        = FLA_Obj_length( A );
  n        = FLA_Obj_width ( A );
  lda      = FLA_Obj_ldim  ( A );

  if ( datatype == FLA_DOUBLE ){
    double *buff;
    int    i, j;

    buff = ( double * ) FLA_Obj_buffer( A );

    for ( j=0; j<n; j++ )
      for ( i=0; i<m; i++ )
        BUFFER( i,j ) = i+j*0.01;
  }
  else FLA_Abort( "Datatype not yet supported", __LINE__, __FILE__ );
}
```

Figure 3.3: A simple routine for filling a matrix

Here `quadrant` can take on the values `FLA_TL`, `FLA_TR`, `FLA_BL`, and `FLA_BR` to indicate that `mb` and `nb` indicate the dimensions of the Top-Left, Top-Right, Bottom-Left, or Bottom-Right quadrant, respectively.

Also from Fig. 2.3, we notice that it is useful to be able to take a $2\times 2$ partitioning of a given matrix $A$ and repartition this so that submatrices can be identified that need to be updated and/or used for computation:

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

Given that `ATL`, `ATR`, `ABL`, and `ABR` were the result of a call to `FLA_Part_2x2`, we would like create new views for this $3 \times 3$ partitioning from this $2 \times 2$ partitioning. To support this, we introduce the call

```
void FLA_Repart_from_2x2_to_3x3
       ( FLA_Obj ATL, FLA_Obj ATR,   FLA_Obj *A00, FLA_Obj *A01, FLA_Obj *A02,
                                     FLA_Obj *A10, FLA_Obj *A11, FLA_Obj *A12,
         FLA_Obj ABL, FLA_Obj ABR,   FLA_Obj *A20, FLA_Obj *A21, FLA_Obj *A22,
         int mb, int nb, int quadrant )
```

**Purpose:** Repartition a $2\times 2$ partitioning of matrix $A$ into a $3\times 3$ partitioning where `mb`$\times$`nb` submatrix $A_{11}$ is split from the quadrant indicated by `quadrant`.

| | |
|---|---|
| `ATL-ABR` | views of TL, TR, BL, and BR quadrants |
| `mb, nb` | row and column dimensions of $A_{11}$ |
| `quadrant` | quadrant from which $A_{11}$ is partitioned |
| `A00-A22` | views of $A_{00}$–$A_{22}$ |

Here `quadrant` can again take on the values `FLA_TL`, `FLA_TR`, `FLA_BL`, and `FLA_BR` to indicate that `mb` and `nb` submatrix `A11` is split from submatrix `ATL`, `ATR`, `ABL`, or `ABR`, respectively.

In order to update the partitioning of $A$ into the four quadrants, we need to be able to update the descriptions of $A_{TL}$, $A_{TR}$, $A_{BL}$, and $A_{BR}$:

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

This update of the views is accomplished by a call to

```
void FLA_Cont_with_3x3_to_2x2
      ( FLA_Obj *ATL, FLA_Obj *ATR,   FLA_Obj A00, FLA_Obj A01, FLA_Obj A02,
                                      FLA_Obj A10, FLA_Obj A11, FLA_Obj A12,
        FLA_Obj ABL, FLA_Obj ABR,     FLA_Obj A20, FLA_Obj A21, FLA_Obj A22,
        int quadrant )
```

**Purpose:** Update the $2 \times 2$ partitioning of matrix $A$ by moving the boundaries so that $A_{11}$ is added to the quadrant indicated by `quadrant`.

| | |
|---|---|
| `ATL-ABR` | views of TL, TR, BL, and BR quadrants |
| `A00-A22` | views of $A_{00}$–$A_{22}$ |
| `quadrant` | quadrant to which $A_{11}$ is to be added |

This time the value of `quadrant` (`FLA_TL`, `FLA_TR`, `FLA_BL`, or `FLA_BR`) indicates to which quadrant submatrix `A11` is to be added.

We will see in subsequent chapters that we frequently will want to create a $2 \times 1$ partitioning of a given matrix $A$:

$$\textbf{partition } A \rightarrow \left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \textbf{ where } A_T \textbf{ has } m_b \textbf{ rows}$$

For this we introduce the call

```
void FLA_Part_2x1 ( FLA_Obj A, FLA_Obj *AT,
                               FLA_Obj *AB, int mb, int side )
```

**Purpose:** Partition matrix $A$ into a top and bottom side where the side indicated by `side` has `mb` rows.

| | |
|---|---|
| `A` | matrix to be partitioned |
| `mb` | row dimension of side indicated by `side` |
| `side` | side for which row dimension is given |
| `AT, AB` | view of Top and Bottom part |

Here `side` can take on the values `FLA_TOP` or `FLA_BOTTOM` to indicate that `mb` indicates the row dimension of $A_T$ or $A_B$, respectively.

Given that matrix $A$ is already partitioned like

$$\left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right)$$

a repartitioning like

$$\textbf{repartition } \left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline\hline A_1 \\ \hline A_2 \end{array}\right) \textbf{ where } A_1 \textbf{ has } m_b \textbf{ rows}$$

is accomplished by the call

```
void FLA_Repart_from_2x1_to_3x1 ( FLA_Obj AT,   FLA_Obj *A0,
                                                FLA_Obj *A1,
                                  FLA_Obj AB,   FLA_Obj *A2,
                                  int mb, int side )
```

**Purpose:** Repartition a $2 \times 1$ partitioning of matrix $A$ into a $3 \times 1$ partitioning where submatrix $A_1$ with `mb` rows is split from the side indicated by `side`.

| | |
|---|---|
| `AT, AB` | views of Top and Bottom sides |
| `mb` | row dimension of $A_1$ |
| `side` | side from which $A_1$ is partitioned |
| `A0-A2` | views of $A_0$–$A_2$ |

Here side can take on the values FLA_TOP or FLA_Bottom to indicate that mb submatrix $A_1$ is partitioned from $A_T$ or $A_B$, respectively.

Given a $3 \times 1$ partitioning of a given matrix $A$, we may wish to update a $2 \times 1$ partitioning by adding $A_1$ to either $A_T$ or $A_B$:

$$\textbf{continue with} \quad \left( \frac{A_T}{\overline{A_B}} \right) \leftarrow \left( \begin{array}{c} A_0 \\ \hline A_1 \\ \hline\hline A_2 \end{array} \right)$$

For this FLAME provides the call

```
void FLA_Cont_with_3x1_to_2x1 ( FLA_Obj *AT,  FLA_Obj A0,
                                              FLA_Obj A1,
                                FLA_Obj *AB,  FLA_Obj A2,
                                int side )
```

**Purpose:** Update the $2 \times 1$ partitioning of matrix $A$ by moving the boundaries so that $A_1$ is added to the side indicated by side.

| | |
|---|---|
| AT, AB | views of Top and Bottom sides |
| A0-A2 | views of $A_0$–$A_2$ |
| side | side from which $A_1$ is partitioned |

Now side indicates whether $A_1$ is to be added to $A_T$ or $A_B$.

Similarly, we may wish to create a $1 \times 2$ partitioning of a given matrix $A$:

$$\textbf{partition } A \rightarrow \left( \frac{A_T}{\overline{A_B}} \right) \textbf{ where } \; A_T \textbf{ has } n \textbf{ rows}$$

For this we introduce the call

```
void FLA_Part_1x2 ( FLA_Obj A, FLA_Obj *AT, FLA_Obj *AB,
                    int nb, int side )
```

**Purpose:** Partition matrix $A$ into a left and right side where the side indicated by side has nb columns

| | |
|---|---|
| A | matrix to be partitioned |
| nb | column dimension of side indicated by side |
| side | side for which column dimension is given |
| AL, AR | view of Left and Right part |

Here side can take on the values FLA_LEFT or FLA_RIGHT to indicate that nb equals the column dimension of $A_L$ or $A_R$, respectively.

Given that matrix $A$ is already partitioned like

$$\left( \; A_L \; \| \; A_R \; \right)$$

a repartitioning like

$$\textbf{repartition} \quad \left( \frac{A_T}{\overline{A_B}} \right) \rightarrow \left( \begin{array}{c} A_0 \\ \hline\hline A_1 \\ \hline A_2 \end{array} \right) \textbf{ where } \; A_1 \textbf{ has } n \textbf{ rows}$$

is accomplished by the call

```
void FLA_Repart_from_1x2_to_1x3
       ( FLA_Obj AL, FLA_Obj AR,     FLA_Obj *A0, FLA_Obj *A1, FLA_Obj *A2,
         int nb, int side )
```

**Purpose:** Repartition a $1 \times 2$ partitioning of matrix $A$ into a $1 \times 3$ partitioning where submatrix $A_1$ with `nb` columns is split from the side indicated by `side`.

| | |
|---|---|
| `AL, AR` | views of Left and Right sides |
| `A0-A2` | views of $A_0$–$A_2$ |
| `nb` | column dimension of $A_1$ |
| `side` | side from which $A_1$ is partitioned |

Now `side` indicates whether $A_1$ is partitioned from $A_L$ or $A_R$.

Given a $1 \times 3$ partitioning of a given matrix $A$, updating a $1 \times 2$ partitioning by adding $A_1$ to either $A_L$ or $A_R$,

$$\left( \begin{array}{c|c} A_L \| A_R \end{array} \right) \rightarrow \left( A_0 \,\|\, A_1 \,|\, A_2 \right)$$

is accomplished by a call to

```
void FLA_Cont_with_1x3_to_1x2 ( FLA_Obj *AL, FLA_Obj *AR,
                                FLA_Obj A0, FLA_Obj A1, FLA_Obj A2, int side )
```

**Purpose:** Update the $1 \times 2$ partitioning of matrix $A$ by moving the boundaries so that $A_1$ is added to the side indicated by `side`.

| | |
|---|---|
| `AL, AR` | views of Left and Right sides |
| `side` | side to which $A_1$ is added |
| `A0-A2` | views of $A_0$–$A_2$ |

Parameter `side` indicates whether $A_1$ is added to $A_L$ or $A_R$.

## 3.5 Other useful routines

To examine the contents of an object, we recommend the following routine:

```
void FLA_Obj_show( char *string1, FLA_Obj A, char *format, char *string2 )
```

**Purpose:** Print the contents of $A$.

| | |
|---|---|
| `string1` | string to be printed before contents |
| `A` | descriptor for $A$ |
| `format` | format to be used to print each individual element |
| `string2` | string to be printed after contents |

In particular, the result of

```
FLA_Obj_show( "A =", A, "%lf ", "]" );
```

is something like

```
A = [
< entries >
]
```

which can then be fed to MATLAB. This becomes useful when checking results against a MATLAB implementation of an operation.

# Chapter 4

# Matrix-Matrix Multiplication: The Key to High Performance

by

**John A. Gunnels**

**Greg M. Henry**

**Robert A. van de Geijn**

As will will see in subsequent chapters, most important dense linear algebra operations can be organized so that most of the computation is in matrix-matrix multiplication. Thus, it is important to understand why and how matrix-matrix multiplication can be implemented to achieve high performance on modern microprocessors with hierarchical memories. In this chapter, we describe the basic approach used by our ITXGEMM matrix-matrix multiplication implementation [17].

## 4.1 The object of the game

The basics behind the design of a highly efficient matrix multiplication implementation are rather simple. To implement $C = AB + C$ where $C$, $A$, and $B$ are $m \times n$, $m \times k$, and $k \times n$ matrices, respectively, one starts by partitioning these matrices like

$$C = \left(\begin{array}{c|c|c} C_{11} & \dots & C_{1N} \\ \hline \vdots & & \vdots \\ \hline C_{M1} & \dots & C_{MN} \end{array}\right), A = \left(\begin{array}{c|c|c} A_{11} & \dots & A_{1K} \\ \hline \vdots & & \vdots \\ \hline A_{M1} & \dots & A_{MK} \end{array}\right), \text{ and } B = \left(\begin{array}{c|c|c} B_{11} & \dots & B_{1N} \\ \hline \vdots & & \vdots \\ \hline B_{K1} & \dots & B_{KN} \end{array}\right)$$

where $C_{ij}$ is $m_b \times n_b$, $A_{ip}$ is $m_b \times k_b$, and $B_{pj}$ is $k_b \times n_b$. (Naturally, some blocks may not be exactly this block size, a minor detail.) Now, $C_{ij} = A_{i1}B_{1j} + \cdots + A_{iK}B_{Kj} + C_{ij}$. Given that $C$, $A$, and $B$ all reside in main memory, the blockings of these matrices and the ordering of the updates $C_{ij} = A_{ip}B_{pj} + C_{ij}$ needs to be orchestrated so that movement into the caches of the processor is best amortized over computation.

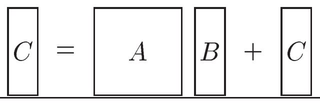
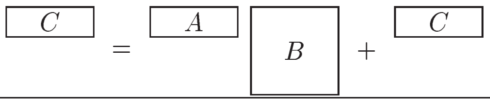
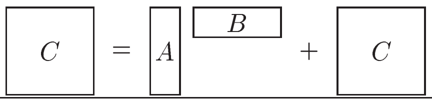
For the moment considering an architecture with two layers of cache memory, this initial partitioning creates blocks to be moved in and out of the L2 cache. Now $C_{ij}$, $A_{ip}$, and $B_{pj}$ are themselves blocked and the computation with these even smaller blocks is orchestrated to optimally utilize the L1 cache. Finally, one further blocking is necessary to optimally utilize the registers. The purpose of the game now is to determine the optimal block size and the optimal ordering of the loops so that data movement between levels of the memory hierarchy is amortized over as much computation as possible.

Notice that for each level of the memory hierarchy we may need as many as three nested loops, not counting the registers. Thus, for a typical architecture with two caches and a main memory, one must consider as many as nine nested loops. Since the ordering of these loops will affect memory access patterns, one must consider up to $9! = 362880$ different loop orderings. By realizing that it is only the three loops for a given level of the hierarchy that need to be ordered, there are $3 \times 3! = 18$ possible loop orderings. For each of these loop orderings, one needs to consider different blockings at each level of the memory hierarchy. In other words, without some reasonable way of pruning the space of possible algorithms, one faces a formidable task.

The theory that we develop in the first part of this paper will allow us to propose a sensible heuristic for pruning the space of possible algorithms. By combining this heuristic with practical considerations we can reduce the number of different algorithms to only eight. Simultaneously, theoretical and practical considerations allow us to severely restrict the range of reasonable block sizes. The net result is a highly efficient implementation of matrix multiplication.

## 4.2 Special cases of matrix-matrix multiplication

The general form of a matrix-matrix multiply is $C \leftarrow \alpha AB + \beta C$ where $C$ is $m \times n$, $A$ is $m \times k$, and $B$ is $k \times n$. We will use the following terminology when referring to a matrix-matrix multiply when two dimensions are large and one is small:

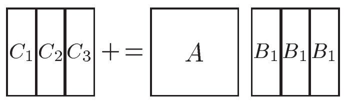
| | Condition | Shape |
|---|---|---|
| Matrix-panel multiply | $n$ is small |  |
| Panel-matrix multiply | $m$ is small |  |
| Panel-panel multiply | $k$ is small |  |

The following observation will become key to understanding concepts encountered in the rest of the paper: Partition

$$X = \left( X_1 \mid \cdots \mid X_{N_X} \right) = \left( \begin{array}{c} \hat{X}_1 \\ \hline \vdots \\ \hline \hat{X}_{M_X} \end{array} \right)$$

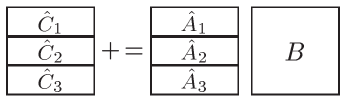
for $X \in \{A, B, C\}$, where $C_j$ is $m \times n_j$, $\hat{C}_i$ is $m_i \times n$, $A_p$ is $m \times k_p$, $\hat{A}_i$ is $m_i \times k$, $B_j$ is $k \times n_j$, and $\hat{B}_p$ is $k_p \times n$. Then $C \leftarrow AB + C$ can be achieved as
multiple matrix-panel multiplies:

$$C_j \leftarrow AB_j + C_j \text{ for } j = 1, \ldots, N_C$$



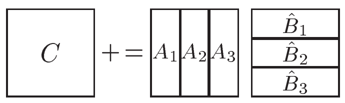
multiple panel-matrix multiplies:

$$\hat{C}_i \leftarrow \hat{A}_i B + \hat{C}_i \text{ for } i = 1, \ldots, M_C$$



or multiple panel-panel multiplies

$$C \leftarrow A_1 \hat{B}_1 + \cdots + A_{N_A} \hat{B}_{N_A}$$



## 4.3 A cost model for hierarchical memories

The memory hierarchy of a modern microprocessor is often viewed as the pyramid given in Fig. 4.1: At the

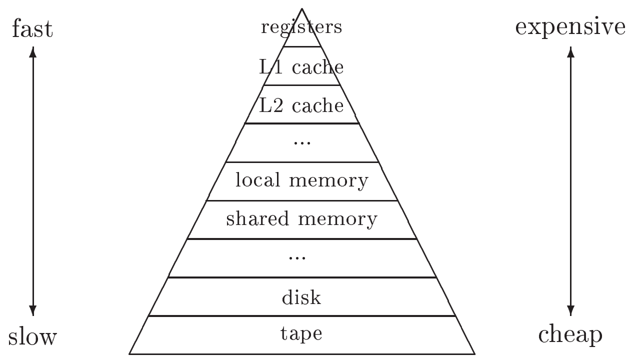


Figure 4.1: The hierarchical memories viewed as a pyramid.

top of the pyramid, there are the processor registers, with extremely fast access. At the bottom, there are disks and even slower media. As one goes down the pyramid, the amount of memory increases as well as the time required to access that that memory, while the cost of memory decreases.

We will model the above mentioned hierarchy naively as follows:

1. The memory hierarchy consists of $L$ levels, indexed $0, \dots, L-1$. Level 0 corresponds to the registers. We will often denote the $i$th level by $L_i$. Notice that on a typical current architecture $L_1$ and $L_2$ correspond the level 1 and level 2 data caches and $L_3$ corresponds to RAM.
2. Level $h$ of the memory hierarchy can store $S_h$ floating point numbers. Generally $S_0 \leq S_1 \leq \cdots \leq S_{L-1}$.
3. Loading a floating point number stored in level $h+1$ to level $h$ costs time $\rho_h$. We will assume that $\rho_0 < \rho_1 < \cdots < \rho_{L-1}$.
4. Storing a floating point number from level $h$ to level $h+1$ costs time $\sigma_h$. We will assume that $\sigma_0 < \sigma_1 < \cdots < \sigma_{L-1}$.
5. If $m_h \times n_h$ matrix $C$, $m_h \times k_h$ matrix $A$, and $k_h \times n_h$ matrix $B$ are all stored in level $h$ of the memory hierarchy then forming $C \leftarrow AB + C$ costs time $2m_h n_h k_h \gamma_h$. (Notice that $\gamma_h$ will depend on $m_h$, $n_h$, and $k_h$).

## 4.4 Building-blocks for matrix multiplication

Consider the matrix multiplication $C \leftarrow AB + C$ where $m_{h+1} \times n_{h+1}$ matrix $C$, $m_{h+1} \times k_{h+1}$ matrix $A$, and $k_{h+1} \times n_{h+1}$ matrix $B$ are all stored in $L_{h+1}$. Let us assume that somehow an efficient matrix multiplication kernel exists for matrices stored in $L_h$. In this section, we develop three distinct approaches for matrix multiplication kernels for matrices stored in $L_{h+1}$.

Partition

$$(4.1)\quad C = \left(\begin{array}{c|c|c} C_{11} & \cdots & C_{1N} \\ \hline \vdots & & \vdots \\ \hline C_{M1} & \cdots & C_{MN} \end{array}\right), A = \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \\ \hline \vdots & & \vdots \\ \hline A_{M1} & \cdots & A_{MK} \end{array}\right), \text{ and } B = \left(\begin{array}{c|c|c} B_{11} & \cdots & B_{1N} \\ \hline \vdots & & \vdots \\ \hline B_{K1} & \cdots & B_{KN} \end{array}\right)$$

where $C_{ij}$ is $m_h \times n_h$, $A_{ip}$ is $m_h \times k_h$, and $B_{pj}$ is $k_h \times n_h$. We must now determine the optimal $m_h$, $n_h$, and $k_h$.

**Algorithm 1**

```
for $j = 1, \ldots, N$
    for $i = 1, \ldots, M$
        Load $C_{ij}$ from $L_{h+1}$ to $L_h$.                    $m_h n_h \rho_h$
        for $p = 1, \ldots, K$
            Load $A_{ip}$ from $L_{h+1}$ to $L_h$.                $m_h k_h \rho_h$
            Load $B_{pj}$ from $L_{h+1}$ to $L_h$.                $k_h n_h \rho_h$
            Update $C_{ij} \leftarrow A_{ip} B_{pj} + C_{ij}$     $2 m_h n_h k_h \gamma_h$
        endfor
        Store $C_{ij}$ from $L_h$ to $L_{h+1}$                    $m_h n_h \sigma_h$
    endfor
endfor
```

Figure 4.2: Multiple panel-panel multiply based blocked matrix-matrix multiplication.

### 4.4.1 Multiple panel-panel multiplies in $L_h$

Noting that $C_{ij} \leftarrow \sum_{p=1}^{K} A_{ip} B_{pj} + C_{ij}$, let us consider the algorithm in Fig. 4.2 for computing the matrix-matrix multiplication. In that figure the costs of the various operations are shown to the right. The order of the outer-most loops is irrelevant to the analysis.

The cost for updating $C$ is given by

$$\sum_{i=1}^{M} \sum_{j=1}^{N} \left[ m_h n_h \rho_h + m_h n_h \sigma_h + \sum_{p=1}^{K} [k_h m_h \rho_h + k_h n_h \rho_h + 2 m_h n_h k_h \gamma_h] \right]$$

$$= m_{h+1} n_{h+1} (\rho_h + \sigma_h) + m_{h+1} n_{h+1} k_{h+1} \frac{\rho_h}{n_h} + m_{h+1} n_{h+1} k_{h+1} \frac{\rho_h}{m_h} + 2 m_{h+1} n_{h+1} k_{h+1} \gamma_h \tag{4.2}$$

Since $\gamma_{h+1}$ is defined to be the cost of a floating point operation when all three matrices are stored in $L_{h+1}$, we find that by we also have that the cost is given by

$$2 m_{h+1} n_{h+1} k_{h+1} \gamma_{h+1} \tag{4.3}$$

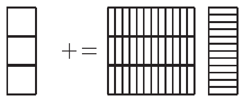
Thus, by dividing 4.2 by $2 m_{h+1} n_{h+1} k_{h+1}$ the effective cost per floating point operation at this level is given by

$$\gamma_{h+1} = \frac{\rho_h + \sigma_h}{2 k_{h+1}} + \frac{\rho_h}{2 n_h} + \frac{\rho_h}{2 m_h} + \gamma_h$$

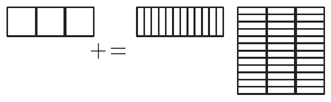
The question now is how to find the $m_h$, $n_h$, and $k_h$ that minimize $\gamma_{h+1}$ under the constraint that $C_{ij}$, $A_{ik}$ and $B_{kj}$ all fit in $L_h$, i.e., $m_h n_h + m_h k_h + n_h k_h \leq S_h$. The smaller $k_h$, the more space in $L_h$ can be dedicated to $C_{ij}$ and thus the smaller the fractions $\rho_h / m_h$ and $\rho_h / n_h$ can be made. A good strategy is thus to let essentially all of $L_h$ be dedicated to $C_{ij}$, i.e., $m_h n_h \approx S_h$. The minimum is then attained when essentially $m_h \approx n_h \approx \sqrt{S_h}$.

Notice that it suffices to have $m_{h+1} = m_h$ or $n_{h+1} = n_h$ for the above cost of $\gamma_{h+1}$ to be achieved. Thus, the above already for the special cases

$$\begin{pmatrix} C_{11} \\ \hline\hline \vdots \\ \hline\hline C_{M1} \end{pmatrix} += \begin{pmatrix} A_{11} & \cdots & A_{1K} \\ \hline\hline \vdots & & \vdots \\ \hline\hline A_{M1} & \cdots & A_{MK} \end{pmatrix} \begin{pmatrix} B_{11} \\ \hline \vdots \\ \hline B_{K1} \end{pmatrix} \tag{4.4}$$



$$\begin{pmatrix} C_{11} \| \cdots \| C_{1N} \end{pmatrix} += \begin{pmatrix} A_{11} | \cdots | A_{1K} \end{pmatrix} \begin{pmatrix} B_{11} & \cdots & B_{1N} \\ \hline \vdots & & \vdots \\ \hline B_{K1} & \cdots & B_{KN} \end{pmatrix} \tag{4.5}$$



Here the distance between single/thin lines is $k_h$ and between double/thick lines $m_h = n_h$, where $k_h$ is much smaller than $m_h$ and $n_h$. The significance of this will become apparent later.

**Algorithm 2**

```
for $p = 1, \ldots, K$
    for $i = 1, \ldots, M$
        Load $A_{ip}$ from level $h+1$ to level $h$.          $m_h k_h \rho_h$
        for $j = 1, \ldots, N$
            Load $C_{ij}$ from level $h+1$ to level $h$.      $m_h n_h \rho_h$
            Load $B_{pj}$ from level $h+1$ to level $h$.      $k_h n_h \rho_h$
            Update $C_{ij} \leftarrow A_{ip} B_{pj} + C_{ij}$    $2 m_h n_h k_h \gamma_h$
            Store $C_{ij}$ from level $h$ to level $h+1$      $m_h n_h \sigma_h$
        endfor
    endfor
endfor
```

Figure 4.3: Multiple matrix-panel multiply based blocked matrix-matrix multiplication.

**Note 1** *The above analysis shows that for the ordering of the loops given in Alg. 1, the strategy should be to load $L_h$ with blocks of $C$ that fill most of $L_h$. The intuitive reason is that the cost of moving blocks $C_{ij}$ between $L_h$ and $L_{h+1}$ is amortized over computation with many smaller blocks $A_{ip}$ and $B_{pj}$, which are "streamed" from $L_{h+1}$. Simultaneously, the cost of bringing each of these smaller blocks into $L_h$ is itself amortized over many computations, since $C_{ij}$ is essentially as large as possible and almost square.*

The inner-most loop in Alg. 1 implements multiple panel-panel multiplies since $k_h$ is small relative to $m_h$ and $n_h$. Thus the name of this section.

## 4.4.2 Multiple matrix-panel multiplies in $L_h$

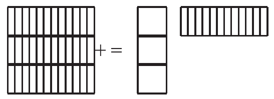
Moving the loops over $l$ and $i$ to the outside we get the algorithm in Fig. 4.3. Performing an analysis similar to that given in Section 4.4.1 the effective cost of a floating point operation is now given by

$$\gamma_{h+1} = \frac{\rho_h}{2n_{h+1}} + \frac{\rho_h + \sigma_h}{2k_h} + \frac{\rho_h}{2m_h} + \gamma_h \tag{4.6}$$

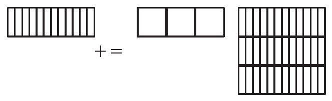
Again, the question is how to find the $m_h$, $n_h$, and $k_h$ that minimize $\gamma_{h+1}$ under the constraint that $C_{ij}$, $A_{ik}$ and $B_{kj}$ all fit in $L_h$, i.e., $m_h n_h + m_h k_h + n_h k_h \leq S_h$. Note that the smaller $n_h$, the more space in $L_h$ can be dedicated to $A_{il}$ and thus the smaller the fractions $(\rho_h + \sigma_h)/2k_h$ and $\rho_h/2m_h$ can be made. A good strategy is thus to let essentially all of $L_h$ be dedicated to $A_{il}$, i.e., $m_h k_h \approx S_h$. The minimum is then attained when essentially $m_h \approx k_h \approx \sqrt{S_h}$.

Notice that it suffices to have $m_{h+1} = m_h$ or $k_{h+1} = k_h$ for the above cost of $\gamma_{h+1}$ to be achieved. In other words, the above holds for the special cases

$$\left(\begin{array}{c|c|c} C_{11} & \cdots & C_{1N} \\ \hline\hline \vdots & & \vdots \\ \hline\hline C_{M1} & \cdots & C_{MN} \end{array}\right) \mathrel{+}= \left(\begin{array}{c} A_{11} \\ \hline\hline \vdots \\ \hline\hline A_{M1} \end{array}\right) \left(\begin{array}{c|c|c} B_{11} & \cdots & B_{1N} \end{array}\right) \tag{4.7}$$



$$\left(\begin{array}{c||c||c} C_{11} & \cdots & C_{1N} \end{array}\right) \mathrel{+}= \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \end{array}\right) \left(\begin{array}{c||c||c} B_{11} & \cdots & B_{1N} \\ \hline \vdots & & \vdots \\ \hline B_{K1} & \cdots & B_{KN} \end{array}\right) \tag{4.8}$$



here the distance between single/thin lines is $n_h$ and between double/thick lines is $m_h = k_h$, where $n_h$ is much smaller than $m_h$ and $k_h$. This will become important later when we notice that these occur naturally as we move up and down the memory hierarchy.

**Note 2** *The above analysis shows that for the ordering of the loops given in Alg. 2, the strategy should be to load $L_h$ with blocks of $A$ that fill most of $L_h$. The intuitive reason is that the cost of moving blocks*

**Algorithm 3**

```
for $j = 1, \ldots, N$
    for $p = 1, \ldots, K$
        Load $B_{pj}$ from level $h+1$ to level $h$.                $k_h n_h \rho_h$
        for $i = 1, \ldots, M$
            Load $C_{ij}$ from level $h+1$ to level $h$.            $m_h n_h \rho_h$
            Load $A_{ip}$ from level $h+1$ to level $h$.            $m_h k_h \rho_h$
            Update $C_{ij} \leftarrow A_{ip} B_{pj} + C_{ij}$        $2 m_h n_h k_h \gamma_h$
            Store $C_{ij}$ from level $h$ to level $h+1$             $m_h n_h \sigma_h$
        endfor
    endfor
endfor
```

Figure 4.4: Multiple panel-matrix multiply based blocked matrix-matrix multiplication.

*$A_{ip}$ between $L_h$ and $L_{h+1}$ is amortized over computation with many smaller blocks $C_{ij}$ and $B_{pj}$, which are "streamed" from $L_{h+1}$. Simultaneously, the cost of bringing each of these smaller blocks into $L_h$ is itself amortized over many computations, since $A_{ip}$ is essentially as large as possible and almost square.*

The inner-most loop in Alg. 2 implements multiple matrix-panel multiplies since $n_h$ is small relative to $m_h$ and $k_h$. Thus the name of this section.

### 4.4.3 Multiple panel-matrix multiplies in $L_h$

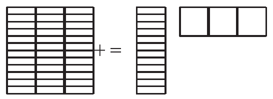
Finally, moving the loops over $p$ and $j$ to the outside we get the algorithm given in Fig. 4.4. This time, the effective cost of a floating point operation is given by

$$\gamma_{h+1} = \frac{\rho_h}{2m_{h+1}} + \frac{\rho_h + \sigma_h}{2k_h} + \frac{\rho_h}{2n_h} + \gamma_h \tag{4.9}$$

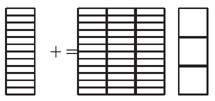
Again, the question is how to find the $m_h$, $n_h$, and $k_h$ that minimize $\gamma_{h+1}$ under the constraint that $C_{ij}$, $A_{ik}$ and $B_{kj}$ all fit in $L_h$, i.e., $m_h n_h + m_h k_h + n_h k_h \leq S_h$. Note that the smaller $m_h$, the more space in $L_h$ can be dedicated to $B_{pj}$ and thus the smaller the fractions $(\rho_h + \sigma_h)/2k_h$ and $\rho_h/2n_h$ can be made. A good strategy is thus to let essentially all of $L_h$ be dedicated to $B_{pj}$, i.e., $n_h k_h \approx S_h$. The minimum is then attained when essentially $n_h \approx k_h \approx \sqrt{S_h}$.

Notice that it suffices to have $n_{h+1} = n_h$ and/or $k_{h+1} = k_h$ for the above cost of $\gamma_{h+1}$ to be achieved. In other words, the above holds for the special cases

$$\left(\begin{array}{c|c|c} C_{11} & \cdots & C_{1N} \\ \hline\hline \vdots & & \vdots \\ \hline\hline C_{M1} & \cdots & C_{MN} \end{array}\right) += \left(\begin{array}{c} A_{11} \\ \hline\hline \vdots \\ \hline\hline A_{M1} \end{array}\right) \left(\begin{array}{c|c|c} B_{11} & \cdots & B_{1N} \end{array}\right) \tag{4.10}$$



$$\left(\begin{array}{c} C_{11} \\ \hline\hline \vdots \\ \hline\hline C_{M1} \end{array}\right) += \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \\ \hline\hline \vdots & & \vdots \\ \hline\hline A_{M1} & \cdots & A_{MK} \end{array}\right) \left(\begin{array}{c} B_{11} \\ \hline\hline \vdots \\ \hline\hline B_{K1} \end{array}\right) \tag{4.11}$$



an observation that will become important later.

**Note 3** *The above analysis shows that for the ordering of the loops given in Alg. 3, the strategy should be to load $L_h$ with blocks of $B$ that fill most of $L_h$. The intuitive reason is that the cost of moving block $B_{pj}$ between $L_h$ and $L_{h+1}$ is amortized over computation with many smaller blocks $C_{ij}$ and $A_{ip}$, which are "streamed" from $L_{h+1}$. Simultaneously, the cost of bringing each of these smaller blocks into $L_h$ is itself amortized over many computations, since $B_{pj}$ is essentially as large as possible and almost square.*

## 4.5 A heuristic for a multi-level algorithm

Key observations so far are

- From Section 4.4: If one were to perform a matrix-matrix multiplication with all operands stored in $L_{L-1}$ (as would naturally occur as part of an application) then this operation should be staged to perform multiple panel-panel, matrix-panel, or panel-matrix multiplies, moving data to and/from $L_{L-2}$.
- Each of the individual panel-panel, matrix-panel, or panel-matrix multiplication has the property that all operands reside in $L_{L-2}$ and should be staged itself be implemented by utilizing $L_{L-3}$ efficiently.
- Whenever all operands of the matrix-matrix multiply fill most of $L_{h+1}$
  - a panel-panel multiply can be efficiently implemented by performing multiple matrix-panel or panel-matrix multiplies in $L_h$. This follows from (4.7) and (4.10).
  - a matrix-panel multiply can be efficiently implemented by performing multiple panel-panel or panel-matrix multiplies in $L_h$. This follows from (4.4) and (4.11).
  - a panel-matrix multiply can be efficiently implemented by performing multiple panel-panel or matrix-panel multiplies in $L_h$. This follows from (4.5) and (4.8).
- From Section 4.2 we conclude that even the matrix-matrix multiply in $L_{L-1}$ can be staged as multiple panel-panel, matrix-panel, or panel-matrix multiplies.

Thus, we conclude that at each layer of the memory hierarchy we should stage the matrix-matrix multiply as multiple panel-panel, matrix-panel, or panel-matrix multiplies.

These observations leads to the following heuristic for implementing the matrix-matrix multiply:

- If in level $L_{h+1}$ one encounters a panel-panel multiply, an optimal implementation will utilize a matrix-panel or panel-matrix multiply in $L_h$. Moreover, the optimal matrix-panel or panel-matrix multiply in $L_h$ will pick $k_h \approx \sqrt{S_h}$ and thus $k_{h+1} = k_h \approx \sqrt{S_h}$. (Recall that we already determined that $m_{h+1} \approx n_{h+1} \approx \sqrt{S_{h+1}}$ was a desirable blocking.)
- If in level $L_{h+1}$ one encounters a matrix-panel multiply, an optimal implementation will utilize a panel-panel or panel-matrix multiply in $L_h$. Moreover, the optimal panel-panel or panel-matrix multiply in $L_h$ will pick $n_h \approx \sqrt{S_h}$ and thus $n_{h+1} = n_h \approx \sqrt{S_h}$.
- If in level $L_{h+1}$ one encounters a panel-matrix multiply, an optimal implementation will utilize a panel-panel or matrix-panel multiply in $L_h$. Moreover, the optimal panel-panel or matrix-panel multiply in $L_h$ will pick $m_h \approx \sqrt{S_h}$ and thus $m_{h+1} = m_h \approx \sqrt{S_h}$.

The decision made at a give level $L_{h+1}$ is summarized in Fig. 4.5. In other words, at each level of the hierarchy, $L_h$, one of the three operands is chosen to be approximately $\sqrt{S_h} \times \sqrt{S_h}$ and fills most of that memory layer while the other two operands are either approximately $\sqrt{S_{h-1}} \times \sqrt{S_h}$ of $\sqrt{S_h} \times \sqrt{S_{h-1}}$. Another way of viewing this is that one of the operands is moved into level $L_h$ while the other two operands are streamed (moved in in smaller submatrices) from level $L_{h+1}$. Notice that if $m_0$, $n_0$, and $k_0$ are now given, all block sizes are approximately determined by the above analysis.

The above heuristic leaves a number of questions:

1. What choices to make in memory layer $L_{L-1}$ since there the shape of the matrices may not cleanly fall into any of these categories. In particular, What if at some level $L_{L-1}$ the smallest dimension is much larger than $\sqrt{S_{L-2}}$? Notice that our theory actually does answer this question since...
2. What if at some level $L_{h+1}$ the "small" dimension is much smaller than $\sqrt{S_h}$. Indeed, what if more than one dimension is "small" relative to $\sqrt{S_h}$.

We will visit these questions later.

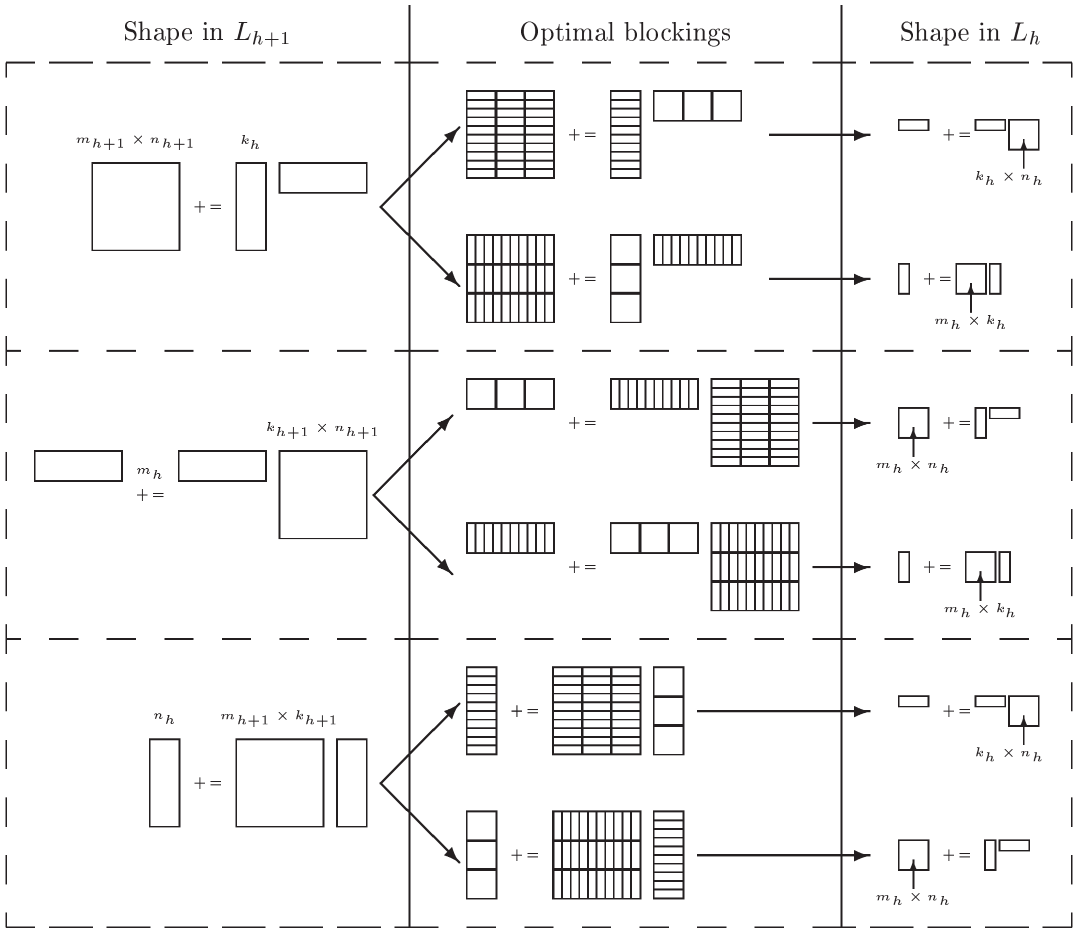


Figure 4.5: Optimal partitioning at memory level $L_h$ and resulting shapes in level $L_{h+1}$.

## 4.6 Practical considerations

In the previous section we developed a heuristic for implementation of matrix-matrix multiplication that best amortizes movement of data between memory hierarchies from a local point of view. However, there are many issues associated with actual implementation that are ignored by the analysis and the heuristic. In this section we discuss implementation details that do take some of those issues into account. We do so by noting certain machine characteristics that to our knowledge hold for a wide variety of architectures.

Rather than making the register level our lowest level, we start with $L_1$, the L1 cache. The reason for this is that at that level loop indexing is a major concern and thus a lot of machine details must be considered. We will not discuss how registers come into play since this goes beyond the scope of this chapter.

### 4.6.1 $L_1$-kernels (lowest level)

While previously we have discussed the shape of the computation to be performed in the $L_1$ cache to be a panel-panel, matrix-panel or panel-matrix multiply, in order to keep loop indexing down to a minimum, at that level our kernels actually perform multiple such operations. Specifically, this allows the loops to be unrolled to eliminate most of the loop overhead.

#### Matrix-panel $L_1$-kernel

Our theory indicates that one of the operations that may be encountered at the $L_1$ level is a matrix-panel multiply. Instead, we consider the operation $C \leftarrow AB + C$ where $C$ is $m_1 \times n$, $A$ is $m_1 \times k_1$, and $B$ is $k_1 \times n$, with $n >> n_0$. The idea is that the overhead of performing the multiple matrix-panel multiplies encountered in the matrix-panel multiply based approaches discussed in Sec. 4.4.2 is amortized over many such matrix-panel multiplies.

The question is how to perform the computation so that elements of $A$ are used with a frequency so that the cache-replacement policy keeps $A$ in the $L_1$ cache. To achieve this, $C$ is computed a few columns at a time. For example, if $C$ is computed a single column at a time, for every $m_1$ elements of $C$ and $k_1$ elements of $B$ all $m_1 \times k_1$ elements of $A$ are accessed and which tends to keep $A$ in the L1 cache.

#### Panel-matrix $L_1$-kernel

We similar treat the case where the shape of matrices in $L_1$ is a panel-matrix multiply. This time, we consider the operation $C \leftarrow AB + C$ where $C$ is $m \times n_1$, $A$ is $m \times k_1$, and $B$ is $k_1 \times n_1$, with $m >> m_0$. The idea is that the overhead of performing the multiple panel-matrix multiplies encountered in the panel-matrix multiply based approaches discussed in Sec. 4.4.3 is amortized over many such panel-matrix multiplies.

This time, the computation must be orchestrated in such a way that the elements of $B$ are used with a frequency so that the cache-replacement policy keeps $B$ in the $L_1$ cache. To achieve this, $C$ is computed a few rows at a time.

#### Panel-panel $L_1$-kernel

We could similarly treat the case where the shape of matrices in $L_1$ is a panel-panel multiply. If so, we would consider the operation $C \leftarrow AB + C$ where $C$ is $m_1 \times n_1$, $A$ is ${}_1m \times k$, and $B$ is $k \times n_1$, with $k >> k_0$. However, for this approach leads to a utilization of the registers that requires elements of $C$ to be loaded to and stored from registers with great frequency. This inherently leads to an $L_1$-kernel that is slower than either the matrix-panel or panel-matrix multiply kernel. Thus, we don't consider this approach to be a viable $L_1$-kernel. We will analyze approaches for $L_2$-kernels under the assumption that this particular $L_1$-kernel is not available. **Thus, in Fig. 4.6 we delete from Fig. 4.5 the panel-panel multiply as a possible shape in the $L_1$ level.**

### 4.6.2 $L_2$-kernel

We now ask ourselves the question of what possible algorithms can be implemented when we perform a multiplication with matrices that are stored in $L_2$. To answer this question, we turn to Fig. 4.6 in which we delete the shapes and algorithms in Fig. 4.5 that now cannot be supported.

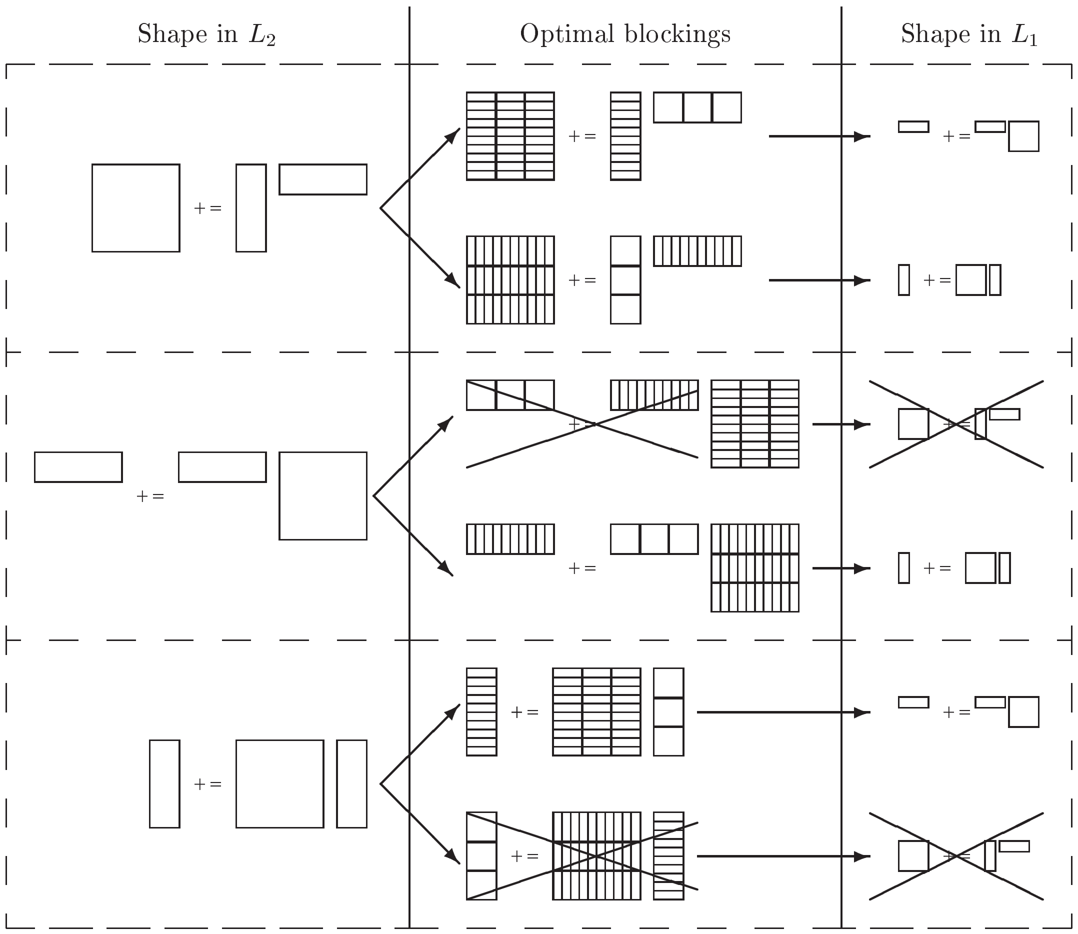


Figure 4.6: Possible algorithms for matrices in memory level $L_2$ given that our kernel at $L_1$ can only accommodate matrix-panel and panel-matrix multiplication.

Just like for the $L_1$-kernel, our $L_2$-kernel assumes that one matrix of size approximately $\sqrt{S_2} \times \sqrt{S_2}$ is moved into $L_2$ after which multiple panel-panel, matrix-panel, or panel-matrix multiplies commence. Depending on which matrix occupies most of $L_2$, one of the four blockings remaining in Fig. 4.6 is used to smaller subproblems that can be passed to the $L_1$-kernel.

For the sake of consistency, our $L_2$-kernel performs multiple panel-panel, matrix-panel, or panel-matrix multiplies, like the $L_1$-kernel. This is illustrated in Fig. 4.7.

Notice that if the $L_2$-kernel implements multiple panel-panel multiplies, there is a choice of two possible blockings. One leads to a panel-matrix multiply as basic operation in the $L_1$ level, the other to a matrix-panel multiply. The question naturally becomes which of the two to use. Notice from (4.9) that the panel-matrix multiply based algorithm has a cost of

$$\gamma_2^{\mathrm{PM}} = \frac{\rho_1}{2m_2} + \frac{\rho_1 + \sigma_1}{2k_1} + \frac{\rho_1}{2n_1} + \gamma_1$$

while (4.6) shows the matrix-panel multiply based algorithm has a cost of

$$\gamma_2^{\mathrm{MP}} = \frac{\rho_1}{2n_2} + \frac{\rho_1 + \sigma_1}{2k_1} + \frac{\rho_1}{2m_1} + \gamma_1$$

If parameters $\rho_1$ and $\sigma_1$ are equal in both these equations, the panel-matrix multiply based algorithm outperforms the matrix-panel multiply based algorithm when $\gamma_2^{\mathrm{PM}} < \gamma_2^{\mathrm{MP}}$ or

$$\frac{1}{m_2} + \frac{1}{n_1} < \frac{1}{n_2} + \frac{1}{m_1}$$

Note that one can expect $\gamma_1$ to be equal for both equations since in our situation the $L_1$-kernel is one and the same for both approaches.

### 4.6.3 $L_3$-kernel

For current generation microprocessors, the $L_3$ level is typically the primary RAM of the processor. For this reason, our discussion will target that situation.

Notice that while in this level on the surface it may appear that one should analyze the general matrix-matrix multiplication $C \leftarrow \alpha AB + \beta C$ for general $m \times n$ matrix $C$, $m \times k$ matrix $A$, and $k \times n$ matrix $B$. However, a common use of matrix-matrix multiplication is as part of the implementation of other dense linear algebra algorithms, e.g. for factorization operations like LU, Cholesky, and QR factorization. In those algorithms, as implemented in LAPACK, the matrix-matrix multiply invariably appears as a panel-panel, matrix-panel or panel-matrix multiply. Indeed, the width of the panels involved are determined by the width that makes matrix-matrix multiplication operate at peak performance. Thus, the most important cases of to analyze are exactly those where one of $m$, $n$, or $k$ equals approximately $\sqrt{S_2}$. Thus, we again analyze the panel-panel, matrix-panel, and panel-matrix multiply before proceeding with the general case. To do so, again consider Fig. 4.5.

#### $L_3$-kernel for panel-panel multiply

In this case $k = k_2 \approx \sqrt{S_2}$. As for the $L_2$-kernel there are now two choices for implementation: a panel-matrix multiply based algorithm and a matrix-panel multiply based algorithm. The first yields an effective cost per floating point operation of

$$\gamma_3^{\mathrm{PM}} = \frac{\rho_2}{2m} + \frac{\rho_2 + \sigma_2}{2k_2} + \frac{\rho_2}{2n_2} + \gamma_2$$

while the second yields

$$\gamma_3^{\mathrm{MP}} = \frac{\rho_2}{2n} + \frac{\rho_2 + \sigma_2}{2k_2} + \frac{\rho_2}{2m_2} + \gamma_2$$

If parameters $\rho_2$, $\sigma_2$, and $\gamma_2$ are equal in both these equations, the panel-matrix multiply based algorithm outperforms the matrix-panel multiply based algorithm when $\gamma_3^{\mathrm{PM}} < \gamma_3^{\mathrm{MP}}$ or

$$\frac{1}{m} + \frac{1}{n_2} < \frac{1}{n} + \frac{1}{m_2}$$

Note that one cannot expect $\gamma_2$ to be equal for both equations since in our situation the $L_2$-kernel for each of the $L_2$-kernels is not the same for both approaches.

**$L_3$-kernel for panel-matrix multiply**

In this case $m = m_2 \approx \sqrt{S_2}$. Unlike for the $L_2$-kernel there are now two choices for implementation: a panel-panel or matrix-panel multiply based algorithm. The first yields an effective cost per floating point operation of

$$\gamma_3^{\text{PP}} = \frac{\rho_2}{2m_2} + \frac{\rho_2 + \sigma_2}{2k} + \frac{\rho_2}{2n_2} + \gamma_2$$

while the second yields

$$\gamma_3^{\text{MP}} = \frac{\rho_2}{2n} + \frac{\rho_2 + \sigma_2}{2k_2} + \frac{\rho_2}{2m_2} + \gamma_2$$

Again, if parameters $\rho_2$, $\sigma_2$, and $\gamma_2$ are equal in both these equations, the panel-panel multiply based algorithm outperforms the matrix-panel multiply based algorithm when $\gamma_3^{\text{PP}} < \gamma_3^{\text{MP}}$ or

$$??? < \frac{1}{n} + \frac{1}{m_2}$$

Again, one cannot expect $\gamma_2$ to be equal for both equations since in the $L_2$-kernel for the two approaches is not the same.

**$L_3$-kernel for matrix-panel multiply**

In this case $n = n_2 \approx \sqrt{S_2}$. There are two choices for implementation: a panel-panel and a panel-matrix multiply based algorithm. The first yields an effective cost per floating point operation of

$$\gamma_3^{\text{PP}} = \frac{\rho_2}{2m_2} + \frac{\rho_2 + \sigma_2}{2k} + \frac{\rho_2}{2n_2} + \gamma_2$$

while the second yields

$$\gamma_3^{\text{PM}} = \frac{\rho_2}{2m} + \frac{\rho_2 + \sigma_2}{2k_2} + \frac{\rho_2}{2n_2} + \gamma_2$$

If parameters $\rho_2$, $\sigma_2$, and $\gamma_2$ are equal in both these equations, the panel-panel multiply based algorithm outperforms the panel-matrix multiply based algorithm when $\gamma_3^{\text{PP}} < \gamma_3^{\text{PM}}$ or

$$\frac{1}{m} + \frac{1}{n_2} < ???$$

Again, one cannot expect $\gamma_2$ to be equal for both equations since in the $L_2$-kernel for the two approaches is not the same.

## 4.7 A family of algorithms

We now turn the observations made above into a practical implementation.

High-performance implementations of matrix multiplication typically start with an "inner-kernel". This kernel carefully orchestrates the movement of data in and out of the registers and the computation under the assumption that one or more of the operands are in the L1 cache. For our implementation on the Intel Pentium (R) III processor, the inner-kernel performs the operation $C = A^T B + \beta C$ where $64 \times 8$ matrix $A$ is kept in the L1 cache. Matrices $B$ and $C$ have a large number of columns, which we view as multiple-panels, with each panel of width one. Thus, our inner-kernel performs a multiple matrix-panel multiply (MMP) with a transposed resident matrix $A$. The technical reasons why this particular shape was selected go beyond the scope of this paper.

While it may appear that we thus only have one of the three kernels for operation in the L1 cache, notice that for the submatrices with which we compute at that level one can instead compute $C^T = B^T A + C^T$,

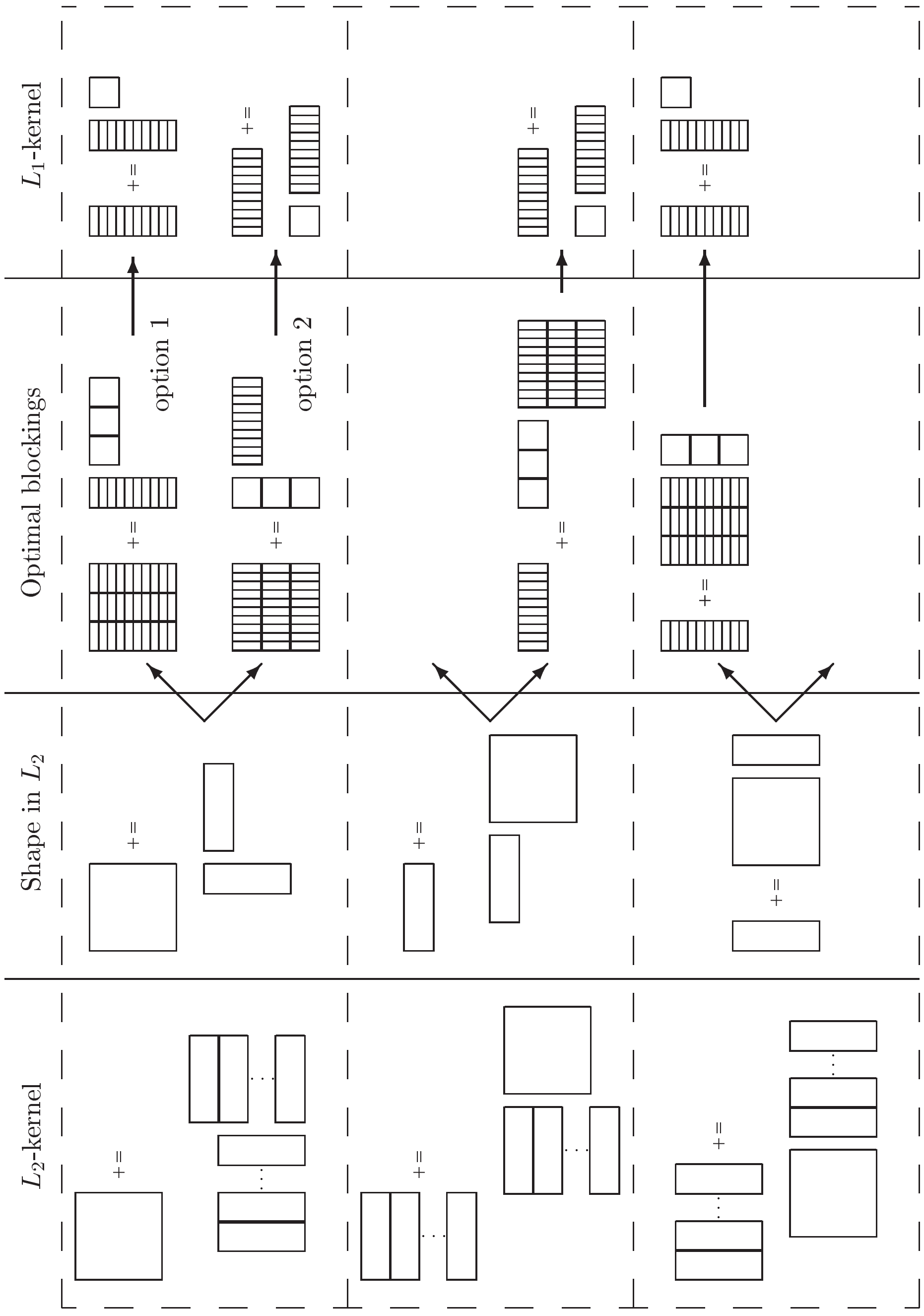


Figure 4.7: A tree of possible algorithms for matrices in memory level $L_3$ given that our kernel at $L_1$ can only accommodate matrix-panel and panel-matrix multiplication.

reversing the role of $A$ and $B$. This simple observation allows us to claim that we also have an inner-kernel that performs a multiple panel-matrix multiply (MPM).

Let us introduce a naming convention for a family of algorithms that perform the discussed algorithms at different levels of the memory hierarchy:

<kernel at $L_3$>-<kernel at $L_2$>-<kernel at $L_1$>.

For example MPP-MPM-MMP will indicate that the $L_3$-kernel uses multiple panel-panel multiplies, calls the $L_2$-kernel that uses multiple matrix-panel multiplies, which in turn calls the $L_1$-kernel that uses multiple panel-matrix multiplies. Given the constraint that only two of the possible three kernel algorithms are implemented at $L_1$, the tree of algorithms in Fig. 4.7 can be constructed.

## 4.8 Performance

In this section, we report performance attained by the different algorithms. Details regarding the performance test bed can be found in 1.4. For the usual matrix dimensions $m$, $n$, and $k$, we use the operation count $2mnk$ for a matrix-matrix multiplication. We tested performance of the operation $C = C - AB$ ($\alpha = -1$ and $\beta = 1$) since this is the case most frequently encountered when matrix multiplication is used in libraries like LAPACK.

### 4.8.1 Implementations tested

It turns out that whenever all operands start by being stored in main memory, there is no noticeable difference between the different loop orderings at that level. In other words, MPM-MMP-MPM achieves performance that is essentially identical to MPP-MMP-MPM. Thus, we only report performance for the following variants: MPM-MMP-MPM, MMP-MPM-MMP, MPM-MPP-MPM, and MPM-MPP-MMP.

### 4.8.2 Determining optimal block sizes

Our first experiment is intended to demonstrate that the block size chosen for the matrix that remains resident in the L2 cache has a clear effect on the overall performance of the matrix multiplication routine. In Fig. 4.8(a) we report performance attained as a function of the fraction of the L2 cache filled with the resident matrix when a matrix multiplication with $k = m = n = 1000$ is executed. This experiments tests our theory that reuse of data in the L2 cache impacts overall performance as well as our theory that the resident matrix should occupy "most" of the L2 cache. Note that performance improves as a larger fraction of the L2 cache is filled with the resident matrix. Once the resident matrix fills more than half of the L2 cache, performance starts to diminish. This is consistent with the theory which tells us that some of the cache must be used for the matrices that are being streamed from main memory. Once more than 3/4 of the L2 cache is filled with the resident matrix, performs drops significantly. This is consistent with the scenario where parts of the other matrices start vacating parts of the resident matrix from the L2 cache.

The exact reason why the MPM-MMP-MPM variant performs better when the block size is chosen appropriately is not entirely clear. Most likely, it has to do with the details of the packing and unpacking routines that are part of the implementation.

Based on the above experiment, we fix the block size for the resident matrix in the L2 cache to $128 \times 128$, which fills exactly half of this cache, for the remaining experiments.

### 4.8.3 Resident matrices

The next set of experiments show that the cost of moving a submatrix into the L2 cache and then amortizing the cost of this memory operation over as much computation as possible is indeed observable in practice.

**Matrix $A$ resident in L2:** In Fig. 4.8(b), dimensions $m$ and $k$ are fixed to 128. This implies that matrix $A$ fills half of the L2 cache. Notice that variant MPM-MMP-MPM will keep $128 \times 128$ submatrices of $A$ resident in the L2 cache. Thus, one would expect performance to increase smoothly as $n$ is increased. For the other variants, one would expect a drop in performance whenever $n$ becomes slightly larger than a multiple of 128, since they attempt to keep $128 \times 128$ submatrices of $C$ or $B$ in the L2 cache: whenever $n$ is slightly larger than 128, one of the submatrices of $A$ or $B$ is relatively small.

**Matrix $B$ resident in L2:** In Fig. 4.8(c), dimensions $n$ and $k$ are fixed to 128. This implies that matrix $B$ fills half of the L2 cache. Notice that variant MMP-MPM-MMP will keep $128 \times 128$ submatrices of $B$ resident in the L2 cache. Thus, one would expect performance to increase smoothly as $m$ is increased. For the other variants, one would expect a drop in performance whenever $k$ becomes slightly larger than a multiple of 128, since they attempt to keep $128 \times 128$ submatrices of $C$ or $A$ in the L2 cache: whenever $m$ is slightly larger than 128, one of the submatrices of $C$ or $A$ is relatively small.

**Matrix $C$ resident in L2:** In Fig. 4.8(d), dimensions $m$ and $n$ are fixed to 128. This implies that matrix $C$ fills half of the L2 cache. Notice that variants MPM-MPP-MPM and MPM-MPP-MMP will both keep $128 \times 128$ submatrices of $C$ resident in the L2 cache. Thus, one would expect performance to increase smoothly as $k$ is increased. For the other variants, one would expect a drop in performance whenever $k$ becomes slightly larger than a multiple of 128, since they attempt to keep $128 \times 128$ submatrices of $A$ or $B$ in the L2 cache: whenever $k$ is slightly larger than 128, one of the submatrices of $A$ or $B$ is relatively small. Unfortunately, we cannot observe this phenomena, since the L1 kernel takes a performance hit every time $k$ is slightly larger than 64.

### 4.8.4 Commonly encountered shapes

The most commonly encountered special cases of matrix-matrix multiplication are the matrix-panel, panel-matrix, and panel-panel multiplications. Not only did they show up in this paper as the shape that is encountered at each level of the memory hierarchy, but it is also the shape that shows up when implementing other matrix operations like LU, Cholesky, and QR factorization, for example as part of LAPACK.

**Matrix-panel multiply** In Fig. 4.9(a) we report performance as a function of $n$ (the number of columns in the panel) when $m$ and $k$ are fixed to be large. Notice that our theory indicates that when a matrix-panel multiply is performed in main memory, the L2 kernel should perform multiple panel-matrix or panel-panel multiplies. The theory indicates that MPP-MMP-MPM and MPM-MMP-MPM, which perform a matrix-panel multiply in the L2 level, should not be good choices. The data in Fig. 4.9(a) supports this.

**Panel-matrix multiply** In Fig. 4.9(b) we report performance as a function of $m$ (the number of columns in the panel) when $n$ and $k$ are fixed to be large. Notice that our theory indicates that when a panel-matrix multiply is performed in main memory, the L2 kernel should perform multiple matrix-panel or panel-panel multiplies. Thus, MPM-MMP-MPM, MPM-MPP-MPM, or MPM-MPP-MMP should perform well. The theory indicates that MMP-MPM-MMP, which perform a panel-matrix multiply in the L2 level, would not be good choice. The data in Fig. 4.9(b) supports this in the sense that MMP-MPM-MMP, which for other matrix shapes frequently did well, does not perform quite as well for small $m$.

**Panel-panel multiply** In Fig. 4.9(c) we report performance as a function of $k$ (the number of columns in the panel) when $m$ and $n$ are fixed to be large. Notice that our theory indicates that when a panel-panel multiply is performed in main memory, the L2 kernel should perform multiple matrix-panel or panel-matrix multiplies. The theory indicates that variants *-MPP-*, which perform multiple panel-panel multiplies in the L2 level, should not be good choices. The data in Fig. 4.9(c) supports this in the range $64 \leq k \leq 128$. Interestingly enough, it does not appear to be true in the range $k \leq 64$ for MPM-MPP-MMP. We believe that can be attributed to the fact that in that range there isn't sufficient opportunity for reuse of data in the L2 cache. As a result, it is best to ignore it, which is essentially what MPM-MPP-MMP does.

### 4.8.5 Square matrices

Matrix multiplication with square matrices is relatively uncommon in practice. However, it is commonly presented in papers. Thus, for good measure, we include Fig. 4.9(a).