# LAFF-On
# Programming for High Performance

Robert A. van de Geijn

Devangi N. Parikh

Jianyu Huang

Margaret E. Myers

Release Date
Sunday 14th October, 2018

This is a work in progress

10 9 8 7 6 5 4 3 2 1

**Library of Congress Cataloging-in-Publication Data** not yet available
Draft Edition, Sunday 14th October, 2018
This "Draft Edition" allows this material to be used while we sort out through what mechanism we will publish the book.

# Contents

# Preface

# Acknowledgments

The cover was derived from an image that is copyrighted by the Texas Advanced Computing Center (TACC). It is used with permission.

Week 1

# Matrix-Matrix Multiplication Basics

## 1.1 Opening Remarks

### 1.1.1 Launch

**Homework 1.1.1.1** Compute

$$\begin{pmatrix} 1 & -2 & 2 \\ -1 & 1 & 3 \\ -2 & 2 & -1 \end{pmatrix} \begin{pmatrix} -2 & 1 \\ 1 & 3 \\ -1 & 2 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ -1 & 2 \\ -2 & 1 \end{pmatrix} =$$

☛ SEE ANSWER

Let $A$, $B$, and $C$ be $m \times k$, $k \times n$, and $m \times n$ matrices, respectively. We can expose their individual entries as

$$A = \begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,k-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,k-1} \\ \vdots & \vdots & & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,k-1} \end{pmatrix}, B = \begin{pmatrix} \beta_{0,0} & \beta_{0,1} & \cdots & \beta_{0,n-1} \\ \beta_{1,0} & \beta_{1,1} & \cdots & \beta_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \beta_{k-1,0} & \beta_{k-1,1} & \cdots & \beta_{k-1,n-1} \end{pmatrix},$$

and

$$C = \begin{pmatrix} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,n-1} \\ \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,n-1} \end{pmatrix}.$$

The computation $C := AB + C$, which adds the result of matrix-matrix multiplication $AB$ to a matrix

```
#include <stdio.h>
#include <stdlib.h>

#define alpha( i,j ) A[ (j)*ldA + i ]   // map alpha( i,j ) to array A
#define beta( i,j )  B[ (j)*ldB + i ]   // map beta( i,j )  to array B
#define gamma( i,j ) C[ (j)*ldC + i ]   // map gamma( i,j ) to array C

void MyGemm( int m, int n, int k, double *A, int ldA,
       double *B, int ldB, double *C, int ldC )
{
  for ( int i=0; i<m; i++ )
    for ( int j=0; j<n; j++ )
      for ( int p=0; p<k; p++ )
        gamma( i,j ) += alpha( i,p ) * beta( p,j );
}
```

Figure 1.1: C implementation of IJP ordering for computing MMM.

$C$, is defined as

$$\gamma_{i,j} = \sum_{p=0}^{k-1} \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$$

for all $0 \leq i < m$ and $0 \leq j < n$. This leads to the following pseudo-code for computing $C := AB + C$:

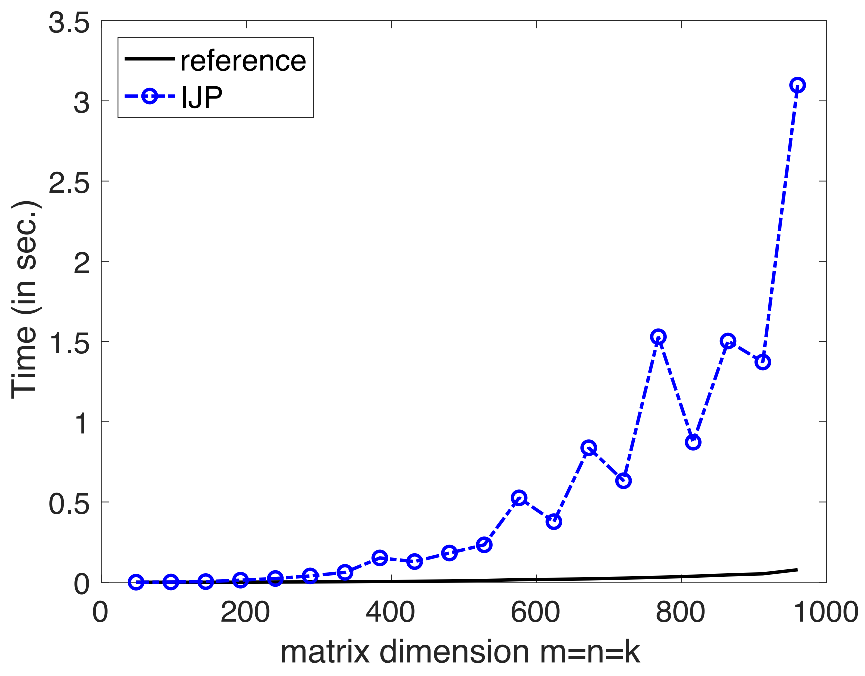
```
for $i := 0, \ldots, m-1$
  for $j := 0, \ldots, n-1$
    for $p := 0, \ldots, k-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end
  end
end
```

The outer two loops visit each element of $C$, and the inner loop updates $\gamma_{i,j}$ with the dot product of the $i$th row of $A$ with the $j$th column of $B$.

**Homework 1.1.1.2** In the file ☛ Assignments/Week1/C/Gemm_IJP.c you will find a simple implementation given in Figure 1.1 that computes $C := AB + C$ (GEMM). Compile, link, and execute it by following the instructions in the MATLAB Live Script ☛ Plot_IJP.mlx in the same directory. (Alternatively, read and execute ☛ Plot_IJP_m.m.)

☛ SEE ANSWER

On Robert's laptop, Homework 1.1.1.2 yields the two graphs given in Figure 1.2 (top) as the curve labeled with IJP. In the first, the time required to compute GEMM as a function of the matrix size is plotted, where $m = n = k$ (each matrix is square). The "dips" in the time required to



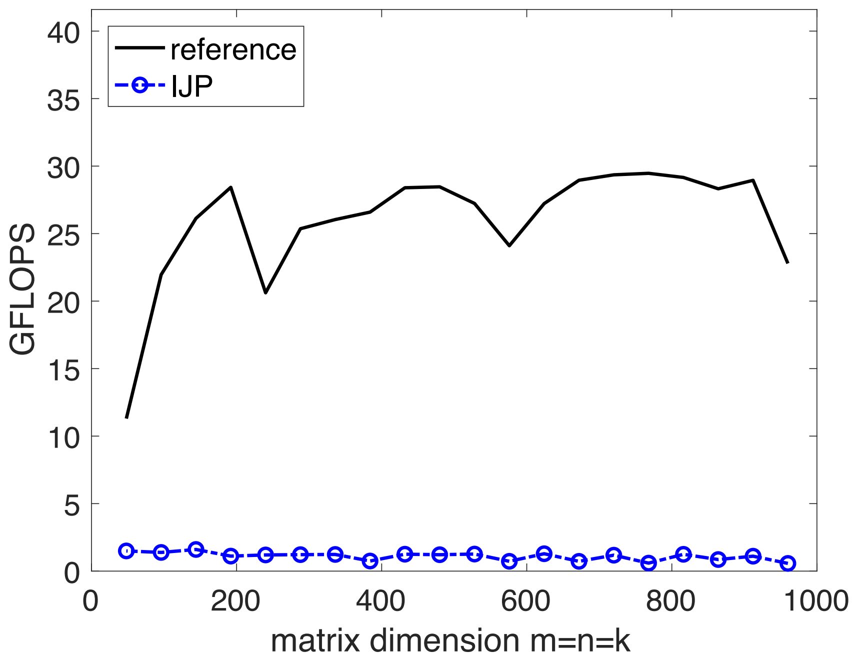


Figure 1.2: Performance comparison of a simple triple-nested loop and a high-performance implementation of the matrix-matrix multiplication $C := AB + C$. Top: Execution time. Bottom: Rate of computation, in billions of floating point operations per second (GFLOPS).

complete can be attributed to a number of factors, including that other processes that are executing on the same processor may be disrupting the computation. One should not be too concerned about those. In the graph, we also show the time it takes to complete the same computations with a highly optimized implementation. To the uninitiated in high-performance computing (HPC), the difference may be a bit of a shock. It makes one realize how inefficient many of the programs we write are.

The performance of a MMM implementation is measured in billions of floating point operations (flops) per second (GFLOPS). The idea is that we know that it takes $2mnk$ flops to compute $C := AB + C$ where $C$ is $m \times n$, $A$ is $m \times k$, and $B$ is $k \times n$. If we measure the time it takes to complete the computation, $T(m,n,k)$, then the rate at which we compute is given by

$$\frac{2mnk}{T(m,n,k)} \times 10^{-9} \text{ GFLOPS}.$$

For our implementation and the reference implementation, this is reported in Figure 1.2 (Bottom). Again, don't worry too much about the dips in the curves. If we controlled the environment in which we performed the experiments, these would largely disappear.

Of course, we don't at this point even know if the reference implementation attains good performance. To investigate this, it pays to compute the theoretical peak performance of the processor. In order to do this,

- You will want to find out what exact processor is in your computer. On a Mac, you can type

  ```
  sysctl -n machdep.cpu.brand_string
  ```

  in a terminal session. On Robert's laptop, this yields

  Intel(R) Core(TM) i5-4250U CPU @ 1.30GHz.

  On a Linux system, you can instead type

  lscpu

  at the command line, which will give you much more detailed information.

- If you then do an internet search, you may find a page like

  ☛ Intel® Core™ i5-4250U Processor

  which tells Robert's laptop is a Haswell processor (look for the "Code name").

- Now this is where it gets interesting: On this page it tells you the Processor Base Frequency is 1.3 GHz, but that the Max Turbo Frequency is 2.60 GHz. The turbo frequency is the frequency at which the processor can operate under special circumstance. The more cores are active, the lower the frequency at which the processor operates to reduce the temperature and power use of the processor.

- You can find out more about what clock rate can be expected by consulting a document like

☛ Intel Xeon Processor E5 v3 Product Family.

This document may make your head spin...

You will want to investigate the above for the processor of the machine on which you perform your experiments.

For the sake of discussion, let's use the clock rate of 2.6 GHz in turbo for Robert's laptop.

- The next thing you need to know is how many flops each core can perform per clock cycle. We happen to know that a Haswell core can perform sixteen flops per cycle.
- Thus, the peak performance of a single core is given by $2.6 \times 16 = 41.6$ GFLOPS when performing double precision real computations (which is what we will focus on in this course).

What we conclude is that the best we can expect, under idealized circumstances, is that our implementation will achieve 41.6 GFLOPS. Looking at Figure 1.2 (bottom), we see that the reference implementation achieves a reasonable rate of computation, while the simple triple loop is dismal in its performance. In that graph, and many that will follow, we make the top of the graph represent the theoretical peak.

### 1.1.2 Outline Week 1

### 1.1.3 What you will learn

In this week, we not only review matrix-matrix multiplication, but also get you to think about this operation in different ways.

Upon completion of this week, you should be able to

- Map matrices to memory;
- Apply conventions to describe how to index into arrays that store matrices;
- Observe the effects of loop order on performance;
- Recognize that simple implementations may not provide the performance that can be achieved;
- Realize that compilers don't automagically do all the optimization for you.

## 1.2 Mapping Matrices to Memory

In this section, learners find out

- How to map matrices to memory.
- Conventions we will use to describe how we index into the arrays that store matrices.
- That loop order affects performance.

The learner is left wondering "why?"

### 1.2.1 Column-major ordering

Matrices are stored in two-dimensional arrays while computer memory is inherently one-dimensional in the way it is addressed. So, we need to agree on how we are going to store matrices in memory.

Consider the matrix

$$\begin{pmatrix} 1 & -2 & 2 \\ -1 & 1 & 3 \\ -2 & 2 & -1 \end{pmatrix}$$

from the opener of this week. In memory, this may be stored in an array `A` by columns:

$$\begin{array}{llr|r|}
\cline{4-4}
\texttt{A} & & \longrightarrow & 1 \\
\texttt{A[1]} & & \longrightarrow & -1 \\
\texttt{A[2]} & & \longrightarrow & -2 \\
\cline{4-4}
\texttt{A[3]} & & \longrightarrow & -2 \\
\vdots & & & 1 \\
 & & & 2 \\
\cline{4-4}
 & & & 2 \\
 & & & 3 \\
 & & & -1 \\
\cline{4-4}
\end{array}$$

which is known as *column-major ordering*.

More generally, consider the matrix

$$\begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,k-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,k-1} \\ \vdots & \vdots & & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,k-1} \end{pmatrix}.$$

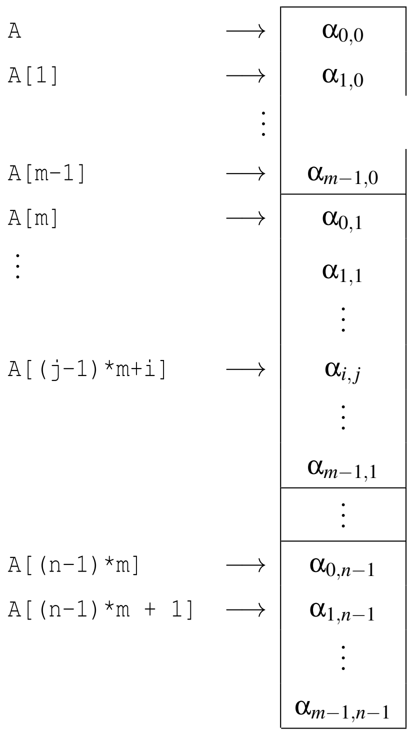


Figure 1.3: Mapping of $m \times n$ matrix $A$ to memory with column-major order.

Column-major ordering would store this in array `A` as illustrated in Figure 1.3. Obviously, one could use the alternative known as *row-major ordering*.

**Homework 1.2.1.1** Let the following picture represent data stored in memory starting at address `A`:

$$\texttt{A} \longrightarrow \begin{array}{|r|} 1 \\ -1 \\ -2 \\ -2 \\ 1 \\ 2 \\ 2 \end{array}$$

and let $A$ be the $2 \times 3$ matrix stored there in column-major order. Then

$$A = \begin{pmatrix} \square & \square & \square \\ \square & \square & \square \end{pmatrix}$$

☛ SEE ANSWER

**Homework 1.2.1.2** Let the following picture represent data stored in memory starting at address `A`.

$$\texttt{A} \longrightarrow \begin{array}{|r|} 1 \\ -1 \\ -2 \\ -2 \\ 1 \\ 2 \\ 2 \end{array}$$

and let $A$ be the $2 \times 3$ matrix stored there in row-major order. Then

$$A = \begin{pmatrix} \square & \square & \square \\ \square & \square & \square \end{pmatrix}$$

☛ SEE ANSWER

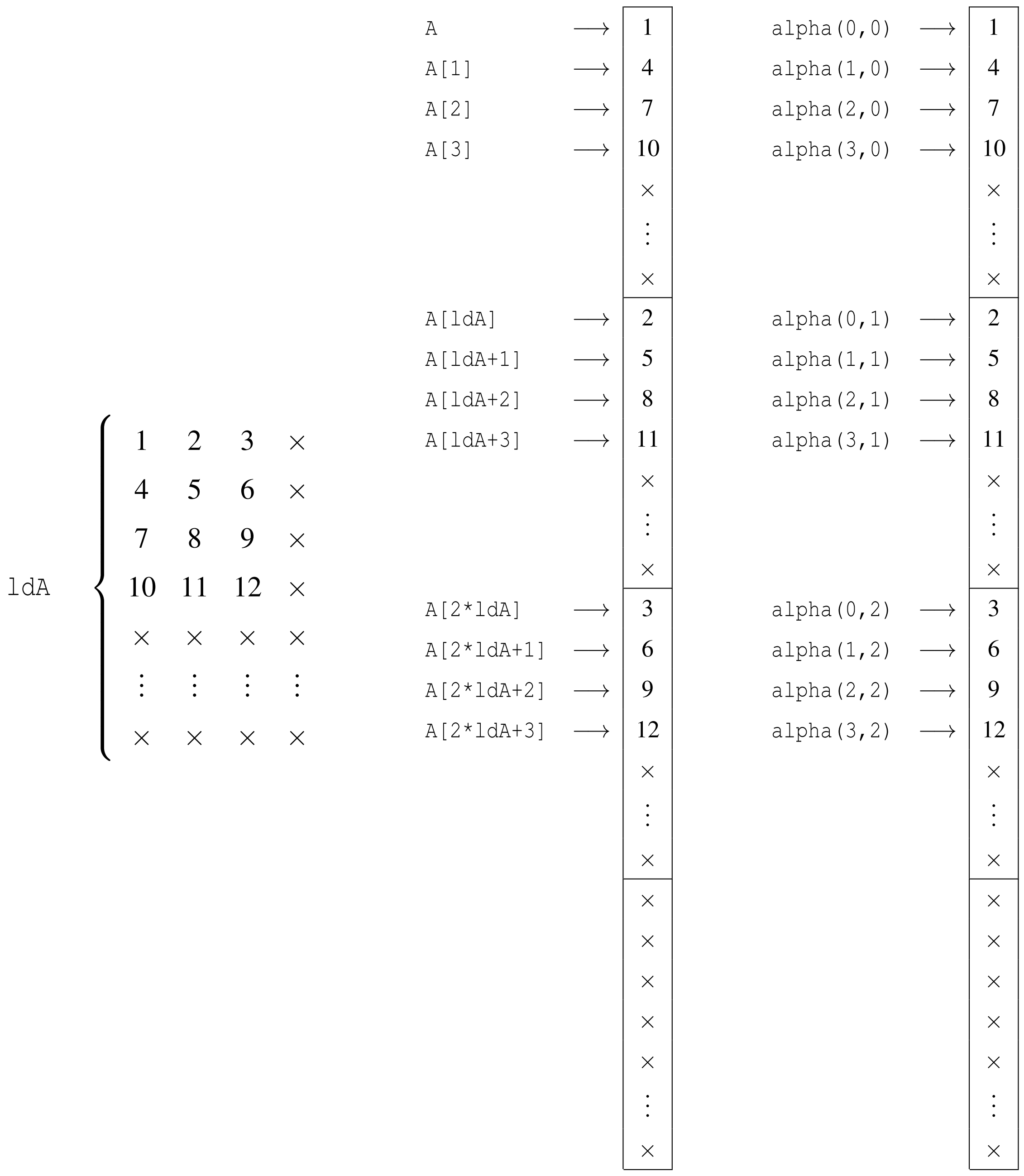


Figure 1.4: Addressing a matrix embedded in an array with ldA rows. At the left we illustrate a $4 \times 3$ submatrix of a ldA $\times$ 4 matrix. In the middle, we illustrate how this is mapped into an linear array a. In the right, we show how defining the C macro `#define alpha(i,j) A[ (j)*ldA + (i)]` allows us to address the matrix in a more natural way.

### 1.2.2 The leading dimension

Very frequently, we will work with a matrix that is a submatrix of a larger matrix. Consider Figure 1.4. What we depict there is a matrix that is embedded in a larger matrix. The larger matrix consists of ldA (the *leading dimension*) rows and some number of columns. If column-major order is used to store the larger matrix, then addressing the elements in the submatrix requires knowledge of the leading dimension of the larger matrix. In the C programming language, if the top-left element of the submatrix is stored at address `A`, then one can address the $(i, j)$ element as `A[ j*ldA + i ]`. We typically define a macro that makes addressing the elements more natural:

```
#define alpha(i,j) A[ (j)*ldA + (i) ]
```

where we assume that the variable or constant `ldA` holds the leading dimension parameter.

**Homework 1.2.2.1** Consider the matrix

$$\begin{pmatrix} 0.0 & 0.1 & 0.2 & 0.3 \\ 1.0 & \boxed{1.1} & \boxed{1.2} & \boxed{1.3} \\ 2.0 & \boxed{2.1} & \boxed{2.2} & \boxed{2.3} \\ 3.0 & 3.1 & 3.2 & 3.3 \\ 4.0 & 4.1 & 4.2 & 4.3 \end{pmatrix}$$

If this matrix is stored in column-major order in a linear array `A`,

1. The boxed submatrix starts at `A[ __ ]`.
2. The row size of the boxed submatrix is ___.
3. The column size of the boxed submatrix is ___.
4. The leading dimension of the boxed submatrix is ___.

☛ SEE ANSWER

### 1.2.3 A convention regarding the letter used for the loop index

When we talk about loops for matrix-matrix multiplication (MMM), it helps to keep the picture in Figure 1.5 in mind, which illustrates which loop index (variable name) is used for what row or column of the matrices. We try to be consistent in this use, as should you.

### 1.2.4 Ordering the loops

Consider again a simple algorithm for computing $C := AB + C$.

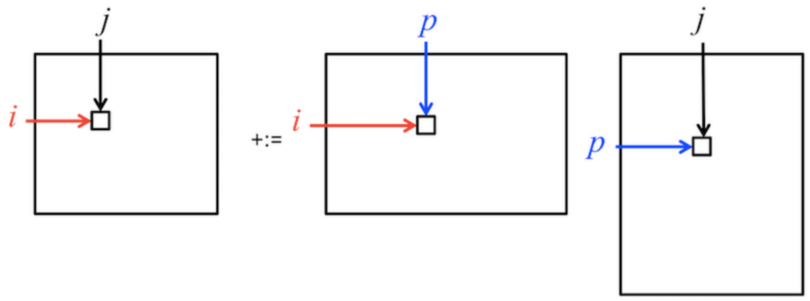


Figure 1.5: We embrace the convention that in loops that implement MMM, the variable $i$ indexes the current row of matrix $C$ (and hence the current row of $A$), the variable $j$ indexes the current column of matrix $C$ (and hence the current column of $B$), and variable $p$ indexes the "other" dimension, namely the current column of $A$ (and hence the current row of $B$). In other literature, the index $k$ is often used instead of the index $p$. However, it is customary to talk about $m \times n$ matrix $C$, $m \times k$ matrix $A$, and $k \times n$ matrix $B$. Thus, the letter $k$ is "overloaded" and we use $p$ instead.

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

Given that we now embrace the convention that $i$ indexes rows of $C$ and $A$, $j$ indexes columns of $C$ and $B$, and $p$ indexes "the other dimension," we can call this the IJP ordering of the loops around the assignment statement.

The order in which the elements of $C$ are updated, and the order in which terms of

$$\alpha_{i,0}\beta_{0,j} + \cdots + \alpha_{i,k-1}\beta_{k-1,j}$$

are added to $\gamma_{i,j}$ is mathematically equivalent. (There would be a possible change in the result due to round-off errors when floating point arithmetic occurs, but this is ignored.) What this means is that the three loops can be reordered without changing the result[1].

**Homework 1.2.4.1** The IJP ordering is one possible ordering of the loops. How many distinct reorderings of those loops are there?

☛ SEE ANSWER

[1]In exact arithmetic, the result does not change. However, computation is typically performed in floating point arithmetic, in which case roundoff error may accumulate slightly differently, depending on the order of the loops. For more details, see Unit **??**.

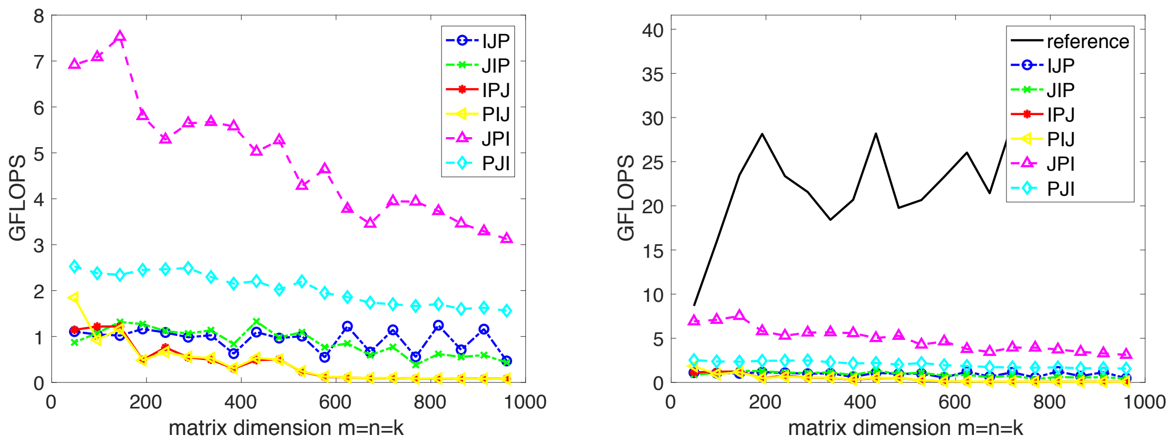


Figure 1.6: Performance comparison of all different orderings of the loops, on Robert's laptop. In the graph on the right, the performance of the reference implementation is added and the top of the graph is chosen to represent the theoretical peak of the processor. This captures that simple implementations aren't magically transformed into high-performance implementations by compilers or architectural features. The most pronounced "zigzagging" of the reference plot is likely due to other tasks being run on the same processor.

**Homework 1.2.4.2** In directory `Assignments/Week1/C/` make copies of ☛ `Gemm_IJP.c` into files with names that reflect the different loop orderings (`Gemm_JIP.c`, etc.). Next, make the necessary changes to the loops in each file to reflect the ordering encoded in its name. Test the implementions by executing `'make JIP'`, etc., for each of the implementations and view the resulting performance by making the indicated changes to the Live Script in ☛ data/Plot_All_Orderings.mlx). (Alternatively, use the script in ☛ `data/Plot_All_Orderings_m.m`.) If you have implemented them all, you can test them all by executing `'make All_Orderings'`.

☛ SEE ANSWER

**Homework 1.2.4.3** In Figure 1.6, the results of Homework 1.2.4.2 on Robert's laptop are reported. What do the two loop orderings that result in the best performances have in common?

☛ SEE ANSWER

What is obvious is that ordering the loops matters. Changing the order changes how data in memory are accessed, and that can have a considerable impact on the performance that is achieved. The bottom graph includes a plot of the performance of the reference implementation and scales the y-axis so that the top of the graph represents the theoretical peak of a single core of the processor. What it demonstrates is that there is a lot of room for improvement.

## 1.3 Thinking in Terms of Vector-Vector Operations

In this section, learners find out

- Our notational conventions for matrices and vectors.
- How to think of a matrix in terms of its rows or columns.
- How the inner-most loop of MMM expresses one of two vector-vector operations: the dot product (dot) or the update of a vector bu adding a multiple of another vector (AXPY).
- That vector-vector operations attained high performance on the vector supercomputers of the 1970s. In Week 2, we will see how similarly modern cores attain high performance with vector instructions.
- How to implement vector-vector operations with calls to the Basic Linear Algebra Subprograms (BLAS) and the alternative interface provided by the BLAS-like Library Instantiation Software (BLIS).
- 
- How to link to high-performance implementations of vector-vector operations.

### 1.3.1 The Basic Linear Algebra Subprograms (BLAS)

Linear algebra operations are fundamental to computational science. In the 1970s, when vector supercomputers reigned supreme, it was recognized that if applications and software libraries are written in terms of a standardized interface to routines that implement operations with vectors, and vendors of computers provide high-performance instantiations for that interface, then applications would attain portable high performance across different computer platforms. This observation yielded the original Basic Linear Algebra Subprograms (BLAS) interface [**?**] for Fortran 77, which are now referred to as the *level-1 BLAS*. The interface was expanded in the 1980s to encompass matrix-vector operations (level-2 BLAS) and matrix-matrix operations (level-3 BLAS). We will learn more about the level-2 and level-3 BLAS later in the course.

Expressing code in terms of the BLAS has another benefit: the call to the routine hides the loop that otherwise implements the vector-vector operation and clearly reveals the operation being performed, thus improving readability of the code.

### 1.3.2 Notation

In our discussions, we use capital letters for matrices ($A, B, C, \ldots$), lower case letters for vectors ($a, b, c, \ldots$), and lower case Greek letters for scalars ($\alpha, \beta, \gamma, \ldots$). Exceptions are integer scalars, for which we will use $i, j, k, m, n$, and $p$.

Vectors in our universe are column vectors or, equivalently, $n \times 1$ matrices if the vector has $n$ components (size $n$). A row vector we view as a column vector that has been transposed. So, $x$ is a column vector and $x^T$ is a row vector.

In the subsequent discussion, we will want to expose the rows or columns of a matrix. If $X$ is an $m \times n$ matrix, then we expose its columns as

$$X = \left( \begin{array}{c|c|c|c} x_0 & x_1 & \cdots & x_{n-1} \end{array} \right)$$

so that $x_j$ equals the column with index $j$. We expose its rows as

$$X = \begin{pmatrix} \widetilde{x}_0^T \\ \widetilde{x}_1^T \\ \vdots \\ \widetilde{x}_{m-1}^T \end{pmatrix}$$

so that $\widetilde{x}_i^T$ equals the row with index $i$. Here the $^T$ indicates it is a row (a column vector that has been transposed). The tilde is added for clarity since $x_i^T$ would in this setting equal the column indexed with $i$ that has been transposed, rather than the row indexed with $i$. When there isn't a cause for confusion, we will sometimes leave the $\widetilde{\ }$ off. We use the lower case letter that corresponds to the upper case letter used to denote the matrix, as an added visual clue that $x_j$ is a column of $X$ and $\widetilde{x}_i^T$ is a row of $X$.

We have already seen that the scalars that constitute the elements of a matrix or vector are denoted with the lower Greek letter that corresponds to the letter used for the matrix of vector:

$$X = \begin{pmatrix} \chi_{0,0} & \chi_{0,1} & \cdots & \chi_{0,n-1} \\ \chi_{1,0} & \chi_{1,1} & \cdots & \chi_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \chi_{m-1,0} & \chi_{m-1,1} & \cdots & \chi_{m-1,n-1} \end{pmatrix} \quad \text{and} \quad x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{m-1} \end{pmatrix}.$$

If you look carefully, you will notice the difference between $x$ and $\chi$. The latter is the lower case Greek letter "chi."

### 1.3.3 The dot product (inner product)

Given two vectors $x$ and $y$ of size $n$

$$x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{n-1} \end{pmatrix} \quad \text{and} \quad y = \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{n-1} \end{pmatrix},$$

their dot product is given by

$$x^T y = \sum_{i=0}^{n-1} \chi_i \psi_i.$$

The notation $x^T y$ comes from the fact that the dot product also equals the result of multiplying $1 \times n$ matrix $x^T$ times $n \times 1$ matrix $y$.

A routine. coded in C, that computes $x^T y + \gamma$ where $x$ and $y$ are stored at location `x` with stride `incx` and location `y` with stride `incy`, respectively, and $\gamma$ is stored at location `gamma` is given by

```
#define chi( i ) x[ (i)*incx ]   // map chi( i ) to array x
#define psi( i ) y[ (i)*incy ]   // map psi( i ) to array y

void Dots( int n, double *x, int incx, double *y, int incy, double *gamma )
{
  for ( int i=0; i<n; i++ )
    *gamma += chi( i ) * psi( i );
}
```

in ☛ `./Assignments/Week1/C/Dots.c`. Here stride refers to the number of items in memory between the stored components of the vector. For example, the stride when accessing a row of a matrix is `lda` when the matrix is stored in column-major order with leading dimension `lda`, .

The BLAS include a function for computing the dot operation. Its calling sequence in Fortran, for double precision data, is

```
DDOT( N, X, INCX, Y, INCY )
```

where

- (input) `N` is an integer that equals the size of the vectors.
- (input) `X` is the address where $x$ is stored.
- (input) `INCX` is the stride in memory between entries of $x$.
- (input) `Y` is the address where $y$ is stored.
- (input) `INCY` is the stride in memory between entries of $y$.

The function returns the result as a scalar of type double precision. If the datatype were single precision, complex double precision, or complex single precision, then the first `D` is replaced by `S`, `Z`, or `C`, respectively.

To call the same routine in a code written in C, it is important to keep in mind that Fortran passes data by address. The call

```
Dots( n, x, incx, y, incy, &gamma );
```

which, recall, adds the result of the dot product to the value in `gamma`, translates to

```
gamma += ddot_( &n, x, &incx, y, &incy );
```

When one of the strides equals one, as in

```
Dots( n, x, 1, y, incy, &gamma );
```

one has to declare an integer variable (e.g, `i_one`) with value one and pass the address of that variable:

```
int i_one=1;
gamma += ddot_( &n, x, &i_one, y, &incy );
```

We will see examples of this later in this section.

In this course, we use the BLIS implementation of the BLAS as our library. This library also has its own (native) BLAS-like interface. A "quickguide" for this BLIS native interface can be found at

https://github.com/flame/blis/blob/master/docs/BLISTypedAPI.md.

There, we find the routine `bli_ddotxv` that computes $\gamma := \alpha x^T y + \beta \gamma$, optionally conjugating the elements of the vectors. The call

```
Dots( n, x, incx, y, incy, &gamma );
```

translates to

```
#include "blis.h"

double one=1.0;

bli_ddotxv( BLIS_NO_CONJUGATE, BLIS_NO_CONJUGATE,
            n, &one, x, incx, y, incy,
               &one, &gamma );
```

The `BLIS_NO_CONJUGATE` is to indicate that the vectors are *not* to be conjugated. Those parameters are there for consistency with the complex versions of this routine (`bli_zdotxv` and `bli_cdotxv`).

### 1.3.4 The IJP and JIP orderings

Let us return once again to the IJP ordering of the loops that compute MMM:

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

This pseudo-code translates into the routine coded in C given in Figure 1.1.

Using the notation that we introduced in Unit 1.3.2, one way to think of the above algorithm is to view matrix $C$ as its individual elements, matrix $A$ as its rows, and matrix $B$ as its columns:

$$C = \left(\begin{array}{c|c|c|c} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,n-1} \\ \hline \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,n-1} \\ \hline \vdots & & \vdots & \\ \hline \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,n-1} \end{array}\right), \quad A = \left(\begin{array}{c} \widetilde{a}_0^T \\ \hline \widetilde{a}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T \end{array}\right), \quad \text{and } B = \left(\begin{array}{c|c|c|c} b_0 & b_1 & \cdots & b_{n-1} \end{array}\right).$$

We then notice that

$$\begin{array}{l} \left(\begin{array}{c|c|c|c} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,n-1} \\ \hline \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,n-1} \end{array}\right) \\ := \left(\begin{array}{c} \widetilde{a}_0^T \\ \hline \widetilde{a}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T \end{array}\right) \left(\begin{array}{c|c|c|c} b_0 & b_1 & \cdots & b_{n-1} \end{array}\right) + \left(\begin{array}{c|c|c|c} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,n-1} \\ \hline \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,n-1} \end{array}\right) \\ = \left(\begin{array}{c|c|c|c} \widetilde{a}_0^T b_0 + \gamma_{0,0} & \widetilde{a}_0^T b_1 + \gamma_{0,1} & \cdots & \widetilde{a}_0^T b_{n-1} + \gamma_{0,n-1} \\ \hline \widetilde{a}_1^T b_0 + \gamma_{1,0} & \widetilde{a}_1^T b_1 + \gamma_{1,1} & \cdots & \widetilde{a}_1^T b_{n-1} + \gamma_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \widetilde{a}_{m-1}^T b_0 + \gamma_{m-1,0} & \widetilde{a}_{m-1}^T b_1 + \gamma_{m-1,1} & \cdots & \widetilde{a}_{m-1}^T b_{n-1} + \gamma_{m-1,n-1} \end{array}\right). \end{array}$$

If this makes your head spin, you may want to quickly go through Weeks 3-5 of our MOOC titled "Linear Algebra: Foundations to Fontiers." It captures that the outer two loops visit all of the elements in $C$, and the inner loop implements the dot product of the appropriate row of $A$ with the appropriate column of $B$: $\gamma_{i,j} := \widetilde{a}_i^T b_j + \gamma_{i,j}$, as illustrated by

```
for $i := 0, \ldots, m-1$
  for $j := 0, \ldots, n-1$

    for $p := 0, \ldots, k-1$                    \
        $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$      }  $\gamma_{i,j} := \widetilde{a}_i^T b_j + \gamma_{i,j}$
    end                                          /

  end
end
```

which is, again, the IJP ordering of the loops.

**Homework 1.3.4.1** In directory `Assignments/Week1/C/` copy file ☛ `Gemm_IJP.c` into file `Gemm_IJ_Dots.c`. Replace the inner-most loop with an appropriate call to `Dots`, and compile and execute them with

```
make IJ_Dots
```

View the resulting performance by making the necessary changes to the Live Script in ☛ `data/Plot_Inner_P.mlx`. (Alternatively, use the script in ☛ `data/Plot_Inner_P_m.m`.)

☛ SEE ANSWER

**Homework 1.3.4.2** In directory `Assignments/Week1/C/` copy file ☛ `Gemm_IJ_Dots.c` into file `Gemm_IJ_ddot.c`. Replace the call to `Dots` to a call to the BLAS routine `ddot`, and compile and execute them with

```
make IJ_ddot
```

View the resulting performance by making the necessary changes to the Live Script in ☛ `data/Plot_Inner_P.mlx`. (Alternatively, use the script in ☛ `data/Plot_Inner_P_m.m`.)

☛ SEE ANSWER

**Homework 1.3.4.3** In directory `Assignments/Week1/C/` copy file ☛ `Gemm_IJ_Dots.c` into file `Gemm_IJ_ddotxv.c`. Replace the call to `Dots` to a call to the BLIS routine `bli_bli_ddotxv`, and compile and execute them with

```
make IJ_bli_ddotxv
```

View the resulting performance by making the necessary changes to the Live Script in ☛ `data/Plot_Inner_P.mlx`. (Alternatively, use the script in ☛ `data/Plot_Inner_P_m.m`.)

☛ SEE ANSWER

Obviously, one can equally well switch the order of the outer two loops, which just means that the elements of $C$ are computed a column at a time rather than a row at a time:

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

**Homework 1.3.4.4** Repeat the last exercises with the implementation in ☛ `Gemm_JIP.c`. In other words, copy this file into files `Gemm_JI_Dots.c`, `Gemm_JI_ddot.c`, and `Gemm_JI_bli_ddotxv.c`. Make the necessary changes to these file, and compile and execute them with

```
make JI_Dots
make JI_ddot
make JI_bli_ddotxv
```

View the resulting performance with the Live Script in ☛ `data/Plot_Inner_P.mlx`. (Alternatively, use the script in ☛ `data/Plot_Inner_P_m.m`.)

☛ SEE ANSWER

In Figure 1.7 we report the performance of the various implementations from the last homeworks. What we notice is that, at least when using Apple's clang compiler, not much difference results from hiding the inner-most loop in a subroutine.

### 1.3.5 The AXPY operation

Given a scalar, $\alpha$, and two vectors, $x$ and $y$, of size $n$ with elements

$$x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{n-1} \end{pmatrix} \quad \text{and} \quad y = \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{n-1} \end{pmatrix},$$

the scaled vector addition (AXPY) operation is given by

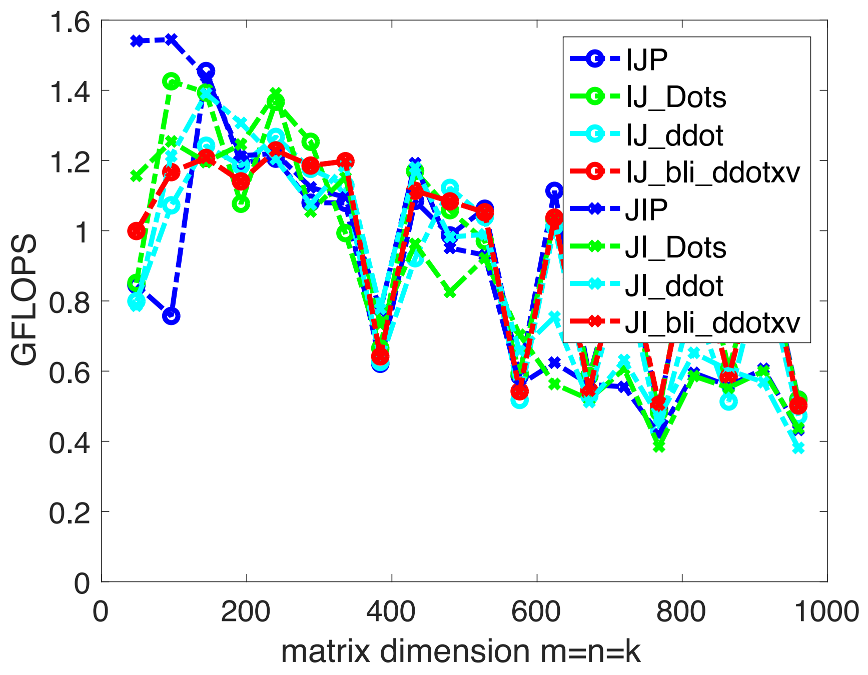
$$y := \alpha x + y$$

which in terms of the elements of the vectors equals

$$\begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{n-1} \end{pmatrix} := \alpha \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{n-1} \end{pmatrix} + \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{n-1} \end{pmatrix} = \begin{pmatrix} \alpha\chi_0 + \psi_0 \\ \alpha\chi_1 + \psi_1 \\ \vdots \\ \alpha\chi_{n-1} + \psi_{n-1} \end{pmatrix}.$$

The name `axpy` comes from the fact that in Fortran 77 only six characters and numbers could be used to designate the names of variables and functions. The operation $\alpha x + y$ can be read out loud as "scalar Alpha times X Plus Y" which yields the acronym AXPY.

An outline for a routine that implements the AXPY operation is given by



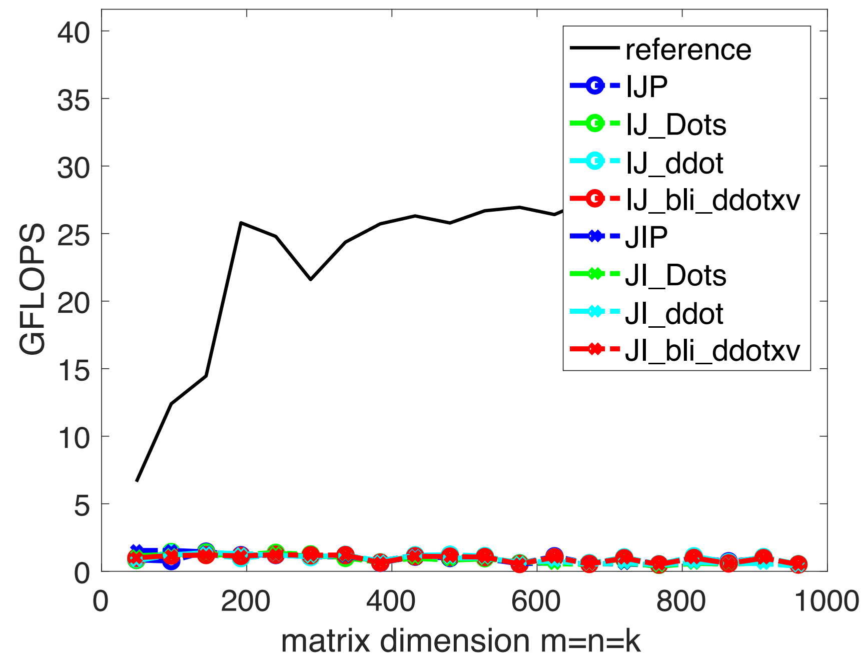


Figure 1.7: Performance of the various implementations of the IJP and JIP orderings. In the graph on the bottom, we include the reference to illustrate there is still a lot of room for improvement.

```
#define chi( i ) x[ (i)*incx ]   // map chi( i ) to array x
#define psi( i ) y[ (i)*incy ]   // map psi( i ) to array y

void Axpy( int n, double alpha, double *x, int incx, double *y, int incy )
{
  for ( int i=0; i<n; i++ )
    psi(i)  +=
}
```

in file ☛ `Axpy.c`.

**Homework 1.3.5.1** Complete the routine in `Axpy.c`. You can find the partial implementation in `Axpy.c`. You will test the implementation by using it in subsequent homeworks.

☛ SEE ANSWER

The BLAS include a function for computing the AXPY operation. Its calling sequence in Fortran, for double precision data, is

```
DAXPY( N, ALPHA, X, INCX, Y, INCY )
```

where

- (input) `N` is the address of the integer that equals the size of the vectors.
- (input) `ALPHA` is the address where $\alpha$ is stored.
- (input) `X` is the address where $x$ is stored.
- (input) `INCX` is the address where the stride between entries of $x$ is stored.
- (input/output) `Y` is the address where $y$ is stored.
- (input) `INCY` is the address where the stride between entries of $y$ is stored.

(It may appear strange that the addresses of `N`, `ALPHA`, `INCX`, and `INCY` are passed in. This is because Fortran passes parameters by address rather than value.) If the datatype were single precision, complex double precision, or complex single precision, then the first `D` is replaced by `S`, `Z`, or `C`, respectively.

In C, the call

```
Axpy( n, alpha, x, incx, y, incy );
```

translates to

```
daxpy_( &n, &alpha, x, &incx, y, &incy );
```

which then calls the Fortran interface. Notice that the scalar parameters `n`, `alpha`, `incx`, and `incy` are passed "by address" because Fortran passes parameters to subroutines by address. We will see examples of its use later in this section.

The BLIS native routine for AXPY is `bli_dapyv`. The call

```
Axpy( n, alpha, x, incx, y, incy );
```

translates to

```
bli_daxpyv( BLIS_NO_CONJUGATE, n, alpha, x, incx, y, incy );
```

The `BLIS_NO_CONJUGATE` is to indicate that vector $x$ is *not* to be conjugated. That parameter is there for consistency with the complex versions of this routine (`bli_zaxpyv` and `bli_caxpyv`).

### 1.3.6 The IPJ and PIJ orderings

What we notice is that there are 3! ways of order the loops: Three choices for the outer loop, two for the second loop, and one choice for the final loop. Let's consider the IPJ ordering:

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

One way to think of the above algorithm is to view matrix $C$ as its rows, matrix $A$ as its individual elements, and matrix $B$ as its rows:

$$C = \left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right), \quad A = \left(\begin{array}{c|c|c|c} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,k-1} \\ \hline \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,k-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,k-1} \end{array}\right), \quad \text{and} \quad B = \left(\begin{array}{c} \widetilde{b}_0^T \\ \hline \widetilde{b}_1^T \\ \hline \vdots \\ \hline \widetilde{b}_{k-1}^T \end{array}\right).$$

We then notice that

$$\begin{aligned} \left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right) &:= \left(\begin{array}{c|c|c|c} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,k-1} \\ \hline \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,k-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,k-1} \end{array}\right) \left(\begin{array}{c} \widetilde{b}_0^T \\ \hline \widetilde{b}_1^T \\ \hline \vdots \\ \hline \widetilde{b}_{k-1}^T \end{array}\right) + \left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right) \\ &= \left(\begin{array}{ccccccccc} \widetilde{c}_0^T & + & \alpha_{0,0}\widehat{b}_0^T & + & \alpha_{0,1}\widehat{b}_1^T & + & \cdots & + & \alpha_{0,k-1}\widehat{b}_{k-1}^T \\ \hline \widetilde{c}_1^T & + & \alpha_{1,0}\widehat{b}_0^T & + & \alpha_{1,1}\widehat{b}_1^T & + & \cdots & + & \alpha_{1,k-1}\widehat{b}_{k-1}^T \\ \hline & & & & \vdots & & & & \\ \hline \widetilde{c}_{m-1}^T & + & \alpha_{m-1,0}\widehat{b}_0^T & + & \alpha_{m-1,1}\widehat{b}_1^T & + & \cdots & + & \alpha_{m-1,k-1}\widehat{b}_{k-1}^T \end{array}\right). \end{aligned}$$

This captures that the outer two loops visit all of the elements in $A$, and the inner loop implements the updating of the $i$th row of $C$ by adding $\alpha_{i,p}$ times the $p$th row of $B$ to it, as captured by

```
for $i := 0, \ldots, m-1$
  for $p := 0, \ldots, k-1$
    for $j := 0, \ldots, n-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$        } $\widetilde{c}_i^T := \alpha_{i,p}\widetilde{b}_p^T + \widetilde{c}_i$
    end
  end
end
```

**Homework 1.3.6.1** In directory `Assignments/Week1/C/` copy file ☛ `Gemm_IPJ.c` into file `Gemm_IP_Axpy.c`. Replace the inner-most loop with a call to `Axpy`, and compile and execute them with

```
make IP_Axpy
```

View the resulting performance by making the necessary changes to the Live Script in `data/Plot_Inner_J.mlx`. (Alternatively, use the script in ☛ `data/Plot_Inner_J_m.m`.)

☛ SEE ANSWER

**Homework 1.3.6.2** In directory `Assignments/Week1/C/` copy file ☛ `Gemm_IPJ.c` into file `Gemm_IP_daxpy.c`. Replace the inner-most loop with a call to the BLAS routine `daxpy`, and compile and execute them with

```
make IP_daxpy
```

View the resulting performance by making the necessary changes to the Live Script in `data/Plot_Inner_J.mlx`. (Alternatively, use the script in ☛ `data/Plot_Inner_J_m.m`.)

☛ SEE ANSWER

**Homework 1.3.6.3** In directory `Assignments/Week1/C/` copy file ☛ `Gemm_IPJ.c` into file `Gemm_IP_bli_daxpyv.c`, and compile and execute them with

```
make IP_bli_daxpyv
```

Replace the inner-most loop with a call to the BLIS routine `bli_daxpyv`. View the resulting performance by making the necessary changes to the Live Script in `data/Plot_Inner_J.mlx`. (Alternatively, use the script in ☛ `data/Plot_Inner_J_m.m`.)

☛ SEE ANSWER

One can switch the order of the outer two loops to get

```
**for** $p := 0, \ldots, k-1$
  **for** $i := 0, \ldots, m-1$
    **for** $j := 0, \ldots, n-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$            } $\widetilde{c}_i^T := \alpha_{i,p}\widehat{b}_p^T + \widetilde{c}_i^T$
    **end**
  **end**
**end**
```

The outer loop in this second algorithm fixes the row of $B$ that is used to to update all rows of $C$, using the appropriate element from $A$ to scale. In the first iteration of the outer loop ($p = 0$), the following computations occur:

$$\left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right) := \left(\begin{array}{ccc} \widetilde{c}_0^T & + & \alpha_{0,0}\widehat{b}_0^T \\ \hline \widetilde{c}_1^T & + & \alpha_{1,0}\widehat{b}_0^T \\ \hline & \vdots & \\ \hline \widetilde{c}_{m-1}^T & + & \alpha_{m-1,0}\widehat{b}_0^T \end{array}\right).$$

In the second iteration of the outer loop ($p = 1$) it computes

$$\left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right) := \left(\begin{array}{ccccc} \widetilde{c}_0^T & + & \alpha_{0,0}\widehat{b}_0^T & + & \alpha_{0,1}\widehat{b}_1^T \\ \hline \widetilde{c}_1^T & + & \alpha_{1,0}\widehat{b}_0^T & + & \alpha_{1,1}\widehat{b}_1^T \\ \hline & & & \vdots & \\ \hline \widetilde{c}_{m-1}^T & + & \alpha_{m-1,0}\widehat{b}_0^T & + & \alpha_{m-1,1}\widehat{b}_1^T \end{array}\right),$$

and so forth.

**Homework 1.3.6.4** Repeat the last exercises with the implementation in ☛ `Gemm_PIJ.c`. In other words, copy this file into files `Gemm_PI_Axpy.c`, `Gemm_PI_daxpy.c`, and `Gemm_PI_bli_daxpyv.c`. Make the necessary changes to these files, and compile and execute them with

```
make PI_Axpy
make PI_daxpy
make PI_bli_daxpyv
```

View the resulting performance with the Live Script in ☛ `data/Plot_Inner_J.mlx`. (Alternatively, use the script in ☛ `data/Plot_Inner_J_m.m`.)

☛ SEE ANSWER

## 1.3.7 The JPI and PJI orderings

Let us consider the JPI ordering:

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

Another way to think of the above algorithm is to view matrix $C$ as its columns, matrix $A$ as its columns, and matrix $B$ as its individual elements. Then

$$\left( \begin{array}{c|c|c|c} c_0 & c_1 & \cdots & c_{n-1} \end{array} \right) :=$$

$$\left( \begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{k-1} \end{array} \right) \left( \begin{array}{c|c|c|c} \beta_{0,0} & \beta_{0,1} & \cdots & \beta_{0,n-1} \\ \hline \beta_{1,0} & \beta_{1,1} & \cdots & \beta_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \beta_{k-1,0} & \beta_{k-1,1} & \cdots & \beta_{k-1,n-1} \end{array} \right) + \left( \begin{array}{c|c|c|c} c_0 & c_1 & \cdots & c_{n-1} \end{array} \right).$$

so that

$$\left( \begin{array}{c|c|c|c} c_0 & c_1 & \cdots & c_{n-1} \end{array} \right) :=$$
$$\left( \begin{array}{c|c|c|c} c_0 + \beta_{0,0}a_0 + \beta_{1,0}a_1 + \cdots & c_1 + \beta_{0,1}a_0 + \beta_{1,1}a_1 + \cdots & \cdots & c_{n-1} + \beta_{0,n-1}a_0 + \beta_{1,n-1}a_1 + \cdots \end{array} \right).$$

The algorithm captures this as

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

**Homework 1.3.7.1** In directory `Assignments/Week1/C/` copy file ☛ `Gemm_JPI.c` into file `Gemm_JP_Axpy.c`. Replace the inner-most loop with a call to `Axpy`, and compile and execute with

```
make JP_Axpy
```

View the resulting performance by making the necessary changes to the Live Script in `data/Plot_Inner_I.mlx`. (Alternatively, use the script in ☛ `data/Plot_Inner_I_m.m`.)

☛ SEE ANSWER

**Homework 1.3.7.2** In directory `Assignments/Week1/C/` copy file ☛ `Gemm_JPI.c` into file `Gemm_JP_daxpy.c`. Replace the inner-most loop with a call to the BLAS routine `daxpy`, and compile and execute with

```
make JP_daxpy
```

View the resulting performance by making the necessary changes to the Live Script in ☛ `data/Plot_Inner_I.mlx`. (Alternatively, use the script in ☛ `data/Plot_Inner_I_m.m`.)

☛ SEE ANSWER

**Homework 1.3.7.3** In directory `Assignments/Week1/C/` copy file ☛ `Gemm_JPI.c` into file `Gemm_JP_bli_daxpyv.c`. Replace the inner-most loop with a call to the BLIS routine `bli_daxpyv`, and compile and execute with

```
make JP_bli_daxpyv
```

View the resulting performance by making the necessary changes to the Live Script in ☛ `data/Plot_Inner_I.mlx`. (Alternatively, use the script in ☛ `data/Plot_Inner_I_m.m`.)

☛ SEE ANSWER

One can switch the order of the outer two loops to get

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

The outer loop in this algorithm fixes the column of $A$ that is used to to update all columns of $C$, using the appropriate element from $B$ to scale. In the first iteration of the outer loop, the following computations occur:

$$\left( \begin{array}{c|c|c|c} c_0 & c_1 & \cdots & c_{n-1} \end{array} \right) := \left( \begin{array}{c|c|c|c} c_0 + \beta_{0,0} a_0 & c_1 + \beta_{0,1} a_0 & \cdots & c_{n-1} + \beta_{0,n-1} a_0 \end{array} \right).$$

In the second iteration of the outer loop it computes

$$\left( \begin{array}{c|c|c|c} c_0 & c_1 & \cdots & c_{n-1} \end{array} \right) := \left( \begin{array}{c|c|c|c} c_0 + \beta_{0,0} a_0 + \beta_{1,0} a_1 & c_1 + \beta_{0,1} a_0 + \beta_{1,1} a_1 & \cdots & c_{n-1} + \beta_{0,n-1} a_0 + \beta_{1,n-1} a_1 \end{array} \right).$$

and so forth.

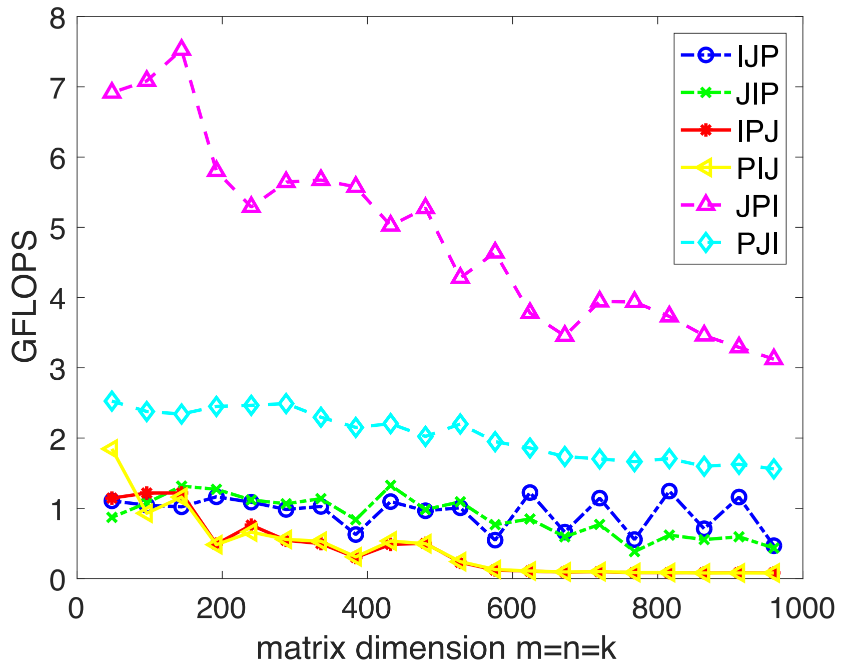


Figure 1.8: Here we repeat the graph from Figure 1.6, reporting the performance of all different orderings of the loops, on Robert's laptop.

**Homework 1.3.7.4** Repeat the last exercises with the implementation in ☛ `Gemm_PJI.c`. In other words, copy this file into files `Gemm_PJ_Axpy.c`, `Gemm_PJ_daxpy.c`, and `Gemm_PJ_bli_daxpyv.c`. Then, make the necessary changes to these files, and compile and execute with

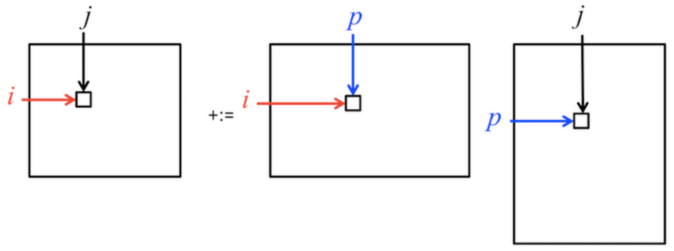
```
make PJ_Axpy
make PJ_daxpy
make PJ_bli_daxpyv
```

View the resulting performance by making the necessary changes to the Live Script in ☛ `data/Plot_Inner_I.mlx`. (Alternatively, use the script in ☛ `data/Plot_Inner_I_m.m`.)

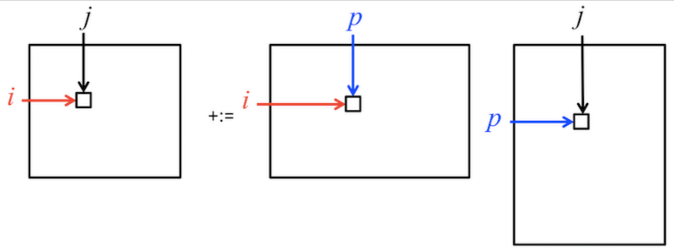
☛ SEE ANSWER

Draw what the inner-most loop computes

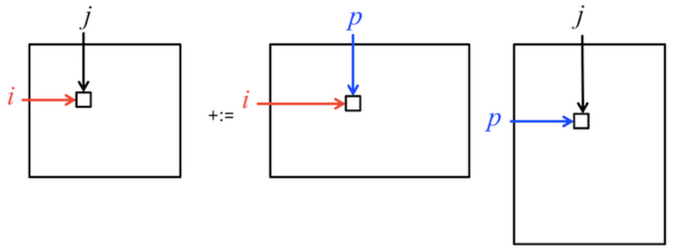
```
for $i := 0, \ldots, m-1$
  for $j := 0, \ldots, n-1$
    for $p := 0, \ldots, k-1$                              }
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$   } $\gamma_{i,j} := \widetilde{a}_i^T b_j + \gamma_{i,j}$
    end                                                    }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



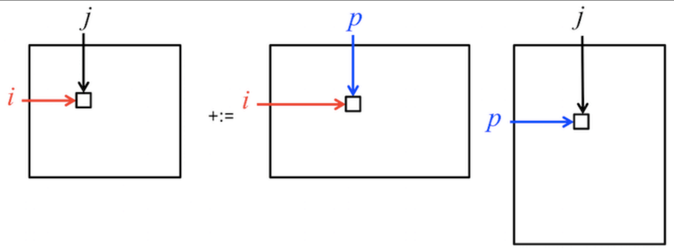
```
for $i := 0, \ldots, m-1$
  for $p := 0, \ldots, k-1$
    for $j := 0, \ldots, n-1$                              }
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$   } $\widetilde{c}_i^T := \alpha_{i,p}\widetilde{b}_p^T + \widetilde{c}_i$
    end                                                    }
  end
end
```



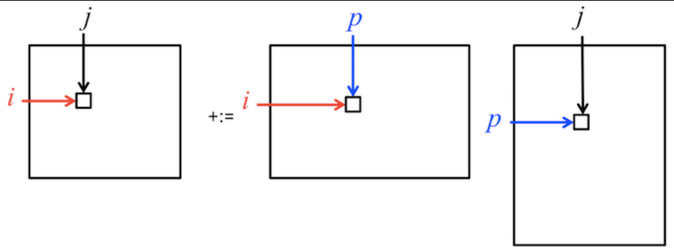
```
for $j := 0, \ldots, n-1$
  for $i := 0, \ldots, m-1$
    for $p := 0, \ldots, k-1$                              }
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$   } $\gamma_{i,j} := \widetilde{a}_i^T b_j + \gamma_{i,j}$
    end                                                    }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



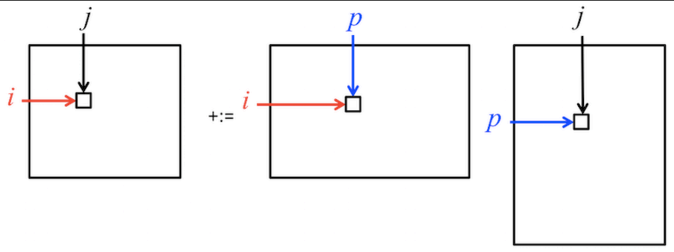
```
for $j := 0, \ldots, n-1$
  for $p := 0, \ldots, k-1$
    for $i := 0, \ldots, m-1$                              }
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$   } $c_j := \beta_{p,j} a_p + c_j$
    end                                                    }
  end
end
```



```
for $p := 0, \ldots, k-1$
  for $i := 0, \ldots, m-1$
    for $j := 0, \ldots, n-1$                              }
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$   } $\widetilde{c}_i^T := \alpha_{i,p}\widetilde{b}_p^T + \widetilde{c}_i^T$
    end                                                    }
  end
end
```



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



### 1.3.8 Discussion

**Homework 1.3.8.1** In Figure 1.8, the results of Homework 1.2.4.2 on Robert's laptop are again reported. What do the two loop orderings that result in the best performances have in common? **??**

☛ SEE ANSWER

**Homework 1.3.8.2** In Homework **??**, why do they get better performance?

☛ SEE ANSWER

**Homework 1.3.8.3** In Homework **??**, why does the implementation that gets the best performance outperform the one that gets the next to best performance?

☛ SEE ANSWER

## 1.4 Thinking in Terms of Matrix-Vector Operations

### 1.4.1 Matrix-vector multiplication via dot products or AXPY operations

**Homework 1.4.1.1**

Compute $\begin{pmatrix} 2 & 2 & -1 & 2 \\ 2 & 1 & 0 & -2 \\ -2 & -2 & 2 & 2 \end{pmatrix} \begin{pmatrix} 2 \\ -1 \\ 0 \\ -1 \end{pmatrix} =$

☛ SEE ANSWER

Consider the matrix-vector multiplication (MVM)

$$y := Ax + y.$$

The way one is usually taught to compute this operations is that each element of $y$, $\psi_i$, is updated with the dot product of the corresponding row of $A$, $\widetilde{a}_i^T$, with vector $x$. With our notation, we can describe this as

$$\left(\begin{array}{c} \psi_0 \\ \hline \psi_1 \\ \hline \vdots \\ \hline \psi_{m-1} \end{array}\right) := \left(\begin{array}{c} \widetilde{a}_0^T \\ \hline \widetilde{a}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T \end{array}\right) x + \left(\begin{array}{c} \psi_0 \\ \hline \psi_1 \\ \hline \vdots \\ \hline \psi_{m-1} \end{array}\right) = \left(\begin{array}{c} \widetilde{a}_0^T x + \psi_0 \\ \hline \widetilde{a}_1^T x + \psi_1 \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T x + \psi_{m-1} \end{array}\right).$$

If we then expose the individual elements of $A$ and $y$ we get

$$\begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{m-1} \end{pmatrix} := \begin{pmatrix} \alpha_{0,0} \; \chi_0 + \alpha_{0,1} \; \chi_1 + \cdots \alpha_{0,n-1} \; \chi_{n-1} + \psi_0 \\ \alpha_{1,0} \; \chi_0 + \alpha_{1,1} \; \chi_1 + \cdots \alpha_{1,n-1} \; \chi_{n-1} + \psi_1 \\ \vdots \\ \alpha_{m-1,0} \, \chi_0 + \alpha_{m-1,1} \, \chi_1 + \cdots \alpha_{m-1,n-1} \, \chi_{n-1} + \psi_{m-1} \end{pmatrix}$$

$$= \begin{pmatrix} \chi_0 \; \alpha_{0,0} + \chi_1 \; \alpha_{0,1} + \cdots \chi_{n-1} \; \alpha_{0,n-1} + \psi_0 \\ \chi_0 \; \alpha_{1,0} + \chi_1 \; \alpha_{1,1} + \cdots \chi_{n-1} \; \alpha_{1,n-1} + \psi_1 \\ \vdots \\ \chi_0 \, \alpha_{m-1,0} + \chi_1 \, \alpha_{m-1,1} + \cdots \chi_{n-1} \, \alpha_{m-1,n-1} + \psi_{m-1} \end{pmatrix}$$

This discussion explains the IJ loop for computing $y := Ax + y$:

```
for $i := 0, \ldots, m-1$
  for $j := 0, \ldots, n-1$
      $\psi_i := \alpha_{i,j}\chi_j + \psi_i$          } $\psi_i := \widetilde{a}_j^T x + \psi_i$
  end
end
```

What it demonstrates is how matrix-vector multiplication can be implemented as a sequence of DOT operations.

**Homework 1.4.1.2** In directory `Assignments/Week1/C/` complete the implementation of matrix-vector multiplication in terms of dot operations

```c
#define alpha( i,j ) A[ (j)*ldA + i ]   // map alpha( i,j ) to array A
#define chi( i )  x[ (i)*incx ]         // map chi( i )  to array x
#define psi( i )  y[ (i)*incy ]         // map psi( i )  to array x

void Dots( int, const double *, int, const double *, int, double * );

void Gemv( int m, int n, double *A, int ldA,
           double *x, int incx, double *y, int incy )
{
  for ( int i=0; i<m; i++ )
    Dots(   ,      ,     ,     ,    ,      );
}
```

in file ☛ `Gemv_I_Dots.c`. You will test this with an implementation of matrix-matrix multiplication in a later homework.

☛ SEE ANSWER

Next, partitioning $m \times n$ matrix $A$ by columns and $x$ by individual elements, we find that

$$\begin{aligned} y &:= \left( \begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{n-1} \end{array} \right) \left( \begin{array}{c} \chi_0 \\ \hline \chi_1 \\ \hline \vdots \\ \hline \chi_{n-1} \end{array} \right) + y \\ &= \chi_0 a_0 + \chi_1 a_1 + \cdots + \chi_{n-1} a_{n-1} + y. \end{aligned}$$

If we then expose the individual elements of $A$ and $y$ we get

$$\begin{aligned} \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{m-1} \end{pmatrix} &:= \chi_0 \begin{pmatrix} \alpha_{0,0} \\ \alpha_{1,0} \\ \vdots \\ \alpha_{m-1,0} \end{pmatrix} + \chi_1 \begin{pmatrix} \alpha_{0,1} \\ \alpha_{1,1} \\ \vdots \\ \alpha_{m-1,1} \end{pmatrix} + \cdots + \chi_{n-1} \begin{pmatrix} \alpha_{0,n-1} \\ \alpha_{1,n-1} \\ \vdots \\ \alpha_{m-1,n-1} \end{pmatrix} + \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{m-1} \end{pmatrix} \\ &= \begin{pmatrix} \chi_0 \; \alpha_{0,0} + \chi_1 \; \alpha_{0,1} + \cdots \chi_{n-1} \; \alpha_{0,n-1} + \psi_0 \\ \chi_0 \; \alpha_{1,0} + \chi_1 \; \alpha_{1,1} + \cdots \chi_{n-1} \; \alpha_{1,n-1} + \psi_1 \\ \vdots \\ \chi_0 \; \alpha_{m-1,0} + \chi_1 \; \alpha_{m-1,1} + \cdots \chi_{n-1} \; \alpha_{m-1,n-1} + \psi_{m-1} \end{pmatrix} \end{aligned}$$

This discussion explains the JI loop for computing $y := Ax + y$:

```
for $j := 0, \ldots, n-1$
  for $i := 0, \ldots, m-1$
     $\psi_i := \alpha_{i,j}\chi_j + \psi_i$      } $y := \chi_j a_j + y$
  end
end
```

What it also demonstrates is how matrix-vector multiplication can be implemented as a sequence of AXPY operations.

**Homework 1.4.1.3** In directory `Assignments/Week1/C/` complete the implementation of matrix-vector multiplication in terms of AXPY operations

```
#define alpha( i,j ) A[ (j)*ldA + i ]   // map alpha( i,j ) to array A
#define chi( i )  x[ (i)*incx ]         // map chi( i )  to array x
#define psi( i )  y[ (i)*incy ]         // map psi( i )  to array x

void Dots( int, const double *, int, const double *, int, double * );

void Gemv( int m, int n, double *A, int ldA,
           double *x, int incx, double *y, int incy )
{
  for ( int j=0; j<n; j++ )
    Axpy(   ,    ,    ,    ,    ,     );
}
```

in file ☛ `Gemv_J_Axpy.c`. You will test this with an implementation of matrix-matrix multiplication in a later homework.

☛ SEE ANSWER

## 1.4.2 Matrix-matrix multiplication via matrix-vector multiplications

**Homework 1.4.2.1**
Fill in the blanks:

$$\begin{pmatrix} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{pmatrix} \left(\begin{array}{c|c|c} 1 & -2 & 0 \\ 2 & -1 & 3 \end{array}\right) + \left(\begin{array}{c|c|c} 3 & 0 & -4 \\ -2 & 1 & -3 \\ 1 & -1 & -2 \end{array}\right) =$$

$$\left(\begin{array}{c|c|c} -1\times\ 1 + 2\times 2 + 3 & -1\times -2 + 2\times -1 + 0 & -1\times\ 0 + 2\times 3 - 4 \\ \boxed{\phantom{0}}\times\boxed{\phantom{0}} + \boxed{\phantom{0}}\times\boxed{\phantom{0}} - 2 & \boxed{\phantom{0}}\times\boxed{\phantom{0}} + \boxed{\phantom{0}}\times\boxed{\phantom{0}} + 1 & \boxed{\phantom{0}}\times\boxed{\phantom{0}} + \boxed{\phantom{0}}\times\boxed{\phantom{0}} - 3 \\ \boxed{\phantom{0}}\times\boxed{\phantom{0}} + \boxed{\phantom{0}}\times\boxed{\phantom{0}} + 1 & \boxed{\phantom{0}}\times\boxed{\phantom{0}} + \boxed{\phantom{0}}\times\boxed{\phantom{0}} - 1 & \boxed{\phantom{0}}\times\boxed{\phantom{0}} + \boxed{\phantom{0}}\times\boxed{\phantom{0}} - 2 \end{array}\right) =$$

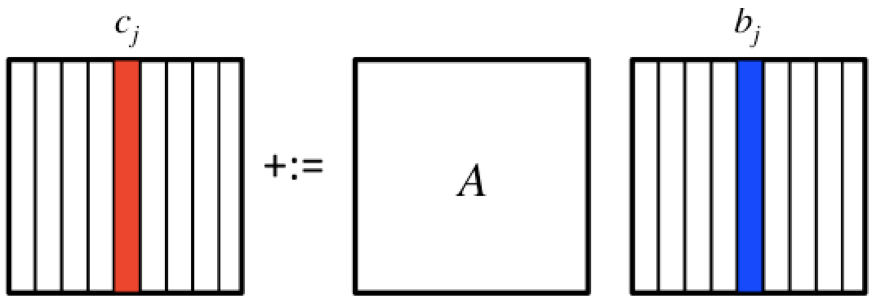
$$\left(\begin{array}{c|c|c} \begin{pmatrix} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{pmatrix} \begin{pmatrix} \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \end{pmatrix} + \begin{pmatrix} 3 \\ -2 \\ 1 \end{pmatrix} & \begin{pmatrix} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{pmatrix} \begin{pmatrix} \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix} & \begin{pmatrix} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{pmatrix} \begin{pmatrix} \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \end{pmatrix} + \begin{pmatrix} -4 \\ -3 \\ -2 \end{pmatrix} \end{array}\right)$$

☛ SEE ANSWER

Now that we are getting comfortable with partitioning matrices and vectors, we can view the six algorithms for $C := AB + C$ is a more layered fashion. If we partition $C$ and $B$ by columns, we find that

$$\begin{array}{rcl} \left( c_0 \mid c_1 \mid \cdots \mid c_{n-1} \right) & := & A \left( b_0 \mid b_1 \mid \cdots \mid b_{n-1} \right) + \left( c_0 \mid c_1 \mid \cdots \mid c_{n-1} \right) \\ & = & \left( Ab_0 + c_0 \mid Ab_1 + c_1 \mid \cdots \mid Ab_{n-1} + c_{n-1} \right) \end{array}$$

A picture that captures this is given by



This illustrates how the JIP and JPI algorithms can be viewed as a loop around matrix-vector multiplications:

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

and

```
for $j := 0, \ldots, n-1$
  for $i := 0, \ldots, m-1$
      for $p := 0, \ldots, k-1$                         }                                     }
          $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$   }  $\gamma_{i,j} := \widetilde{a}_i^T b_j + \gamma_{i,j}$   }  $c_j := Ab_j + c_j$
      end                                          }                                     }
    end                                                                                  }
end
```

**Homework 1.4.2.2** Complete the code in `Assignments/Week1/C/Gemm_J_Gemv.c`. Test two versions:

```
make J_Gemv_I_Dots
make J_Gemv_J_Axpy
```

View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Outer_J.mlx`. (Alternatively, use the script in ☛ `data/Plot_Outer_J_m.m`.)

☛ SEE ANSWER

The BLAS include the routine `dgemv` that computes

$$y := \alpha A x + \beta y \quad \text{or} \quad y := \alpha A^T x + \beta y.$$

If

- $m \times n$ matrix $A$ is stored in array `A` with its leading dimension stored in variable`ldA`,
- $m$ is stored in variable `m` and $n$ is stored in variable `n`,
- vector $x$ is stored in array `x` with its stride stored in variable `incx`,
- vector $y$ is stored in array `y` with its stride stored in variable `incy`, and
- $\alpha$ and $\beta$ are stored in variables `alpha` and `beta`, respectively,

the $y := \alpha A x + \beta y$ translates, from C, into the call

```
dgemv_( "No transpose", &m, &n, &alpha, A, &ldA, x, &incx, &beta, y, &incy );
```

**Homework 1.4.2.3** Complete the code in `Assignments/Week1/C/Gemm_J_dgemv.c` that casts matrix-matrix multiplication in terms of the `dgemv` BLAS routine. Test it by executing

```
make J_dgemv
```

View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Outer_J.mlx`. (Alternatively, use the script in ☛ `data/Plot_Outer_J_m.m`.)

☛ SEE ANSWER

The BLIS "native call" that is similar to the BLAS `dgemv` routine in this setting translates to

```
bli_dgemv( BLIS_NO_TRANSPOSE, BLIS_NO_CONJUGATE, m, n, &alpha,
           A, 1, ldA, x, incx, &beta, y, incy );
```

Notice the two parameters after `A`. BLIS requires a row and column stride to be specified for matrices, thus generalizing beyond column-major order storage.

**Homework 1.4.2.4** Complete the code in `Assignments/Week1/C/Gemm_J_bli_dgemv.c` that casts matrix-matrix multiplication in terms of the `bli_dgemv` BLIS routine. Test it by executing

```
make J_bli_dgemv
```

View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Outer_J.mlx`. (Alternatively, use the script in ☛ `data/Plot_Outer_J_m.m`.)

☛ SEE ANSWER

**Homework 1.4.2.5** What do you notice from the various performance graphs that result from executing the Live Script in ☛ `Plot_Outer_J.mlx`?

☛ SEE ANSWER

## 1.4.3 Rank-1 update RANK-1

An operation that is going to become very important in future discussion and optimization of MMM is the rank-1 update:

$$A := xy^T + A.$$

**Homework 1.4.3.1**

Compute $\begin{pmatrix} 2 \\ -1 \\ 0 \end{pmatrix} \begin{pmatrix} -2 & 0 & 1 & -1 \end{pmatrix} + \begin{pmatrix} 2 & 2 & -1 & 2 \\ 2 & 1 & 0 & -2 \\ -2 & -2 & 2 & 2 \end{pmatrix} =$

☛ SEE ANSWER

Partitioning $m \times n$ matrix $A$ by columns, and $x$ and $y$ by individual elements, we find that

$$\left(\begin{array}{c|c|c|c} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,k-1} \\ \hline \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,k-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,k-1} \end{array}\right) :=$$

$$\left(\begin{array}{c} \chi_0 \\ \hline \chi_1 \\ \hline \vdots \\ \hline \chi_{m-1} \end{array}\right) \left(\begin{array}{c|c|c|c} \psi_0 & \psi_1 & \cdots & \psi_{n-1} \end{array}\right) + \left(\begin{array}{c|c|c|c} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,n-1} \\ \hline \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,n-1} \end{array}\right)$$

$$= \left(\begin{array}{c|c|c|c} \chi_0\psi_0+\alpha_{0,0} & \chi_0\psi_1+\alpha_{0,1} & \cdots & \chi_0\psi_{n-1}+\alpha_{0,n-1} \\ \hline \chi_1\psi_0+\alpha_{1,0} & \chi_1\psi_1+\alpha_{1,1} & \cdots & \chi_1\psi_{n-1}+\alpha_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \chi_{m-1}\psi_0+\alpha_{m-1,0} & \chi_{m-1}\psi_1+\alpha_{m-1,1} & \cdots & \chi_{m-1}\psi_{n-1}+\alpha_{m-1,n-1} \end{array}\right)$$

$$= \left(\begin{array}{c|c|c|c} \begin{pmatrix}\chi_0\\ \chi_1\\ \vdots\\ \chi_{m-1}\end{pmatrix}\psi_0+\begin{pmatrix}\alpha_{0,0}\\ \alpha_{1,0}\\ \vdots\\ \alpha_{m-1,0}\end{pmatrix} & \begin{pmatrix}\chi_0\\ \chi_1\\ \vdots\\ \chi_{m-1}\end{pmatrix}\psi_1+\begin{pmatrix}\alpha_{0,1}\\ \alpha_{1,1}\\ \vdots\\ \alpha_{m-1,1}\end{pmatrix} & \cdots & \begin{pmatrix}\chi_0\\ \chi_1\\ \vdots\\ \chi_{m-1}\end{pmatrix}\psi_{n-1}+\begin{pmatrix}\alpha_{0,n-1}\\ \alpha_{1,n-1}\\ \vdots\\ \alpha_{m-1,n-1}\end{pmatrix} \end{array}\right).$$

What this illustrates is that we could have partitioned $A$ by columns and $y$ by elements to find that

$$\begin{array}{rcl} \left(\begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{n-1}\end{array}\right) & := & x\left(\begin{array}{c|c|c|c} \psi_0 & \psi_1 & \cdots & \psi_{n-1}\end{array}\right)+\left(\begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{n-1}\end{array}\right) \\ & = & \left(\begin{array}{c|c|c|c} x\psi_0+a_0 & x\psi_1+a_1 & \cdots & x\psi_{n-1}+a_{n-1}\end{array}\right) \\ & = & \left(\begin{array}{c|c|c|c} \psi_0 x+a_0 & \psi_1 x+a_1 & \cdots & \psi_{n-1}x+a_{n-1}\end{array}\right). \end{array}$$

This discussion explains the JI loop ordering for computing $A := xy^T + A$:

```
for $j := 0, \ldots, n-1$
  for $i := 0, \ldots, m-1$
     $\alpha_{i,j} := \chi_i \psi_j + \alpha_{i,j}$      } $a_j := \psi_i x + a_j$
  end
end
```

What it also demonstrates is how the rank-1 operation can be implemented as a sequence of AXPY operations.

**Homework 1.4.3.2** In `Assignments/Week1/C/Ger_J_Axpy.c` complete the implementation of RANK-1 in terms of AXPY operations. You will test this with an implementation of matrix-matrix multiplication in a later homework.

☛ SEE ANSWER

Notice that there is also an IJ loop ordering that can be explained by partitioning $A$ by rows and $x$ by elements:

$$\left(\begin{array}{c} \widetilde{a}_0^T \\ \hline \widetilde{a}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T \end{array}\right) := \left(\begin{array}{c} \chi_0 \\ \hline \chi_1 \\ \hline \vdots \\ \hline \chi_{m-1} \end{array}\right) y^T + \left(\begin{array}{c} \widetilde{a}_0^T \\ \hline \widetilde{a}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T \end{array}\right) = \left(\begin{array}{c} \chi_0 y^T + \widetilde{a}_0^T \\ \hline \chi_1 y^T + \widetilde{a}_1^T \\ \hline \vdots \\ \hline \chi_{m-1} y^T + \widetilde{a}_{m-1}^T \end{array}\right)$$

leading to the algorithm

```
for $i := 0, \ldots, n-1$
  for $j := 0, \ldots, m-1$
      $\alpha_{i,j} := \chi_i \psi_j + \alpha_{i,j}$
  end
end
```

$$\left.\begin{array}{l} \textbf{for } j := 0, \ldots, m-1 \\ \quad \alpha_{i,j} := \chi_i \psi_j + \alpha_{i,j} \\ \textbf{end} \end{array}\right\} \widetilde{a}_i^T := \chi_i y^T + \widetilde{a}_i^T$$

and corresponding implementation.

**Homework 1.4.3.3** In `Assignments/Week1/C/Ger_I_Axpy.c` complete the implementation of RANK-1 in terms of AXPY operations (by rows). You will test this with an implementation of MMM in a later homework.

☛ SEE ANSWER

### 1.4.4 Matrix-matrix multiplication via rank-1 updates

**Homework 1.4.4.1**
Fill in the blanks:

$$\left(\begin{array}{r|r} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{array}\right)\left(\begin{array}{rrr} 1 & -2 & 0 \\ \hline 2 & -1 & 3 \end{array}\right) + \left(\begin{array}{r|r|r} 3 & 0 & -4 \\ -2 & 1 & -3 \\ 1 & -1 & -2 \end{array}\right) =$$

$$\left(\begin{array}{c|c|c} -1\times\ 1\ +\ 2\times 2\ +3 & -1\times -2\ +\ 2\times -1\ +0 & -1\times\ 0\ +\ 2\times 3\ \ -4 \\ \Box\times\Box+\Box\times\Box\ -2 & \Box\times\Box+\Box\times\Box\ +1 & \Box\times\Box+\Box\times\Box\ -3 \\ \Box\times\Box+\Box\times\Box\ +1 & \Box\times\Box+\Box\times\Box\ -1 & \Box\times\Box+\Box\times\Box\ -2 \end{array}\right)$$

$$= \left(\begin{array}{c|c|c} -1\times\Box & -1\times\Box & -1\times\Box \\ 0\times\Box & 0\times\Box & 0\times\Box \\ -2\times\Box & -2\times\Box & -2\times\Box \end{array}\right) + \left(\begin{array}{c|c|c} 2\times\Box & 2\times\Box & 2\times\Box \\ 1\times\Box & 1\times\Box & 1\times\Box \\ 3\times\Box & 3\times\Box & 3\times\Box \end{array}\right)$$

$$+ \left(\begin{array}{r|r|r} 3 & 0 & -4 \\ -2 & 1 & -3 \\ 1 & -1 & -2 \end{array}\right)$$

$$= \left(\begin{array}{c} \Box \\ \Box \\ \Box \end{array}\right)\left(\begin{array}{ccc} \Box & \Box & \Box \end{array}\right) + \left(\begin{array}{c} \Box \\ \Box \\ \Box \end{array}\right)\left(\begin{array}{ccc} \Box & \Box & \Box \end{array}\right) + \left(\begin{array}{r|r|r} 3 & 0 & -4 \\ -2 & 1 & -3 \\ 1 & -1 & -2 \end{array}\right)$$

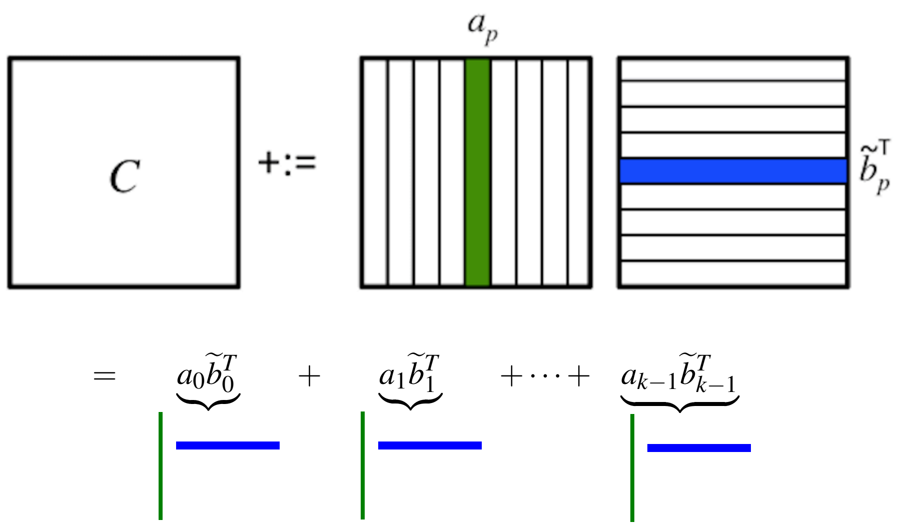
☛ SEE ANSWER

Next, let us partition $A$ by columns and $B$ by rows, so that

$$C \ := \ \left(\begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{k-1} \end{array}\right)\left(\begin{array}{c} \widetilde{b}_0^T \\ \hline \widetilde{b}_1^T \\ \hline \vdots \\ \hline \widetilde{b}_{k-1}^T \end{array}\right) + C$$

$$= \quad a_0\widetilde{b}_0^T + a_1\widetilde{b}_1^T + \cdots | a_{k-1}\widetilde{b}_{k-1}^T + C$$

A picture that captures this is given by



$a_p$
$C$
+:=
$\widetilde{b}_p^T$
$= \underbrace{a_0\widetilde{b}_0^T} + \underbrace{a_1\widetilde{b}_1^T} + \cdots + \underbrace{a_{k-1}\widetilde{b}_{k-1}^T}$

This illustrates how the PJI and PIJ algorithms can be viewed as a loop around matrix-vector multiplications:

```
for $p := 0, \ldots, k-1$
  for $j := 0, \ldots, n-1$                                  ⎫
    for $i := 0, \ldots, m-1$                 ⎫              ⎪
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$   ⎬ $c_j := \beta_{p,j}a_p + c_j$   ⎬ $C := a_p\widetilde{b}_p^T + C$
    end                                       ⎭              ⎪
  end                                                        ⎭
end
```

and

```
for $p := 0, \ldots, k-1$
  for $i := 0, \ldots, m-1$                                  ⎫
    for $j := 0, \ldots, n-1$                 ⎫              ⎪
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$   ⎬ $\widetilde{c}_i^T := \alpha_{i,p}\widetilde{b}_p^T + \widetilde{c}_i^T$   ⎬ $C := a_p\widetilde{b}_p^T + C$
    end                                       ⎭              ⎪
  end                                                        ⎭
end
```

**Homework 1.4.4.2** Complete the code in `Assignments/Week1/C/Gemm_P_Ger.c`. Test two versions:

```
make P_Ger_J_Axpy
make P_Ger_I_Axpy
```

View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Outer_P.mlx`. (Alternatively, use the script in ☛ `data/Plot_Outer_P_m.m`.)

☛ SEE ANSWER

The BLAS include the routine `dger` that computes

$$A := \alpha x y^T + A$$

If

- $m \times n$ matrix $A$ is stored in array `A` with its leading dimension stored in variable`ldA`,
- $m$ is stored in variable `m` and $n$ is stored in variable `n`,
- vector $x$ is stored in array `x` with its stride stored in variable `incx`,
- vector $y$ is stored in array `y` with its stride stored in variable `incy`, and
- $\alpha$ is stored in variable `alpha`,

the $A := \alpha x y^T + A$ translates, from C, into the call

```
dger_( &m, &n, &alpha, x, &incx, &beta, y, &incy, A, &ldA );
```

**Homework 1.4.4.3** Complete the code in `Assignments/Week1/C/Gemm_P_dger.c` that casts matrix-matrix multiplication in terms of the `dger` BLAS routine. Compile and execute with

```
make P_dger
```

View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Outer_P.mlx`. (Alternatively, use the script in ☛ `data/Plot_Outer_P_m.m`.)

☛ SEE ANSWER

The BLIS native call that is similar to the BLAS `dger` routine in this setting translates to

```
bli_dger( BLIS_NO_CONJUGATE, BLIS_NO_CONJUGATE, m, n, &alpha, x, incx,
          &beta, y, incy, A, 1, ldA );
```

Again, notice the two parameters after `A`. BLIS requires a row and column stride to be specified for matrices, thus generalizing beyond column-major order storage.

**Homework 1.4.4.4** Complete the code in `Assignments/Week1/C/Gemm_P_bli_dger.c` that casts matrix-matrix multiplication in terms of the `bli_dger` BLIS routine. Compile and execute with

```
make P_bli_dger
```

View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Outer_P.mlx`. (Alternatively, use the script in ☛ `data/Plot_Outer_P_m.m`.)

☛ SEE ANSWER

## 1.4.5 Matrix-matrix multiplication via row-times-matrix multiplications

**Homework 1.4.5.1**
Fill in the blanks:

$$\left(\begin{array}{rr} -1 & 2 \\ \hline 0 & 1 \\ \hline -2 & 3 \end{array}\right)\left(\begin{array}{rrr} 1 & -2 & 0 \\ 2 & -1 & 3 \end{array}\right) + \left(\begin{array}{rrr} 3 & 0 & -4 \\ \hline -2 & 1 & -3 \\ \hline 1 & -1 & -2 \end{array}\right) =$$

$$\left(\begin{array}{c|c|c} -1\times\ 1 + 2\times 2 + 3 & -1\times -2 + 2\times -1 + 0 & -1\times\ 0 + 2\times 3 - 4 \\ \Box\times\Box + \Box\times\Box - 2 & \Box\times\Box + \Box\times\Box + 1 & \Box\times\Box + \Box\times\Box - 3 \\ \Box\times\Box + \Box\times\Box + 1 & \Box\times\Box + \Box\times\Box - 1 & \Box\times\Box + \Box\times\Box - 2 \end{array}\right) =$$

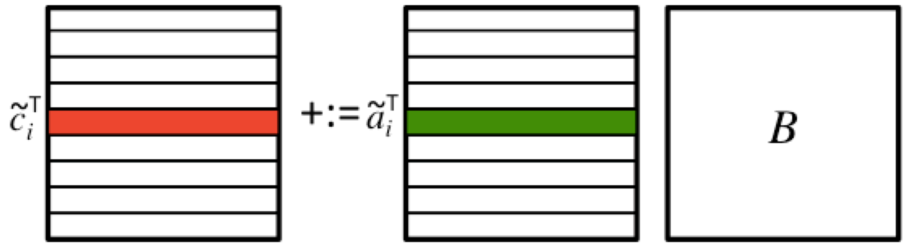
$$\left(\begin{array}{c} \left(\Box\ \Box\right)\left(\begin{array}{rrr} 1 & -2 & 0 \\ 2 & -1 & 3 \end{array}\right) + \left(\begin{array}{rrr} 3 & 0 & -4 \end{array}\right) \\ \hline \left(\Box\ \Box\right)\left(\begin{array}{rrr} 1 & -2 & 0 \\ 2 & -1 & 3 \end{array}\right) + \left(\begin{array}{rrr} -2 & 1 & -3 \end{array}\right) \\ \hline \left(\Box\ \Box\right)\left(\begin{array}{rrr} 1 & -2 & 0 \\ 2 & -1 & 3 \end{array}\right) + \left(\begin{array}{rrr} 1 & -1 & -2 \end{array}\right) \end{array}\right)$$

☛ SEE ANSWER

Finally, let us partition $C$ and $A$ by rows so that

$$\left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right) := \left(\begin{array}{c} \widetilde{a}_0^T \\ \hline \widetilde{a}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T \end{array}\right) B + \left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right) = \left(\begin{array}{c} \widetilde{a}_0^T B + \widetilde{c}_0^T \\ \hline \widetilde{a}_1^T B + \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T B + \widetilde{c}_{m-1}^T \end{array}\right)$$

A picture that captures this is given by



This illustrates how the IJP and IPJ algorithms can be viewed as a loop around the updating of a row of $C$ with the product of the corresponding row of $A$ times matrix $B$:

```
for $i := 0, \ldots, m-1$
  for $j := 0, \ldots, n-1$
      for $p := 0, \ldots, k-1$
          $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
      end                                    } $\widetilde{\gamma}_{i,j} := \widetilde{a}_i^T b_j + \widetilde{\gamma}_{i,j}$
  end                                        } $\widetilde{c}_i^T := \widetilde{a}_i^T B + \widetilde{c}_i^T$
end
```

and

```
for $i := 0, \ldots, m-1$
  for $p := 0, \ldots, k-1$
      for $j := 0, \ldots, n-1$
          $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
      end                                    } $\widetilde{c}_i^T := \alpha_{i,p}\widetilde{b}_p^T + \widetilde{c}_i^T$
  end                                        } $\widetilde{c}_i^T := \widetilde{a}_i^T B + \widetilde{c}_i^T$
end
```

The problem with implementing the above algorithms is that `Gemv_I_Dots` and `Gemv_J_Axpy` implement $y := Ax + y$ rather than $y^T := x^T A + y^T$. Obviously, you could create new routines for this new operation. We will instead look at how to use the BLAS and BLIS interfaces.

Recall that the BLAS include the routine `dgemv` that computes

$$y := \alpha A x + \beta y \quad \text{or} \quad y := \alpha A^T x + \beta y.$$

What we want is a routine that computes

$$y^T := x^T A + y^T.$$

What we remember from linear algebra is that if $A = B$ then $A^T = B^T$, that $(A+B)^T = A^T + B^T$, and that $(AB)^T = B^T A^T$. Thus, starting with the the equality

$$y^T = x^T A + y^T,$$

and transposing both sides, we get that

$$(y^T)^T = (x^T A + y^T)^T$$

which is equivalent to

$$y = (x^T A)^T + (y^T)^T$$

and finally

$$y = A^T x + y.$$

So, updating

$$y^T := x^T A + y^T$$

is equivalent to updating

$$y := A^T x + y.$$

It this all seems unfamiliar, you may want to look at ☛ Unit 3.2.5 of our MOOC titled "Linear Algebra: Foundations to Frontiers."

Now, if

- $m \times n$ matrix $A$ is stored in array `A` with its leading dimension stored in variable`ldA`,
- $m$ is stored in variable `m` and $n$ is stored in variable `n`,
- vector $x$ is stored in array `x` with its stride stored in variable `incx`,
- vector $y$ is stored in array `y` with its stride stored in variable `incy`, and
- $\alpha$ and $\beta$ are stored in variables `alpha` and `beta`, respectively,

then $y := \alpha A^T x + \beta y$ translates, from C, into the call

```
dgemv_( "Transpose", &m, &n, &alpha, A, &ldA, x, &incx, &beta, y, &incy );
```

**Homework 1.4.5.2** In directory `Assignments/Week1/C/` complete the code in file `Gemm_I_dgemv.c` that casts MMM in terms of the `dgemv` BLAS routine, but compute the result by rows. Compile and execute it with

make I_dgemv

View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Outer_I.mlx`. (Alternatively, use the script in ☛ `data/Plot_Outer_I_m.m`.)

☛ SEE ANSWER

The BLIS native call that is similar to the BLAS `dgemv` routine in this setting translates to

```
bli_dgemv( BLIS_TRANSPOSE, BLIS_NO_CONJUGATE, m, n, &alpha, A, ldA, 1,
                                              x, incx, &beta, y, incy );
```

Because of the two parameters after `A` that capture the stride between elements in a column (the row stride) and elements in rows (the column stride), one can alternatively swap these parameters:

```
bli_dgemv( BLIS_NO_TRANSPOSE, BLIS_NO_CONJUGATE, m, n, &alpha, A, ldA, 1,
                                                 x, incx, &beta, y, incy );
```

**Homework 1.4.5.3** In directory `Assignments/Week1/C/` complete the code in file `Gemm_I_bli_dgemv.c` that casts MMM in terms of the `bli_dgemv` BLIS routine. Compile and execute it with

make I_bli_dgemv

View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Outer_I.mlx`. (Alternatively, use the script in ☛ `data/Plot_Outer_I_m.m`.)

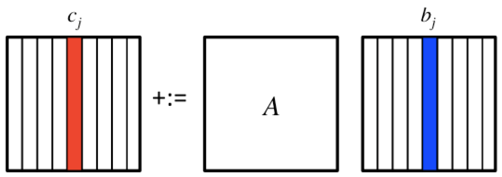
☛ SEE ANSWER

## 1.4.6 Discussion

The point of this section is that one can think of matrix-matrix multiplication as one loop around

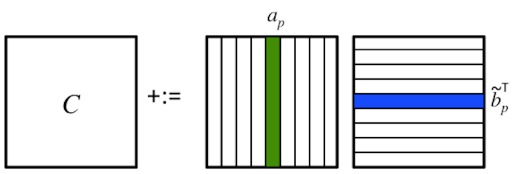
- multiple matrix-vector multiplications:

$$\left( c_0 \middle| \cdots \middle| c_{n-1} \right) = \left( Ab_0 + c_0 \middle| \cdots \middle| Ab_{n-1} + c_{n-1} \right)$$



- multiple rank-1 updates:

$$C = a_0 \widetilde{b}_0^T + a_1 \widetilde{b}_1^T + \cdots + a_{k-1} \widetilde{b}_{k-1}^T + C$$



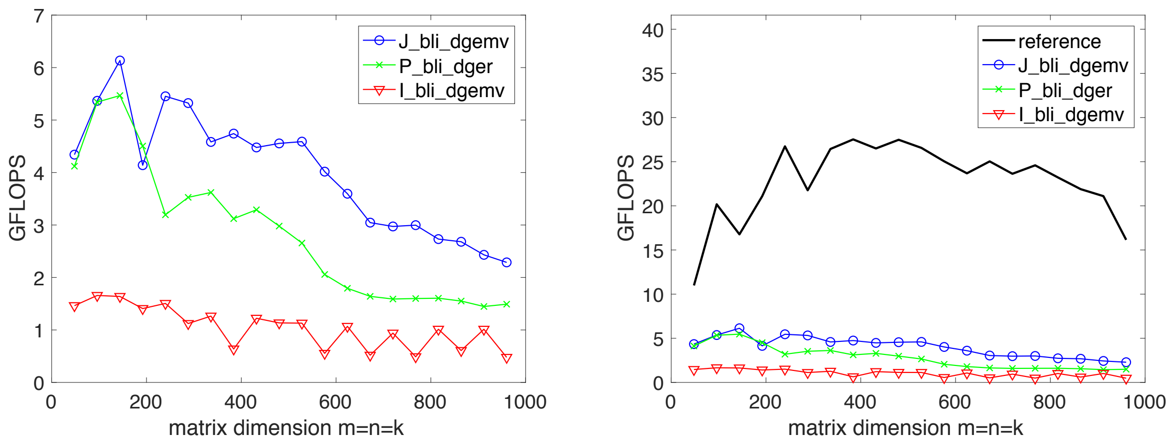
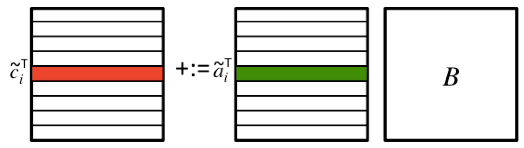


Figure 1.9: Performance for different choices of the outer loop, calling the BLIS typed interface, on Robert's laptop.

- multiple row times matrix multiplications:

$$\left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right) = \left(\begin{array}{c} \widetilde{a}_0^T B + \widetilde{c}_0^T \\ \hline \widetilde{a}_1^T B + \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T B + \widetilde{c}_{m-1}^T \end{array}\right)$$



So, for the outer loop there are three choices: $j$, $p$, or $i$. This may give the appearance that there are only three algorithms. However, the matrix-vector multiply and rank-1 update hide a double loop and the order of these two loops can be chosen, to bring us back to six choices for algorithms. Importantly, matrix-vector multiplication can be organized so that matrices are addressed by columns in the inner-most loop (the JI order for computing GEMV) as can the rank-1 update (the JI order for computing GER). For the implementation that picks the I loop as the outer loop, GEMV is utilized but with the transposed matrix. As a result, there is no way of casting the inner two loops in terms of operations with columns.

**Homework 1.4.6.1** In directory `Assignments/Week1/C/data` use the Live Script in ☛ `Plot_All_Outer.mlx` to compare and contrast the performance of many of the algorithms that use the I, J, and P loop as the outer-most loop. If you want to rerun all the experiments, you can do so by executing

```
make Plot_All_Outer
```

in directory `Assignments/Week1/C/`. What do you notice?

☛ SEE ANSWER

From the performance experiments reported in Figure 1.9, we have observed that accessing matrices by columns, so that the most frequently loaded data is contiguous in memory, yields better performance. Still, the observed performance is much worse than can be achieved.

We have seen many examples of blocked matrix-matrix multiplication already, where matrices were partitioned by rows, columns, or individual entries. This will be explored and exploited further next week.

## 1.5 Enrichment

### 1.5.1 Counting flops

Floating point multiplies or adds are examples of floating point operation (flops). What we have noticed is that for all of our computations (DOTS, AXPY, GEMV, GER, and GEMM) every floating point multiply is paired with a floating point add into a fused multiply-add (FMA).

Determining how many floating point operations are required for the different operations is relatively straight forward: If $x$ and $y$ are of size $n$, then

- $\gamma := x^T y + \gamma = \chi_0\psi_0 + \chi_1\psi_1 + \cdots + \chi_{n-1}\psi_{n-1} + \gamma$ requires $n$ FMAs and hence $2n$ flops.
- $y := \alpha x + y$ requires $n$ FMAs (one per pair of element, one from from $x$ and one from $y$) and hence $2n$ flops.

Similarly, it is pretty easy to establish that if $A$ is $m \times n$, then

- $y := Ax + y$ requires $mn$ FMAs and hence $2mn$ flops.
  $n$ AXPY operations each of size $m$ for $n \times 2m$ flops or $m$ ... operations each of size $n$ for $m \times 2n$ flops.
- $A := xy^T + A$ required $mn$ FMAs and hence $2mn$ flops.
  $n$ AXPY operations each of size $m$ for $n \times 2m$ flops or $m$ AXPY operations each of size $n$ for $m \times 2n$ flops.

Finally, if $C$ is $m \times n$, $A$ is $m \times k$, and $B$ is $k \times n$, then $C := AB + C$ requires $2mnk$ flops. We can estiblish this by recognizing that if $C$ is updated one column at a time, this takes $n$ GEMV operations each with a matrix of size $m \times k$, for a total of $n \times 2mk$. Alternativesly, if $C$ is updated with rank-1 updates, then we get $k \times 2mn$.

When we run experiments, we tend to execute with matrices that are $n \times n$, where $n$ ranges from small to large. Thus, the total operations required equal

$$\sum_{n=\mathtt{n_{first}}}^{\mathtt{n_{last}}} 2n^3 \text{ flops},$$

where $n_{\mathtt{first}}$ is the first problem size, $n_{\mathtt{last}}$ the last, and the summation is in increments of $n_{\mathtt{inc}}$.

If $\mathtt{n_{last}} = \mathtt{n_{inc}} \times N$ and $n_{\mathtt{inc}} = n_{\mathtt{first}}$, then we get

$$\sum_{n=n_{\mathtt{first}}}^{n_{\mathtt{last}}} 2n^3 = \sum_{i=1}^{N} 2(i \times n_{\mathtt{inc}})^3 = 2n_{\mathtt{inc}}^3 \sum_{i=1}^{N} i^3.$$

A standard trick is to recognize that

$$\sum_{i=1}^{N} i^3 \approx \int_0^N x^3 dx = \frac{1}{4} N^4.$$

So,

$$\sum_{n=n_{\texttt{first}}}^{n_{\texttt{last}}} 2n^3 \approx \frac{1}{2} n_{\texttt{inc}}^3 N^4 = \frac{1}{2} \frac{n_{\texttt{inc}}^4 N^4}{n_{\texttt{ninc}}} = \frac{1}{2 n_{\texttt{ninc}}} n^4.$$

The important thing is that every time we double $n_{\texttt{last}}$, we have to wait, approximately, sixteen times as long for our experiments to finish...

An interesting question is why we count flops rather than FMAs. By now, you have noticed we like to report the rate of computation in billions of floating point operations per second, GFLOPS. Now, if we counted FMAs rather than flops, the number that represents the rate of computation would be half cut in half. For marketing purposes, bigger is better and hence flops are reported! Seriously!

## 1.6 Wrapup

### 1.6.1 Additional exercises

### 1.6.2 Summary