



LAFF-On Programming for High Performance

**Robert A. van de Geijn
Devangi N. Parikh
Jianyu Huang
Margaret E. Myers**

LAFF-On

Programming for High Performance

Robert A. van de Geijn

Devangi N. Parikh

Jianyu Huang

Margaret E. Myers

Release Date

Wednesday 14th November, 2018

This is a work in progress

Copyright © 2017, 2018 by Robert A. van de Geijn, Devangi N. Parikh, and Margaret E. Myers.

10 9 8 7 6 5 4 3 2 1

All rights reserved. No part of this book may be reproduced, stored, or transmitted in any manner without the written permission of the publisher. For information, contact any of the authors.

No warranties, express or implied, are made by the publisher, authors, and their employers that the programs contained in this volume are free of error. They should not be relied on as the sole basis to solve a problem whose incorrect solution could result in injury to person or property. If the programs are employed in such a manner, it is at the user's own risk and the publisher, authors, and their employers disclaim all liability for such misuse.

Trademarked names may be used in this book without the inclusion of a trademark symbol. These names are used in an editorial context only; no infringement of trademark is intended.

Library of Congress Cataloging-in-Publication Data not yet available

Draft Edition, Wednesday 14th November, 2018

This “Draft Edition” allows this material to be used while we sort out through what mechanism we will publish the book.

Contents

1. Matrix-Matrix Multiplication Basics	1
1.1. Opening Remarks	1
1.1.1. Launch	1
1.1.2. Outline Week 1	5
1.1.3. What you will learn	6
1.2. Mapping Matrices to Memory	7
1.2.1. Column-major ordering	7
1.2.2. The leading dimension	11
1.2.3. A convention regarding the letter used for the loop index	11
1.2.4. Ordering the loops	11
1.3. Thinking in Terms of Vector-Vector Operations	14
1.3.1. The Basic Linear Algebra Subprograms (BLAS)	14
1.3.2. Notation	14
1.3.3. The dot product (inner product)	15
1.3.4. The IJP and JIP orderings	17
1.3.5. The AXPY operation	20
1.3.6. The IPJ and PIJ orderings	22
1.3.7. The JPI and PJI orderings	25
1.3.8. Discussion	28
1.4. Thinking in Terms of Matrix-Vector Operations	30
1.4.1. Matrix-vector multiplication via dot products or AXPY operations	30
1.4.2. Matrix-matrix multiplication via matrix-vector multiplications	33
1.4.3. Rank-1 update RANK-1	35
1.4.4. Matrix-matrix multiplication via rank-1 updates	38
1.4.5. Matrix-matrix multiplication via row-times-matrix multiplications	41
1.4.6. Discussion	44
1.5. Enrichment	46
1.5.1. Counting flops	46
1.6. Wrapup	47
1.6.1. Additional exercises	47

1.6.2. Summary	47
2. Start Your Engines!	49
2.1. Opener	49
2.1.1. Launch	49
2.1.2. Outline Week 2	50
2.1.3. What you will learn	51
2.2. Blocked Matrix-Matrix Multiplication	52
2.2.1. Element-wise matrix-matrix multiplication vs. blocked matrix-matrix multiplication	52
2.2.2. Haven't we seen this before?	53
2.3. Vector Registers and Vector Instructions	53
2.3.1. A simple model of memory and registers	53
2.3.2. Of vector registers and instructions, and instruction-level parallelism	62
2.4. Optimizing the Microkernel	68
2.4.1. Reuse of data in registers	68
2.4.2. More options	72
2.4.3. Amortizing data movement	73
2.4.4. Discussion	74
2.5. When Optimal Means Optimal	75
2.5.1. Reasoning about optimality	75
2.5.2. A simple model	75
2.5.3. Minimizing data movement	76
2.5.4. A nearly optimal algorithm	78
2.5.5. Discussion	80
2.6. Enrichment	80
2.7. Wrapup	80
3. Pushing the Limits	81
3.1. Opener	81
3.1.1. Launch	81
3.1.2. Outline Week 2	84
3.1.3. What you will learn	85
3.2. Leveraging the Caches	86
3.2.1. Adding cache memory into the mix	86
3.2.2. Streaming submatrices of C and B	89
3.2.3. Blocking for multiple caches	93
3.3. Contiguous Memory Access is Still Important	97
3.3.1. Stride matters	97
3.3.2. Packing	98
3.4. Further Tricks of the Trade	102
3.4.1. Avoiding repeated memory allocations	102
3.4.2. Different $m_R \times n_R$ choices	105
3.4.3. Alignment	105

3.4.4.	Prefetching	106
3.4.5.	Loop unrolling	106
3.4.6.	Different m_C , n_C and/or k_C choices	107
3.4.7.	Using in-line assembly code	107
3.5.	Enrichment	107
3.6.	Wrapup	107
3.6.1.	Additional exercises	107
4.	Multithreaded Parallelism	109
4.1.	Opener	109
4.1.1.	Launch	109
4.1.2.	Outline Week 2	110
4.1.3.	What you will learn	111
4.2.	OpenMP	112
4.2.1.	Of cores and threads	112
4.2.2.	Basics	112
4.2.3.	Hello World!	113
4.3.	Multithreading Matrix-Matrix Multiplication	115
4.3.1.	Parallelizing the second loop around the micro-kernel	115
4.3.2.	Parallelizing the first loop around the micro-kernel	117
4.3.3.	Parallelizing the third loop around the micro-kernel	118
4.4.	Parallelizing More	120
4.4.1.	Speedup, efficiency, etc.	120
4.4.2.	Ahmdahl's law	121
4.4.3.	Parallelizing the packing	122
4.5.	Enrichment	123
4.6.	Wrapup	123
4.	Message-Passing Interface Basics	117
4.1.	Opener	117
4.1.1.	Launch	117
4.1.2.	Outline Week 3	118
4.1.3.	What you will learn	119
4.2.	MPI Basics	120
4.2.1.	Programming distributed memory architectures	120
4.2.2.	MPI: Managing processes	120
4.2.3.	Sending and receiving	123
4.2.4.	Cost of sending/receiving a message	126
4.3.	Collective Communication	127
4.3.1.	MPI.Bcast	127
4.3.2.	Minimum spanning tree broadcast (MST Bcast)	129
4.3.3.	Can we do better yet?	130
4.3.4.	Various collective communications	132
4.4.	More Algorithms for Collective Communication	136

4.4.1.	Minimum Spanning Tree scatter and gather algorithms	136
4.4.2.	MST broadcast and reduce(-to-one)	139
4.4.3.	Bucket allgather and reduce-scatter	140
4.5.	A Calculus of Collective Communication	143
4.5.1.	The short vector case	143
4.5.2.	Leveraging ready-send for composed short-vector algorithms	146
4.5.3.	The long vector case	146
4.5.4.	Viewing the processes as a higher dimensional mesh	148
4.5.5.	From short vector to long vector	151
4.6.	Enrichment	152
4.7.	Wrapup	152

Preface

Acknowledgments

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

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

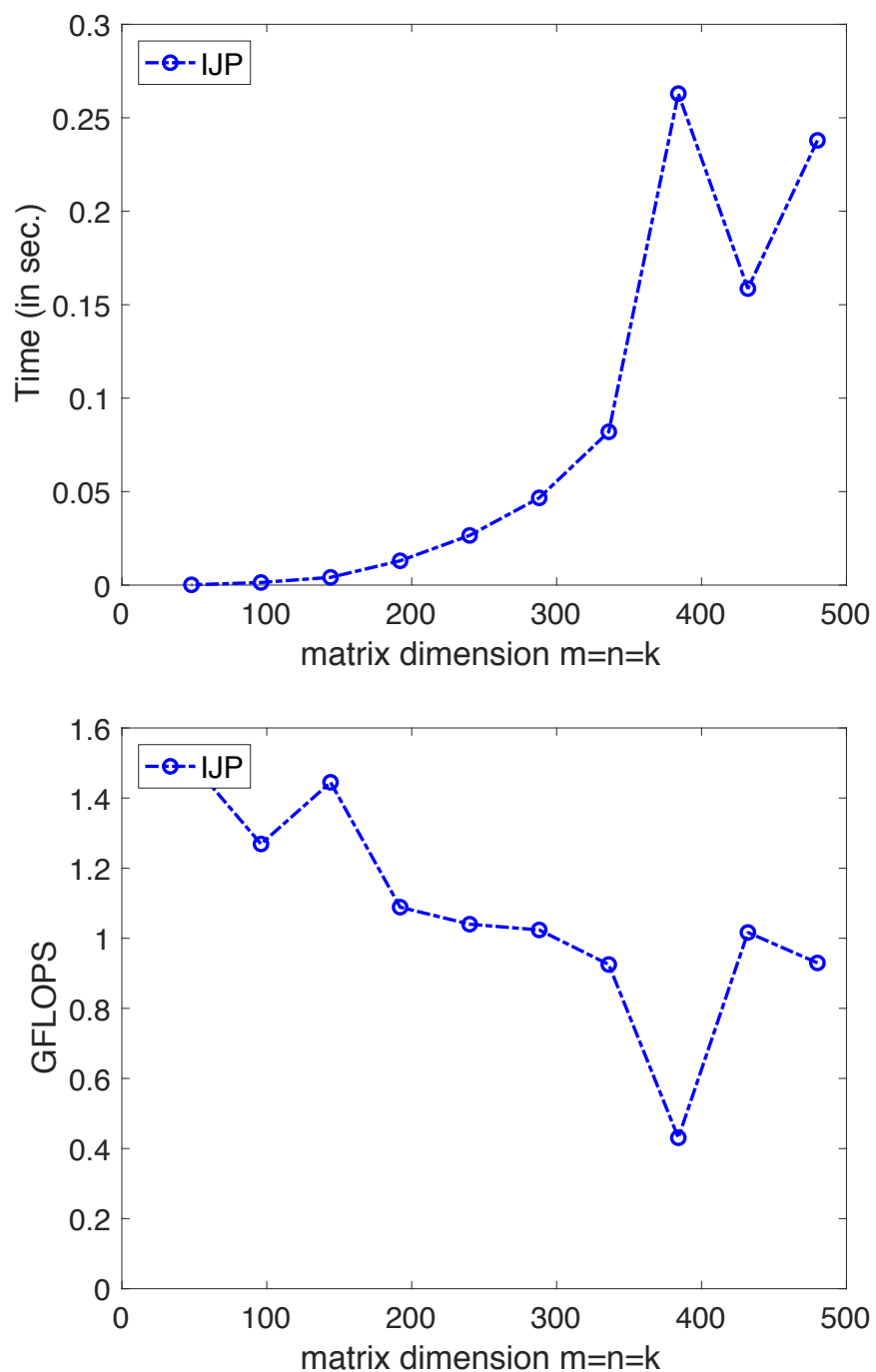


Figure 1.2: Performance comparison of a simple triple-nested loop and a high-performance implementation of the matrix-matrix multiplication $C := AB + C$. Top: Execution time. Bottom: Rate of computation, in billions of floating point operations per second (GFLOPS).

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.

1.1.2 Outline Week 1

1.1. Opening Remarks	1
1.1.1. Launch	1
1.1.2. Outline Week 1	5
1.1.3. What you will learn	6
1.2. Mapping Matrices to Memory	7
1.2.1. Column-major ordering	7
1.2.2. The leading dimension	11
1.2.3. A convention regarding the letter used for the loop index	11
1.2.4. Ordering the loops	11
1.3. Thinking in Terms of Vector-Vector Operations	14
1.3.1. The Basic Linear Algebra Subprograms (BLAS)	14
1.3.2. Notation	14
1.3.3. The dot product (inner product)	15
1.3.4. The IJP and JIP orderings	17
1.3.5. The AXPY operation	20
1.3.6. The IPJ and PIJ orderings	22
1.3.7. The JPI and PJI orderings	25
1.3.8. Discussion	28
1.4. Thinking in Terms of Matrix-Vector Operations	30
1.4.1. Matrix-vector multiplication via dot products or AXPY operations	30
1.4.2. Matrix-matrix multiplication via matrix-vector multiplications	33
1.4.3. Rank-1 update RANK-1	35
1.4.4. Matrix-matrix multiplication via rank-1 updates	38
1.4.5. Matrix-matrix multiplication via row-times-matrix multiplications	41
1.4.6. Discussion	44
1.5. Enrichment	46
1.5.1. Counting flops	46
1.6. Wrapup	47
1.6.1. Additional exercises	47
1.6.2. Summary	47

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 automatically 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

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

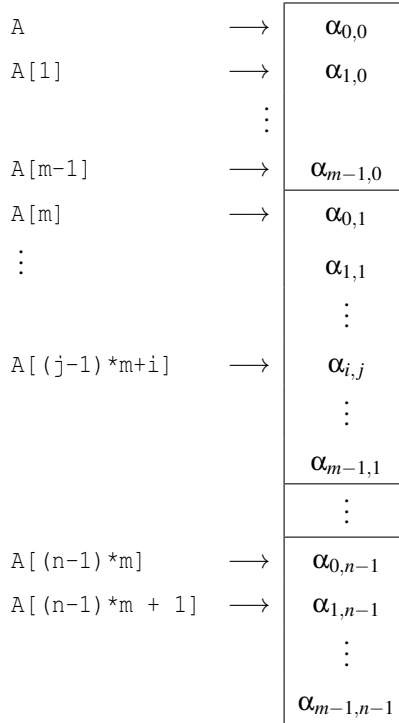
from the opener of this week. In memory, this may be stored in an array A by columns:

A	→	1
A[1]	→	-1
A[2]	→	-2
A[3]	→	-2
⋮		1
		2
		2
		3
		-1

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:

$$A \longrightarrow \begin{array}{|c|} \hline 1 \\ \hline -1 \\ \hline -2 \\ \hline -2 \\ \hline 1 \\ \hline 2 \\ \hline 2 \\ \hline \end{array}$$

and let A be the 2×3 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.

$$A \longrightarrow \begin{array}{|c|} \hline 1 \\ \hline -1 \\ \hline -2 \\ \hline -2 \\ \hline 1 \\ \hline 2 \\ \hline 2 \\ \hline \end{array}$$

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

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

[SEE ANSWER](#)

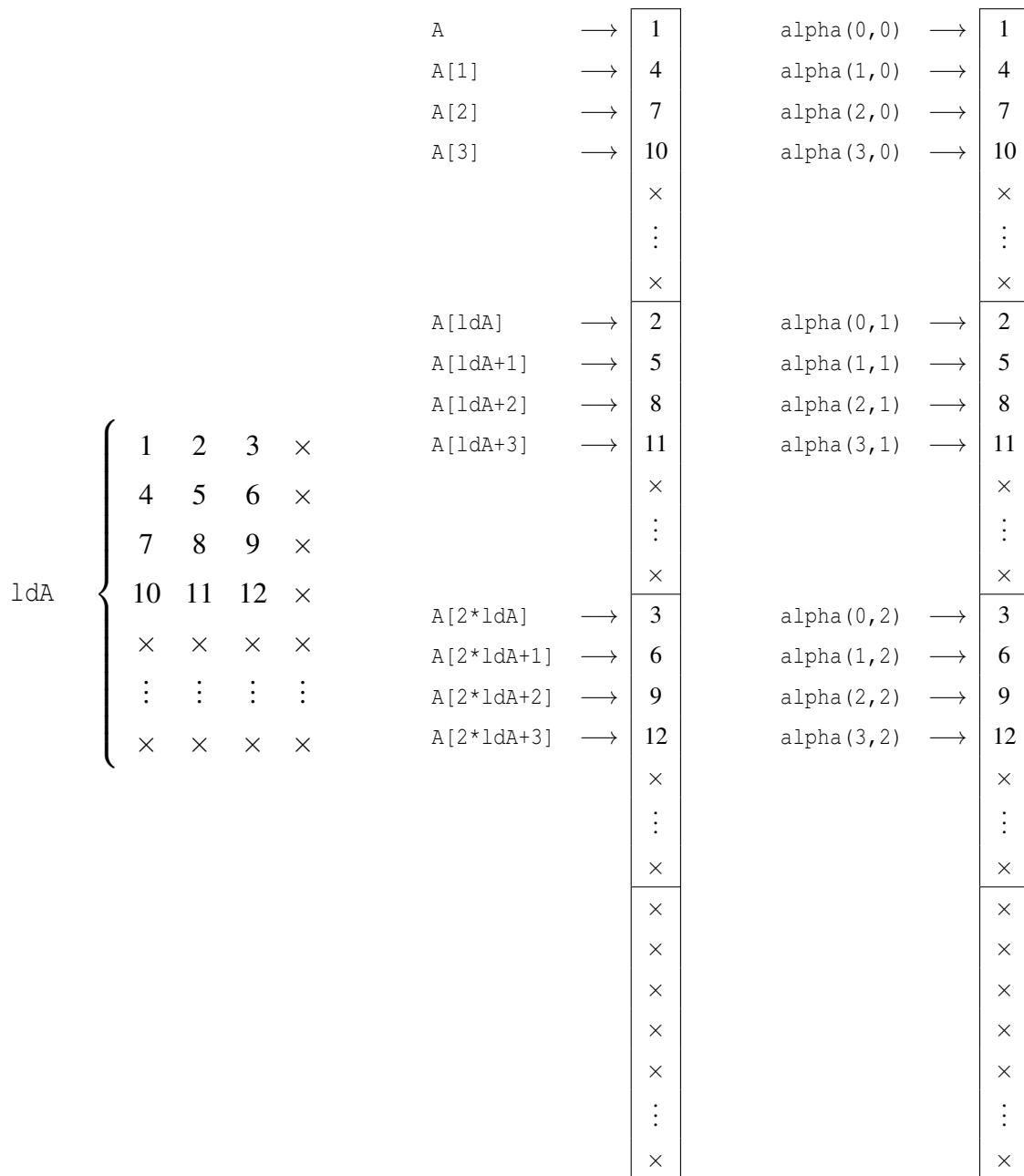


Figure 1.4: Addressing a matrix embedded in an array with ldA rows. At the left we illustrate a 4×3 submatrix of a $\text{ldA} \times 4$ matrix. In the middle, we illustrate how this is mapped into an linear array a. In the right, we show how defining the C macro `#define alpha(i,j) A[(j)*ldA + (i)]`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

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

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

1. The boxed submatrix starts at $A[\text{ }]$.
2. The row size of the boxed submatrix is $\text{ }.$
3. The column size of the boxed submatrix is $\text{ }.$
4. The leading dimension of the boxed submatrix is $\text{ }.$

 [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 $C := AB + C$.

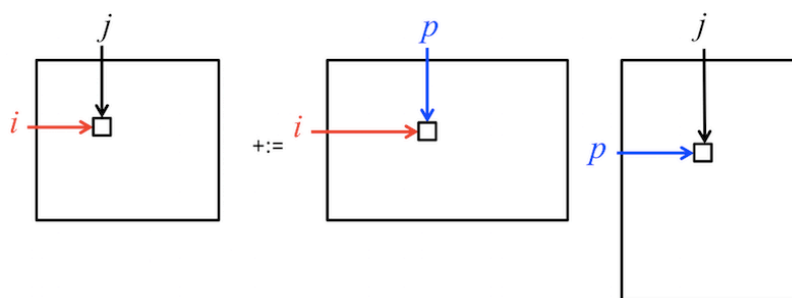


Figure 1.5: We embrace the convention that in loops that implement MMM, the variable i indexes the current row of matrix C (and hence the current row of A), the variable j indexes the current column of matrix C (and hence the current column of B), and variable p indexes the “other” dimension, namely 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  $i := 0, \dots, m - 1$ 
  for  $j := 0, \dots, n - 1$ 
    for  $p := 0, \dots, k - 1$ 
       $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ 
    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

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

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 result¹.

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](#)

¹In 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 ??.

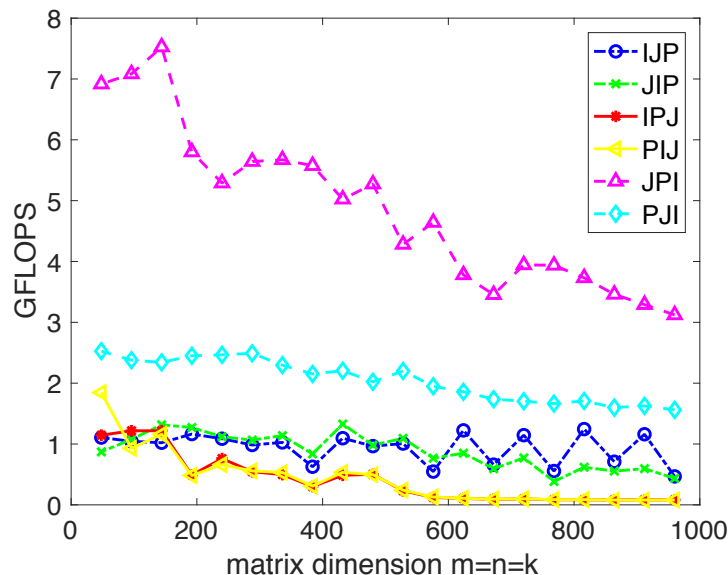


Figure 1.6: Performance comparison of all different orderings of the loops, on Robert's laptop.

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 implementations 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

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, \dots), lower case letters for vectors (a, b, c, \dots), and lower case Greek letters for scalars ($\alpha, \beta, \gamma, \dots$). 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

$$X = \left(x_0 \mid x_1 \mid \cdots \mid x_{n-1} \right)$$

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

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

so that \tilde{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 \sim 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 \tilde{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 or 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 χ . 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 γ 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 γ , 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 that we refer to as the BLIS Typed API². A Users' Guide for this 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

²BLIS is actually a framework for the rapid instantiation of BLAS-like functionality. It comes with four different interfaces to that functionality: The classic Fortran BLAS interface, the CBLAS interface for the C language (which is an interface that is rarely used), the BLIS Typed API with is reminiscent of the BLAS interface, but with added functionality and flexibility, and the BLIS object API, which which a Users' Guide can be found at <https://github.com/flame/blis/blob/master/docs/BLISObjectAPI.md>.

```
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, ..., m - 1
  for j := 0, ..., n - 1
    for p := 0, ..., 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:

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

We then notice that

$$\left(\begin{array}{c|c|c|c} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,n-1} \\ \hline \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,n-1} \end{array} \right)$$

$$\begin{aligned}
 &:= \begin{pmatrix} \tilde{a}_0^T \\ \tilde{a}_1^T \\ \vdots \\ \tilde{a}_{m-1}^T \end{pmatrix} \left(b_0 \mid b_1 \mid \cdots \mid b_{n-1} \right) + \begin{pmatrix} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,n-1} \\ \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,n-1} \end{pmatrix} \\
 &= \begin{pmatrix} \tilde{a}_0^T b_0 + \gamma_{0,0} & \tilde{a}_0^T b_1 + \gamma_{0,1} & \cdots & \tilde{a}_0^T b_{n-1} + \gamma_{0,n-1} \\ \tilde{a}_1^T b_0 + \gamma_{1,0} & \tilde{a}_1^T b_1 + \gamma_{1,1} & \cdots & \tilde{a}_1^T b_{n-1} + \gamma_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \tilde{a}_{m-1}^T b_0 + \gamma_{m-1,0} & \tilde{a}_{m-1}^T b_1 + \gamma_{m-1,1} & \cdots & \tilde{a}_{m-1}^T b_{n-1} + \gamma_{m-1,n-1} \end{pmatrix}.
 \end{aligned}$$


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 : $\gamma_{i,j} := \tilde{a}_i^T b_j + \gamma_{i,j}$, as illustrated by

```



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

$$\left. \begin{array}{l} \text{for } p := 0, \dots, k - 1 \\ \gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j} \\ \text{end} \end{array} \right\} \gamma_{i,j} := \tilde{a}_i^T b_j + \gamma_{i,j}$$

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](#)

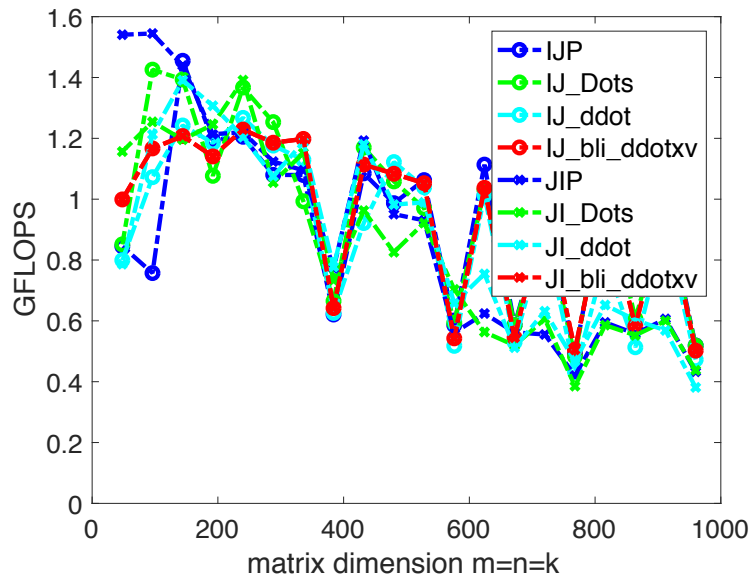


Figure 1.7: Performance of the various implementations of the IJP and JIP orderings.

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, α , 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

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

in file [Axy.c](#).

Homework 1.3.5.1 Complete the routine in `Axy.c`. You can find the partial implementation in `Axy.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 α 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_daxpyv`. 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  $i := 0, \dots, m-1$ 
  for  $p := 0, \dots, k-1$ 
    for  $j := 0, \dots, n-1$ 
       $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ 
    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:

$$C = \begin{pmatrix} \tilde{c}_0^T \\ \tilde{c}_1^T \\ \vdots \\ \tilde{c}_{m-1}^T \end{pmatrix}, \quad A = \begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,k-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,k-1} \\ \vdots & \vdots & & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,k-1} \end{pmatrix}, \quad \text{and} \quad B = \begin{pmatrix} \tilde{b}_0^T \\ \tilde{b}_1^T \\ \vdots \\ \tilde{b}_{k-1}^T \end{pmatrix}.$$

We then notice that

$$\begin{pmatrix} \tilde{c}_0^T \\ \tilde{c}_1^T \\ \vdots \\ \tilde{c}_{m-1}^T \end{pmatrix} := \begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,k-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,k-1} \\ \vdots & \vdots & & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,k-1} \end{pmatrix} \begin{pmatrix} \tilde{b}_0^T \\ \tilde{b}_1^T \\ \vdots \\ \tilde{b}_{k-1}^T \end{pmatrix} + \begin{pmatrix} \tilde{c}_0^T \\ \tilde{c}_1^T \\ \vdots \\ \tilde{c}_{m-1}^T \end{pmatrix}$$

$$= \begin{pmatrix} \tilde{c}_0^T + \alpha_{0,0}\hat{b}_0^T + \alpha_{0,1}\hat{b}_1^T + \cdots + \alpha_{0,k-1}\hat{b}_{k-1}^T \\ \tilde{c}_1^T + \alpha_{1,0}\hat{b}_0^T + \alpha_{1,1}\hat{b}_1^T + \cdots + \alpha_{1,k-1}\hat{b}_{k-1}^T \\ \vdots \\ \tilde{c}_{m-1}^T + \alpha_{m-1,0}\hat{b}_0^T + \alpha_{m-1,1}\hat{b}_1^T + \cdots + \alpha_{m-1,k-1}\hat{b}_{k-1}^T \end{pmatrix}.$$

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

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

$$\begin{pmatrix} \tilde{c}_0^T \\ \tilde{c}_1^T \\ \vdots \\ \tilde{c}_{m-1}^T \end{pmatrix} := \begin{pmatrix} \tilde{c}_0^T & + & \alpha_{0,0} \hat{b}_0^T \\ \tilde{c}_1^T & + & \alpha_{1,0} \hat{b}_0^T \\ \vdots & & \vdots \\ \tilde{c}_{m-1}^T & + & \alpha_{m-1,0} \hat{b}_0^T \end{pmatrix}.$$

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

$$\begin{pmatrix} \tilde{c}_0^T \\ \tilde{c}_1^T \\ \vdots \\ \tilde{c}_{m-1}^T \end{pmatrix} := \begin{pmatrix} \tilde{c}_0^T + \alpha_{0,0}\hat{b}_0^T + \alpha_{0,1}\hat{b}_1^T \\ \tilde{c}_1^T + \alpha_{1,0}\hat{b}_0^T + \alpha_{1,1}\hat{b}_1^T \\ \vdots \\ \tilde{c}_{m-1}^T + \alpha_{m-1,0}\hat{b}_0^T + \alpha_{m-1,1}\hat{b}_1^T \end{pmatrix},$$

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, ..., n-1
  for p := 0, ..., k-1
    for i := 0, ..., 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

$$\left(c_0 \mid c_1 \mid \cdots \mid c_{n-1} \right) := \left(a_0 \mid a_1 \mid \cdots \mid a_{k-1} \right) \begin{pmatrix} \beta_{0,0} & \beta_{0,1} & \cdots & \beta_{0,n-1} \\ \beta_{1,0} & \beta_{1,1} & \cdots & \beta_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \beta_{k-1,0} & \beta_{k-1,1} & \cdots & \beta_{k-1,n-1} \end{pmatrix} + \left(c_0 \mid c_1 \mid \cdots \mid c_{n-1} \right).$$

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  $j := 0, \dots, n-1$ 
  for  $p := 0, \dots, k-1$ 
    for  $i := 0, \dots, m-1$ 
       $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$ 
    end
  }  $c_j := \beta_{p,j}a_p + c_j$ 
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](#).)

[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](#).)

[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](#).)

[SEE ANSWER](#)

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

```

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

```

The outer loop in this algorithm fixes the column of A that is used 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.

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](#).)

[SEE ANSWER](#)

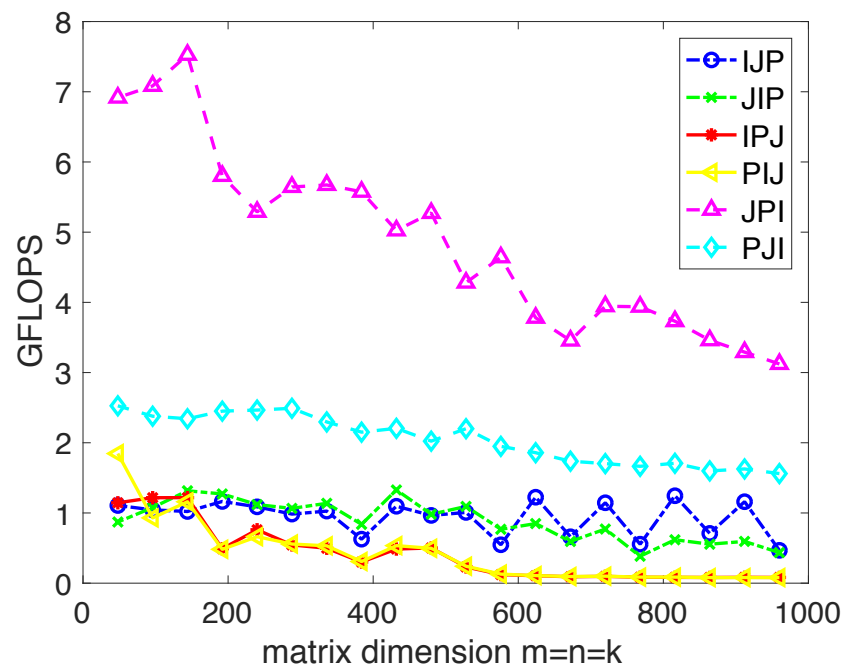


Figure 1.8: Here we repeat the graph from Figure 1.6, reporting the performance of all different orderings of the loops, on Robert's laptop.

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](#)

	Draw what the inner-most loop computes
<pre> for i := 0, ..., m - 1 for j := 0, ..., n - 1 for p := 0, ..., k - 1 $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end end end </pre>	
<pre> for i := 0, ..., m - 1 for p := 0, ..., k - 1 for j := 0, ..., n - 1 $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end end end </pre>	
<pre> for j := 0, ..., n - 1 for i := 0, ..., m - 1 for p := 0, ..., k - 1 $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end end end </pre>	
<pre> for j := 0, ..., n - 1 for p := 0, ..., k - 1 for i := 0, ..., m - 1 $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end end end </pre>	
<pre> for p := 0, ..., k - 1 for i := 0, ..., m - 1 for j := 0, ..., n - 1 $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end end end </pre>	
<pre> for p := 0, ..., k - 1 for j := 0, ..., n - 1 for i := 0, ..., m - 1 $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end end end </pre>	

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

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

[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 , ψ_i , is updated with the dot product of the corresponding row of A , \tilde{a}_i^T , with vector x . With our notation, we can describe this as

$$\begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{m-1} \end{pmatrix} := \begin{pmatrix} \tilde{a}_0^T \\ \tilde{a}_1^T \\ \vdots \\ \tilde{a}_{m-1}^T \end{pmatrix} x + \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{m-1} \end{pmatrix} = \begin{pmatrix} \tilde{a}_0^T x + \psi_0 \\ \tilde{a}_1^T x + \psi_1 \\ \vdots \\ \tilde{a}_{m-1}^T x + \psi_{m-1} \end{pmatrix}.$$

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

```

$$\left. \begin{array}{l} \text{for } i := 0, \dots, m-1 \\ \text{for } j := 0, \dots, n-1 \\ \psi_i := \alpha_{i,j} \chi_j + \psi_i \\ \text{end} \end{array} \right\} \psi_i := \tilde{a}_j^T x + \psi_i$$

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

```

#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

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

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

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

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

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

```

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

```

} $y := \chi_j a_j + y$

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 y

void Axy( int, double, double *, int, double *, int );

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++ )
    Axy( , , , , , );
}

```

in file `Gemv_J_Axy.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:

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

$$\begin{pmatrix} -1 \times 1 + 2 \times 2 + 3 & -1 \times -2 + 2 \times -1 + 0 & -1 \times 0 + 2 \times 3 & -4 \\ \boxed{} \times \boxed{} + \boxed{} \times \boxed{} - 2 & \boxed{} \times \boxed{} + \boxed{} \times \boxed{} + 1 & \boxed{} \times \boxed{} + \boxed{} \times \boxed{} - 3 & \\ \boxed{} \times \boxed{} + \boxed{} \times \boxed{} + 1 & \boxed{} \times \boxed{} + \boxed{} \times \boxed{} - 1 & \boxed{} \times \boxed{} + \boxed{} \times \boxed{} - 2 & \end{pmatrix} =$$

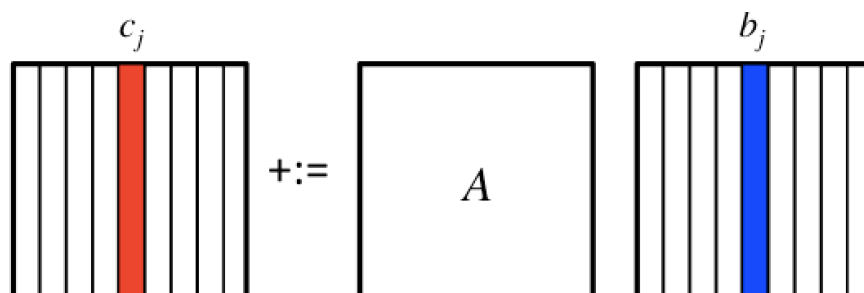
$$\left(\begin{pmatrix} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{pmatrix} \begin{pmatrix} \boxed{} \\ \boxed{} \end{pmatrix} + \begin{pmatrix} 3 \\ -2 \\ 1 \end{pmatrix} \right) \left| \begin{pmatrix} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{pmatrix} \begin{pmatrix} \boxed{} \\ \boxed{} \end{pmatrix} + \begin{pmatrix} 0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

[SEE ANSWER](#)

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

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

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

```

and

```

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

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

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

- α and β are stored in variables `alpha` and `beta`, respectively,

the $y := \alpha Ax + \beta y$ 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

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

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

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

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


```

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

```

$$\left. \begin{array}{l} \text{for } j := 0, \dots, n-1 \\ \text{for } i := 0, \dots, m-1 \\ \alpha_{i,j} := \chi_i \psi_j + \alpha_{i,j} \\ \text{end} \\ \text{end} \end{array} \right\} a_j := \psi_j x + a_j$$

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:

$$\begin{pmatrix} \tilde{a}_0^T \\ \tilde{a}_1^T \\ \vdots \\ \tilde{a}_{m-1}^T \end{pmatrix} := \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{m-1} \end{pmatrix} y^T + \begin{pmatrix} \tilde{a}_0^T \\ \tilde{a}_1^T \\ \vdots \\ \tilde{a}_{m-1}^T \end{pmatrix} = \begin{pmatrix} \chi_0 y^T + \tilde{a}_0^T \\ \chi_1 y^T + \tilde{a}_1^T \\ \vdots \\ \chi_{m-1} y^T + \tilde{a}_{m-1}^T \end{pmatrix}$$

leading to the algorithm

```

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

```

$$\left. \begin{array}{l} \text{for } i := 0, \dots, n-1 \\ \text{for } j := 0, \dots, m-1 \\ \alpha_{i,j} := \chi_i \psi_j + \alpha_{i,j} \\ \text{end} \\ \text{end} \end{array} \right\} \tilde{a}_i^T := \chi_i y^T + \tilde{a}_i^T$$

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:

$$\begin{aligned}
 & \left(\begin{array}{c|c} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{array} \right) \left(\begin{array}{ccc} 1 & -2 & 0 \\ 2 & -1 & 3 \end{array} \right) + \left(\begin{array}{c|c|c} 3 & 0 & -4 \\ -2 & 1 & -3 \\ 1 & -1 & -2 \end{array} \right) = \\
 & \left(\begin{array}{c|c|c} -1 \times 1 + 2 \times 2 + 3 & -1 \times -2 + 2 \times -1 + 0 & -1 \times 0 + 2 \times 3 - 4 \\ \hline \square \times \square + \square \times \square - 2 & \square \times \square + \square \times \square + 1 & \square \times \square + \square \times \square - 3 \\ \hline \square \times \square + \square \times \square + 1 & \square \times \square + \square \times \square - 1 & \square \times \square + \square \times \square - 2 \end{array} \right) \\
 & = \left(\begin{array}{c|c|c} -1 \times \square & -1 \times \square & -1 \times \square \\ 0 \times \square & 0 \times \square & 0 \times \square \\ -2 \times \square & -2 \times \square & -2 \times \square \end{array} \right) + \left(\begin{array}{c|c|c} 2 \times \square & 2 \times \square & 2 \times \square \\ 1 \times \square & 1 \times \square & 1 \times \square \\ 3 \times \square & 3 \times \square & 3 \times \square \end{array} \right) \\
 & \quad + \left(\begin{array}{c|c|c} 3 & 0 & -4 \\ -2 & 1 & -3 \\ 1 & -1 & -2 \end{array} \right) \\
 & = \left(\begin{array}{c} \square \\ \square \\ \square \end{array} \right) \left(\begin{array}{ccc} \square & \square & \square \end{array} \right) + \left(\begin{array}{c} \square \\ \square \\ \square \end{array} \right) \left(\begin{array}{ccc} \square & \square & \square \end{array} \right) + \left(\begin{array}{c|c|c} 3 & 0 & -4 \\ -2 & 1 & -3 \\ 1 & -1 & -2 \end{array} \right)
 \end{aligned}$$

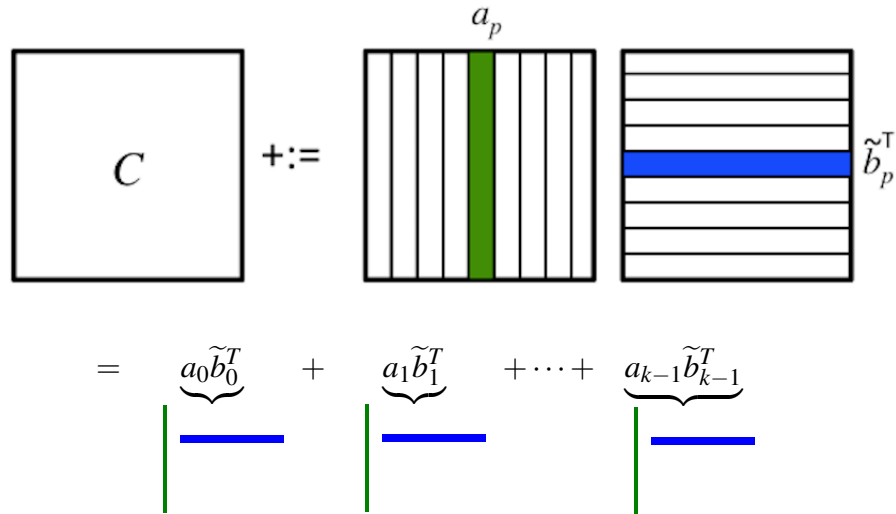
[SEE ANSWER](#)

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

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

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

A picture that captures this is given by



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

```

for  $p := 0, \dots, k-1$ 
  for  $j := 0, \dots, n-1$ 
    for  $i := 0, \dots, m-1$ 
       $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ 
    end
     $c_j := \beta_{p,j} a_p + c_j$ 
  end
   $C := a_p \tilde{b}_p^T + C$ 
end

```

and

```

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

$$A := \alpha xy^T + A$$

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
- α is stored in variable `alpha`,

the $A := \alpha xy^T + A$ 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:

$$\begin{pmatrix} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{pmatrix} \begin{pmatrix} 1 & -2 & 0 \\ 2 & -1 & 3 \end{pmatrix} + \begin{pmatrix} 3 & 0 & -4 \\ -2 & 1 & -3 \\ 1 & -1 & -2 \end{pmatrix} =$$

$$\begin{pmatrix} -1 \times 1 + 2 \times 2 + 3 & -1 \times -2 + 2 \times -1 + 0 & -1 \times 0 + 2 \times 3 - 4 \\ \square \times \square + \square \times \square - 2 & \square \times \square + \square \times \square + 1 & \square \times \square + \square \times \square - 3 \\ \square \times \square + \square \times \square + 1 & \square \times \square + \square \times \square - 1 & \square \times \square + \square \times \square - 2 \end{pmatrix} =$$

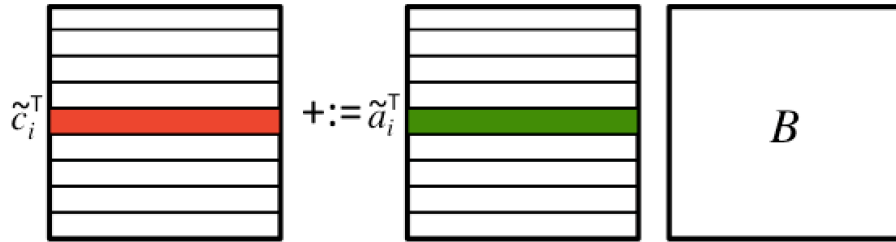
$$\begin{pmatrix} (\square \ \square) \begin{pmatrix} 1 & -2 & 0 \\ 2 & -1 & 3 \end{pmatrix} + (3 \ 0 \ -4) \\ (\square \ \square) \begin{pmatrix} 1 & -2 & 0 \\ 2 & -1 & 3 \end{pmatrix} + (-2 \ 1 \ -3) \\ (\square \ \square) \begin{pmatrix} 1 & -2 & 0 \\ 2 & -1 & 3 \end{pmatrix} + (1 \ -1 \ -2) \end{pmatrix}$$

[SEE ANSWER](#)

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

$$\begin{pmatrix} \tilde{c}_0^T \\ \tilde{c}_1^T \\ \vdots \\ \tilde{c}_{m-1}^T \end{pmatrix} := \begin{pmatrix} \tilde{a}_0^T \\ \tilde{a}_1^T \\ \vdots \\ \tilde{a}_{m-1}^T \end{pmatrix} B + \begin{pmatrix} \tilde{c}_0^T \\ \tilde{c}_1^T \\ \vdots \\ \tilde{c}_{m-1}^T \end{pmatrix} = \begin{pmatrix} \tilde{a}_0^T B + \tilde{c}_0^T \\ \tilde{a}_1^T B + \tilde{c}_1^T \\ \vdots \\ \tilde{a}_{m-1}^T B + \tilde{c}_{m-1}^T \end{pmatrix}$$

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

```

and

```

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

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

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 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

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

If 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
- α and β 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
```

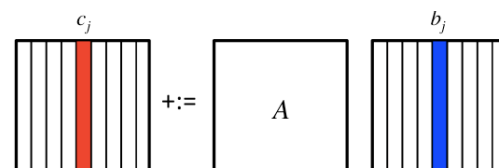
View the resulting performance by making the necessary changes to the Live Script in [Plot_Outer_I.mlx](#). (Alternatively, use the script in [data/Plot_Outer_I.m.m.](#))

[SEE ANSWER](#)

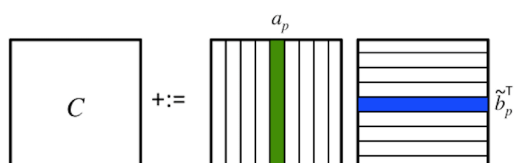
1.4.6 Discussion

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

- multiple matrix-vector multiplications:

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


- multiple rank-1 updates:

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


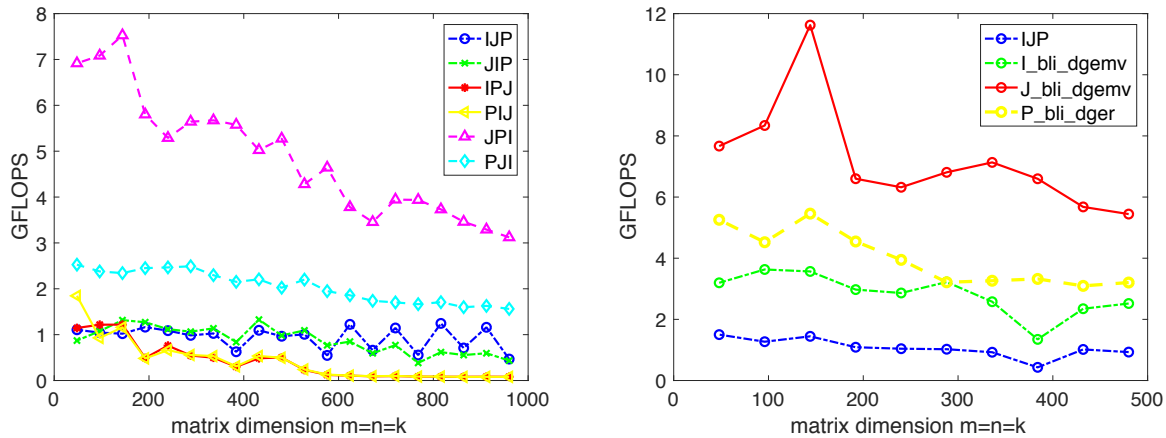
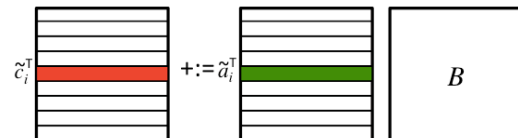


Figure 1.9: Left: Performance of simple implementations of the different loop orderings. Right: Performance for different choices of the outer loop, calling the BLIS typed interface. All experiments were performed on Robert's laptop. Notice that the scale along the y-axis is not consistent between the two graphs.

- multiple row times matrix multiplications:

$$\begin{pmatrix} \tilde{c}_0^T \\ \tilde{c}_1^T \\ \vdots \\ \tilde{c}_{m-1}^T \end{pmatrix} = \begin{pmatrix} \tilde{a}_0^T B + \tilde{c}_0^T \\ \tilde{a}_1^T B + \tilde{c}_1^T \\ \vdots \\ \tilde{a}_{m-1}^T B + \tilde{c}_{m-1}^T \end{pmatrix}$$



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 now partitioning the matrices, what we call “slicing and dicing”, enhances our insight into how different algorithms access data. This will be explored and exploited further in future weeks.

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 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 $m \dots$ 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 establish 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 $n \times 2mk$. Alternatively, 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

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

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

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

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

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,

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

The important thing is that every time we double n_{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

Week 2

Start Your Engines!

2.1 Opener

2.1.1 Launch

Homework 2.1.1.1 At the end of the last Week, you used the Live Script in [./Assignments/Week1/C/data/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. The resulting graph (for experiments on Robert's laptop) is given again in Figure ?? . You may feel pretty happy, given how much the performance has improved. In that Live Script, change the (0) in

```
if ( 0 )  
    plot( data(:,1), data(:,3), 'MarkerSize', 8, 'LineWidth', 1, ...  
          'LineStyle', '-', 'DisplayName', 'Ref', 'Color', plot_colors( 1,: ) );  
end
```

to (1) and regenerate the graph. What do you observe?

[SEE ANSWER](#)

2.1.2 Outline Week 2

2.1. Opener	49
2.1.1. Launch	49
2.1.2. Outline Week 2	50
2.1.3. What you will learn	51
2.2. Blocked Matrix-Matrix Multiplication	52
2.2.1. Element-wise matrix-matrix multiplication vs. blocked matrix-matrix multiplication	52
2.2.2. Haven't we seen this before?	53
2.3. Vector Registers and Vector Instructions	53
2.3.1. A simple model of memory and registers	53
2.3.2. Of vector registers and instructions, and instruction-level parallelism	62
2.4. Optimizing the Microkernel	68
2.4.1. Reuse of data in registers	68
2.4.2. More options	72
2.4.3. Amortizing data movement	73
2.4.4. Discussion	74
2.5. When Optimal Means Optimal	75
2.5.1. Reasoning about optimality	75
2.5.2. A simple model	75
2.5.3. Minimizing data movement	76
2.5.4. A nearly optimal algorithm	78
2.5.5. Discussion	80
2.6. Enrichment	80
2.7. Wrapup	80

2.1.3 What you will learn

2.2 Blocked Matrix-Matrix Multiplication

2.2.1 Element-wise matrix-matrix multiplication vs. blocked matrix-matrix multiplication

If the material in this section makes you scratch your head, you may want to go through the materials in Week 5 of our other MOOC: [Linear Algebra: Foundations to Frontiers](#).

Key to understanding how we are going to optimize MMM is to understand blocked MMM (the multiplication of matrices that have been partitioned into submatrices).

Consider $C := AB + C$. In terms of the elements of the matrices, this means that each $\gamma_{i,j}$ is computed by

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

Now, partition the matrices into submatrices:

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

and

$$B = \left(\begin{array}{c|c|c|c} B_{0,0} & B_{0,1} & \cdots & B_{0,N-1} \\ \hline B_{1,0} & B_{1,1} & \cdots & B_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline B_{K-1,0} & B_{K-1,1} & \cdots & B_{K-1,N-1} \end{array} \right),$$

where

- $C_{i,j}$ is $m_i \times n_j$,
- $A_{i,p}$ is $m_i \times k_p$, and
- $B_{p,j}$ is $k_p \times n_j$

with

$\sum_{i=0}^{M-1} m_i = m$, $\sum_{j=0}^{N-1} n_j = m$, and $\sum_{p=0}^{K-1} k_p = m$. Then

$$C_{i,j} := \sum_{p=0}^{K-1} A_{i,p} B_{p,j} + C_{i,j}.$$

In other words, provided the partitioning of the matrices is “conformal,” MMM with partitioning matrices proceeds exactly as does MMM with the elements of the matrices *except* that the individual multiplications with the submatrices do not necessarily commute.

We illustrate how blocking C into $m_b \times n_b$ submatrices, A into $m_b \times n_b$ submatrices, and B into $k_b \times n_b$ submatrices in Figure 2.1. An implementation of one such algorithm is given in Figure 2.2.

Homework 2.2.1.2 In directory `Assignments/Week2/C/` examine and time the code in file `Gemm_IJP_mbxnbxkb.c` by executing `make IJP_mbxnbxkb`. View its performance with `data/Plot_Blocked_MMM.mlx`.

👉 [SEE ANSWER](#)

2.2.2 Haven’t we seen this before?

We already employed various cases of blocked matrix-matrix multiplication in Week 1.

Homework 2.2.2.3 In Section 1.3, we already used the blocking of matrices to cast matrix-matrix multiplications in terms of dot products and AXPY operations. This is captured in the figure on Page 55. For each of partitionings in the right column, indicate how m_b , n_b , and k_b are chosen.

👉 [SEE ANSWER](#)

Homework 2.2.2.4 In Section 1.4, we already used the blocking of matrices to cast matrix-matrix multiplications in terms of matrix-vector multiplication and rank-1 updates. This is captured in the figure on Page 56. For each of partitionings in the right column, indicate how m_b , n_b , and k_b are chosen.

👉 [SEE ANSWER](#)

2.3 Vector Registers and Vector Instructions

2.3.1 A simple model of memory and registers

For computation to happen, input data must be brought from memory into registers and, sooner or later, output data must be written back to memory.

For now let us make the following assumptions:

- Our processor has only one core.
- That core only has two levels of memory: registers and main memory.
- Moving data between main memory and registers takes time β per double.

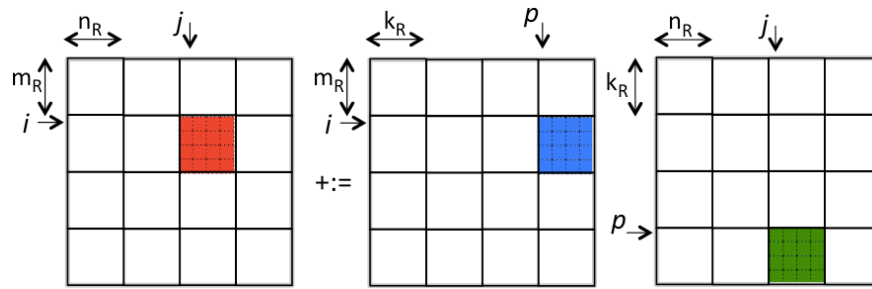


Figure 2.1: An illustration of a blocked algorithm where $m_R = n_R = k_R = 4$.

```

16 void MyGemm( int m, int n, int k, double *A, int ldA,
17             double *B, int ldB, double *C, int ldC )
18 {
19     for ( int j=0; j<n; j+=NB ){
20         int jb = min( n-j, NB ); /* Size for "finge" block */
21         for ( int i=0; i<m; i+=MB ){
22             int ib = min( m-i, MB ); /* Size for "finge" block */
23             for ( int p=0; p<k; p+=KB ){
24                 int pb = min( k-p, KB ); /* Size for "finge" block */
25                 Gemm_mbxnbxkb( ib, jb, pb, &alpha( i,p ), ldA, &beta( p,j ), ldB,
26                             &gamma( i,j ), ldC );
27             }
28         }
29     }
30 }
31
32 void Gemm_mbxnbxkb( int m, int n, int k, double *A, int ldA,
33                   double *B, int ldB, double *C, int ldC )
34 {
35     for ( int p=0; p<k; p++ )
36         for ( int j=0; j<n; j++ )
37             for ( int i=0; i<m; i++ )
38                 gamma( i,j ) += alpha( i,p ) * beta( p,j );
39 }

```

Assignments/Week2/C/Gemm_JIP_mbxnbxkb.c

Figure 2.2: Simple blocked algorithm that calls a kernel that updates a $m_b \times n_b$ submatrix of C with the result of multiplying an $m_b \times k_b$ submatrix of A times a $k_b \times n_b$ submatrix of B .

- The registers can hold 64 doubles.

	Blocked matrix-matrix multiplication
<pre> for $i := 0, \dots, m-1$ for $j := 0, \dots, n-1$ for $p := 0, \dots, k-1$ $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end $\gamma_{i,j} := \tilde{a}_i^T b_j + \gamma_{i,j}$ end end </pre>	$m_b = \boxed{1}, n_b = \boxed{1}, \text{ and } k_b = \boxed{k} :$ $\left(\begin{array}{c c c} \gamma_{0,0} & \cdots & \gamma_{0,n-1} \\ \hline \vdots & & \vdots \\ \hline \gamma_{m-1,0} & \cdots & \gamma_{m-1,n-1} \end{array} \right) + := \left(\begin{array}{c} \tilde{a}_0^T \\ \vdots \\ \tilde{a}_{m-1}^T \end{array} \right) \left(b_0 \mid \cdots \mid b_{n-1} \right)$
<pre> for $i := 0, \dots, m-1$ for $p := 0, \dots, k-1$ for $j := 0, \dots, n-1$ $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end $\tilde{c}_i^T := \alpha_{i,p} \tilde{b}_p^T + \tilde{c}_i^T$ end end </pre>	$m_b = \boxed{}, n_b = \boxed{}, \text{ and } k_b = \boxed{} :$ $\left(\begin{array}{c} \tilde{c}_0^T \\ \vdots \\ \tilde{c}_{m-1}^T \end{array} \right) + := \left(\begin{array}{c c c} \alpha_{0,0} & \cdots & \alpha_{0,k-1} \\ \hline \vdots & & \vdots \\ \hline \alpha_{m-1,0} & \cdots & \alpha_{m-1,k-1} \end{array} \right) \left(\begin{array}{c} \tilde{b}_0^T \\ \vdots \\ \tilde{b}_{k-1}^T \end{array} \right)$
<pre> for $j := 0, \dots, n-1$ for $i := 0, \dots, m-1$ for $p := 0, \dots, k-1$ $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end $\gamma_{i,j} := \tilde{a}_i^T b_j + \gamma_{i,j}$ end end </pre>	$m_b = \boxed{}, n_b = \boxed{}, \text{ and } k_b = \boxed{} :$ $\left(\begin{array}{c c c} \gamma_{0,0} & \cdots & \gamma_{0,n-1} \\ \hline \vdots & & \vdots \\ \hline \gamma_{m-1,0} & \cdots & \gamma_{m-1,n-1} \end{array} \right) + := \left(\begin{array}{c} \tilde{a}_0^T \\ \vdots \\ \tilde{a}_{m-1}^T \end{array} \right) \left(b_0 \mid \cdots \mid b_{n-1} \right)$
<pre> for $j := 0, \dots, n-1$ for $p := 0, \dots, k-1$ for $i := 0, \dots, m-1$ $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end $c_j := \beta_{p,j} a_p + c_j$ end end </pre>	$m_b = \boxed{}, n_b = \boxed{}, \text{ and } k_b = \boxed{} :$ $\left(c_0 \mid \cdots \mid c_{n-1} \right) + := \left(a_0 \mid \cdots \mid a_{k-1} \right) \left(\begin{array}{c c c} \beta_{0,0} & \cdots & \beta_{0,n-1} \\ \hline \vdots & & \vdots \\ \hline \beta_{k-1,0} & \cdots & \beta_{k-1,n-1} \end{array} \right)$
<pre> for $p := 0, \dots, k-1$ for $i := 0, \dots, m-1$ for $j := 0, \dots, n-1$ $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end $\tilde{c}_i^T := \alpha_{i,p} \tilde{b}_p^T + \tilde{c}_i^T$ end end </pre>	$m_b = \boxed{}, n_b = \boxed{}, \text{ and } k_b = \boxed{} :$ $\left(\begin{array}{c} \tilde{c}_0^T \\ \vdots \\ \tilde{c}_{m-1}^T \end{array} \right) + := \left(\begin{array}{c c c} \alpha_{0,0} & \cdots & \alpha_{0,k-1} \\ \hline \vdots & & \vdots \\ \hline \alpha_{m-1,0} & \cdots & \alpha_{m-1,k-1} \end{array} \right) \left(\begin{array}{c} \tilde{b}_0^T \\ \vdots \\ \tilde{b}_{k-1}^T \end{array} \right)$
<pre> for $p := 0, \dots, k-1$ for $j := 0, \dots, n-1$ for $i := 0, \dots, m-1$ $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end $c_j := \beta_{p,j} a_p + c_j$ end end </pre>	$m_b = \boxed{}, n_b = \boxed{}, \text{ and } k_b = \boxed{} :$ $\left(c_0 \mid \cdots \mid c_{n-1} \right) + := \left(a_0 \mid \cdots \mid a_{k-1} \right) \left(\begin{array}{c c c} \beta_{0,0} & \cdots & \beta_{0,n-1} \\ \hline \vdots & & \vdots \\ \hline \beta_{k-1,0} & \cdots & \beta_{k-1,n-1} \end{array} \right)$

	Blocked matrix-matrix multiplication
<pre> for i := 0, ..., m - 1 for j := 0, ..., n - 1 for p := 0, ..., k - 1 $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end end end </pre>	$m_b = \square, n_b = \square, \text{ and } k_b = \square:$ $\begin{pmatrix} \tilde{c}_0^T \\ \vdots \\ \tilde{c}_{m-1}^T \end{pmatrix} +:= \begin{pmatrix} \tilde{a}_0^T \\ \vdots \\ \tilde{a}_{m-1}^T \end{pmatrix} B$
<pre> for i := 0, ..., m - 1 for p := 0, ..., k - 1 for j := 0, ..., n - 1 $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end end end </pre>	$m_b = \square, n_b = \square, \text{ and } k_b = \square:$ $\begin{pmatrix} \tilde{c}_0^T \\ \vdots \\ \tilde{c}_{m-1}^T \end{pmatrix} +:= \begin{pmatrix} \tilde{a}_0^T \\ \vdots \\ \tilde{a}_{m-1}^T \end{pmatrix} B$
<pre> for j := 0, ..., n - 1 for i := 0, ..., m - 1 for p := 0, ..., k - 1 $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end end end </pre>	$m_b = \square, n_b = \square, \text{ and } k_b = \square:$ $\left(c_0 \mid \cdots \mid c_{n-1} \right) +:= A \left(b_0 \mid \cdots \mid b_{n-1} \right)$
<pre> for j := 0, ..., n - 1 for i := 0, ..., m - 1 for p := 0, ..., k - 1 $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end end end </pre>	$m_b = \square, n_b = \square, \text{ and } k_b = \square:$ $\left(c_0 \mid \cdots \mid c_{n-1} \right) +:= A \left(b_0 \mid \cdots \mid b_{n-1} \right)$
<pre> for p := 0, ..., k - 1 for i := 0, ..., m - 1 for j := 0, ..., n - 1 $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end end end </pre>	$m_b = \square, n_b = \square, \text{ and } k_b = \square:$ $C +:= \left(a_0 \mid \cdots \mid a_{k-1} \right) \begin{pmatrix} \tilde{b}_0^T \\ \vdots \\ \tilde{b}_{k-1}^T \end{pmatrix}$
<pre> for p := 0, ..., k - 1 for j := 0, ..., n - 1 for i := 0, ..., m - 1 $\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$ end end end </pre>	$m_b = \square, n_b = \square, \text{ and } k_b = \square:$ $C +:= \left(a_0 \mid \cdots \mid a_{k-1} \right) \begin{pmatrix} \tilde{b}_0^T \\ \vdots \\ \tilde{b}_{k-1}^T \end{pmatrix}$

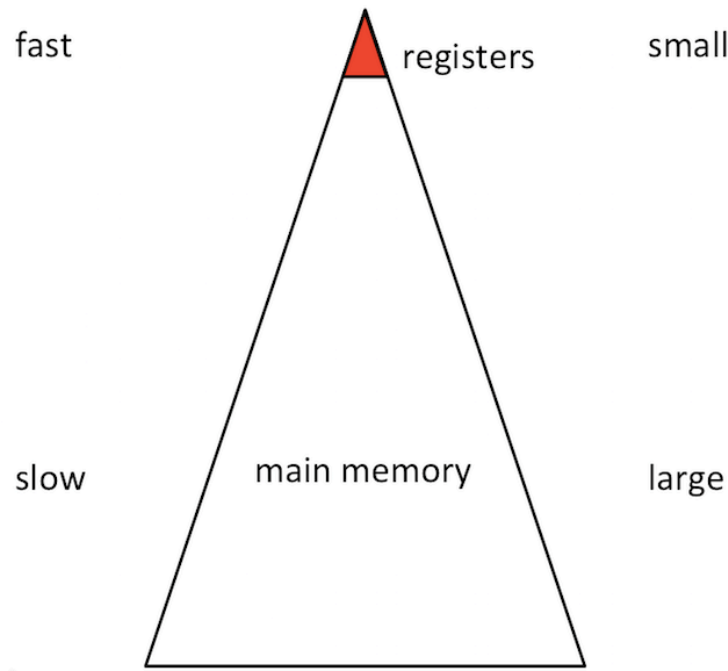


Figure 2.3: A simple model of the memory hierarchy: slow memory and fast registers.

- Performing a flop with data in registers takes time γ_R .
- Data movement and computation cannot overlap.

Later, we will change some of these assumptions.

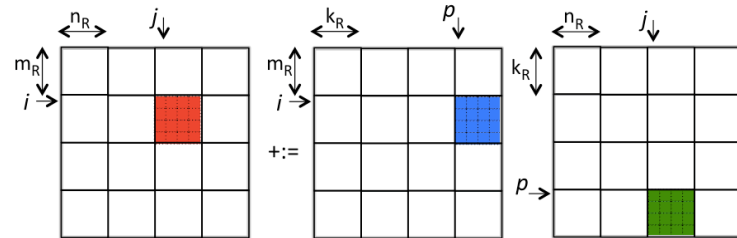
Given that the registers can hold 64 doubles, let's first consider blocking C , A , and B into 4×4 submatrices. In other words, consider the example from Unit 2.2.1, given in Figure 2.1, with $m_R = n_R = k_R = 4$. This means $3 \times 4 \times 4 = 48$ registers are being used for storing the submatrices. (Yes, we are not using all registers. We'll get to that later.) Let us call the routine that computes with three blocks "the kernel". Thus, the blocked algorithm is a triple-nested loop around a call to this kernel:

```

for  $i := 0, \dots, M - 1$ 
  for  $j := 0, \dots, N - 1$ 
    for  $p := 0, \dots, K - 1$ 
      Load  $C_{i,j}$ ,  $A_{i,p}$ , and  $B_{p,j}$ 
       $C_{i,j} := A_{i,p}B_{p,j} + C_{i,j}$ 
      Store  $C_{i,j}$ 
    end
  end
end

```

as illustrated by



The time required to complete this algorithm can be estimated as

$$\begin{aligned}
 & MNK(2m_Rn_Rk_R)\gamma_R + MNK(2m_Rn_R + m_Rk_R + k_Rn_R)\beta \\
 &= 2mnk\gamma_R + mnk\left(\frac{2}{k_R} + \frac{1}{n_R} + \frac{1}{m_R}\right)\beta,
 \end{aligned}$$

since $M = m/m_R$, $N = n/n_R$, and $K = k/k_R$,

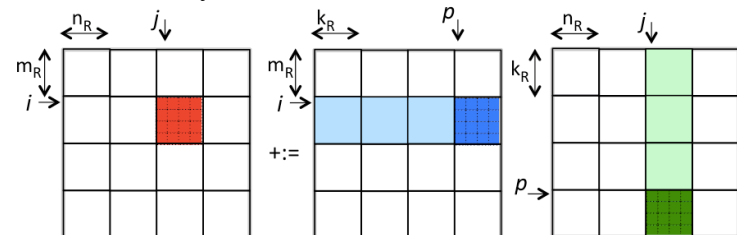
We next recognize that since the loop indexed with p is the inner-most loop, the loading and storing of $C_{i,j}$ does not need to happen before every call to the kernel, yielding the algorithm

```

for  $i := 0, \dots, M - 1$ 
  for  $j := 0, \dots, N - 1$ 
    Load  $C_{i,j}$ 
    for  $p := 0, \dots, K - 1$ 
      Load  $A_{i,p}$  and  $B_{p,j}$ 
       $C_{i,j} := A_{i,p}B_{p,j} + C_{i,j}$ 
    end
    Store  $C_{i,j}$ 
  end
end

```

as illustrated by



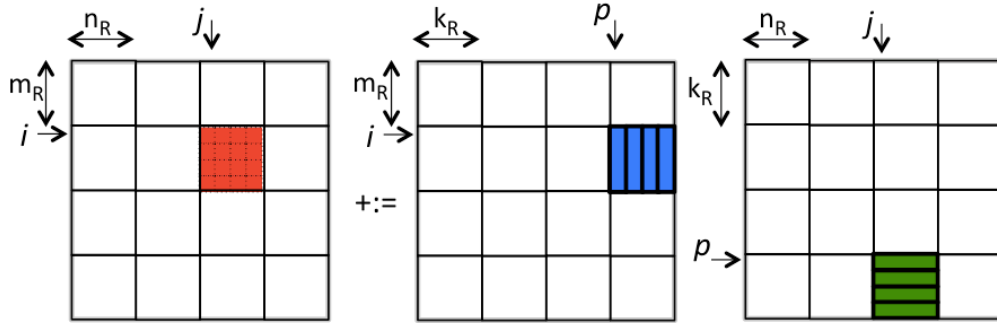
Homework 2.3.1.1 Compute the estimated cost of the last algorithm.

[SEE ANSWER](#)

Homework 2.3.1.2 In directory `Assignments/Week2/C/` copy the file `Gemm_IJP_mbxnbxb.c` into `Gemm_IJ_mbxnbxb.c` and remove the loop indexed by P . View its performance with `data/Plot_Blocked_MMM.mlx`.

[SEE ANSWER](#)

Now, if the submatrix $C_{i,j}$ were larger (i.e., m_R and n_R were larger), then the overhead can be reduced. Obviously, we can use more registers (we are only storing 48 of a possible 64 doubles in registers when $m_R = n_R = k_R = 4$). However, let's first discuss how to require fewer than 48 doubles to be stored while keeping $m_R = n_R = k_R = 4$. Recall from Week 1 that if the P loop of MMM is the outer-most loop, then the inner two loops implement a rank-1 update. If we do this for the kernel, then the result is illustrated by

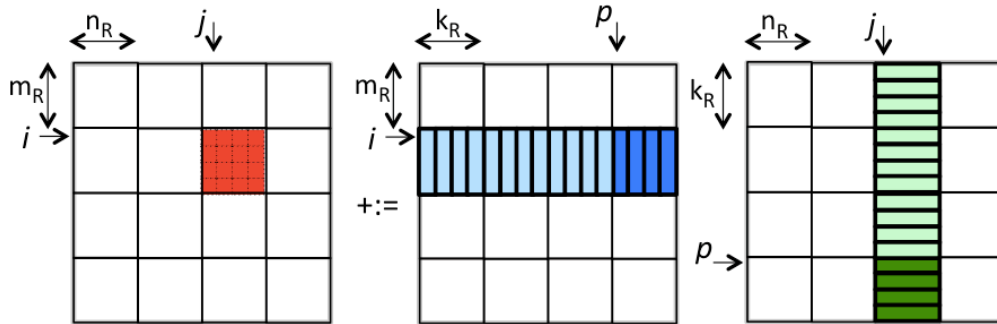


If we now consider the computation

$$\text{Red Block} + := \text{Blue Block} \text{ Green Block} = \text{Blue Column} \text{ Green Row} + \text{Blue Column} \text{ Green Row} + \text{Blue Column} \text{ Green Row} + \text{Blue Column} \text{ Green Row}$$

We recognize that after each rank-1 update, the column of $A_{i,p}$ and row of $B_{p,j}$ that participate in that rank-1 update are not needed again in the computation $C_{i,j} := A_{i,p}B_{p,j} + C_{i,j}$. Thus, only room for one such column of $A_{i,p}$ and one such row of $B_{p,j}$ needs to be reserved in the registers, reducing the total use of registers to $m_R \times n_R + m_R + n_R$. If $m_R = n_R = 4$ this equals $16 + 4 + 4 = 24$ doubles. That means in turn that, later, m_R and n_R can be chosen to be larger than if the entire blocks of A and B were stored.

If we now view the entire computation performed by the loop indexed with p , this can be illustrated by



and implemented, in terms of a call to the BLIS rank-1 update routine, in Figure 2.4.

What we now recognize is that matrices A and B need not be partitioned into $m_R \times k_R$ and $k_R \times n_R$ submatrices (respectively). Instead, A can be partitioned into $m_R \times k$ row panels and B into $k \times n_R$ column panels:

```

27 void Gemm_IJP_P_bli_dger( int m, int n, int k, double *A, int ldA,
28     double *B, int ldB, double *C, int ldC )
29 {
30     for ( int i=0; i<m; i+=MR ) /* m is assumed to be a multiple of MR */
31         for ( int j=0; j<n; j+=NR ) /* n is assumed to be a multiple of NR */
32             for ( int p=0; p<k; p+=KR ) /* k is assumed to be a multiple of KR */
33                 Gemm_P_bli_dger( MR, NR, KR, &alpha( i,p ), ldA,
34                     &beta( p,j ), ldB, &gamma( i,j ), ldC );
35 }
36
37 void Gemm_P_bli_dger( int m, int n, int k, double *A, int ldA,
38     double *B, int ldB, double *C, int ldC )
39 {
40     double d_one = 1.0;
41
42     for ( int p=0; p<k; p++ )
43         bli_dger( BLIS_NO_CONJUGATE, BLIS_NO_CONJUGATE, m, n, &d_one, &alpha( 0,p ), 1,
44             &beta( p,0 ), ldB, C, 1, ldC, NULL );
45 }

```

Assignments/Week2/C/Gemm_IJP_P_bli_dger.c

Figure 2.4: Blocked matrix-matrix multiplication with kernel that casts $C_{i,j} = A_{i,p}B_{p,j} + C_{i,j}$ in terms of rank-1 updates.

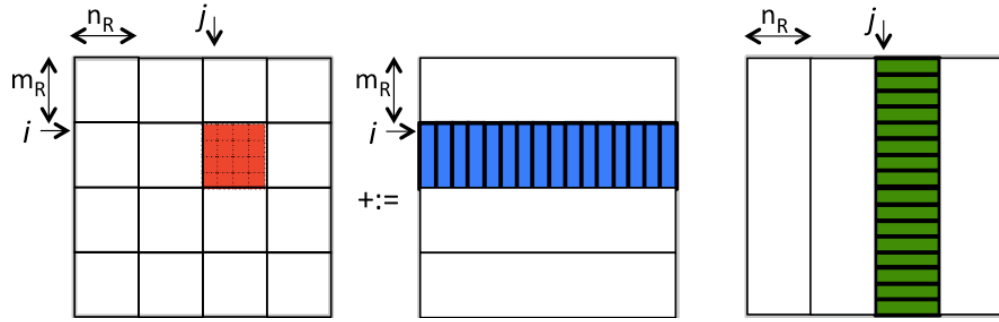
```

26
27 void Gemm_P_bli_dger( int m, int n, int k, double *A, int ldA,
28     double *B, int ldB, double *C, int ldC )
29 {
30     double d_one = 1.0;
31
32     for ( int p=0; p<k; p++ )
33         bli_dger( BLIS_NO_CONJUGATE, BLIS_NO_CONJUGATE, m, n, &d_one, &alpha( 0,p ), 1,
34             &beta( p,0 ), ldB, C, 1, ldC );
35 }

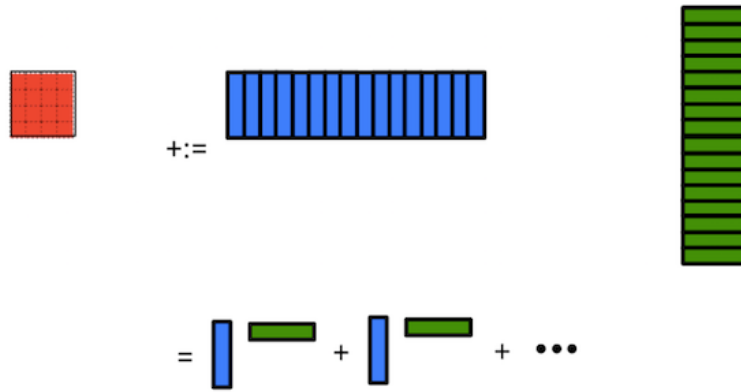
```

Assignments/Week2/C/Gemm_IJ_P_bli_dger.c

Figure 2.5: Blocked matrix-matrix multiplication that partitions A into row panels and B into column panels and casts the kernel in terms of rank-1 updates.



Another way of looking at this is that the inner loop indexed with p in the blocked algorithm can be merged with the outer loop of the kernel. The entire update of the $m_R \times n_R$ submatrix of C can then be pictured as



This is the computation that we will call the *micro-kernel* from here on.

Bringing $A_{i,p}$ one column at a time into registers and $B_{p,j}$ one row at a time into registers does not change the cost of reading that data. So, the cost of the resulting approach is still estimated as

$$MNK(2m_R n_R k_R) \gamma_R + [MN(2m_R n_R) + MNK(m_R k_R + k_R n_R)] \beta$$

$$= 2mnk \gamma_R + \left[2mn + mnk \left(\frac{1}{n_R} + \frac{1}{m_R} \right) \right] \beta.$$

We can arrive at the same algorithm by partitioning

$$C = \begin{pmatrix} C_{0,0} & C_{0,1} & \cdots & C_{0,N-1} \\ C_{1,0} & C_{1,1} & \cdots & C_{1,N-1} \\ \vdots & \vdots & & \vdots \\ C_{M-1,0} & C_{M-1,1} & \cdots & C_{M-1,N-1} \end{pmatrix}, A = \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_{M-1} \end{pmatrix}, \text{ and } B = \left(B_0 \mid B_1 \mid \cdots \mid B_{0,N-1} \right),$$

where $C_{i,j}$ is $m_R \times n_R$, A_i is $m_R \times k$, and B_j is $k \times n_R$. Then computing $C := AB + C$ means updating $C_{i,j} := A_i B_j + C_{i,j}$ for all i, j :

```

for  $i := 0, \dots, M - 1$ 
  for  $j := 0, \dots, N - 1$ 
     $C_{i,j} := A_i B_j + C_{i,j}$     } Computed with the micro-kernel
  end
end

```

Obviously, the order of the two loops can be switched. Again, the computation $C_{i,j} := A_i B_j + C_{i,j}$ where $C_{i,j}$ fits in registers now becomes what we will call the *micro-kernel*.

Homework 2.3.1.3 In directory `Assignments/Week2/C/` examine the file [Gemm_IJ_P_bli_dger.c](#). Execute it with `make IJ_P_bli_dger` and view its performance with [data/Plot_Blocked_MMM.mlx](#).

[SEE ANSWER](#)

2.3.2 Of vector registers and instructions, and instruction-level parallelism

As the reader should have noticed by now, in MMM for every floating point multiplication a corresponding floating point addition is encountered to accumulate the result. For this reason, such floating point computations are usually cast in terms of *fused multiply add* (FMA) operations, performed by a floating point unit (FPU) of the core.

What is faster than computing one FMA at a time? Computing multiple FMAs at a time! To accelerate computation, modern cores are equipped with vector registers, vector instructions, and vector floating point units that can simultaneously perform multiple FMA operations. This is called *instruction-level parallelism*.

In this section, we are going to use Intel's *vector intrinsic instructions* for Intel's AVX instruction set to illustrate the ideas. Vector intrinsics are supported in the C programming language for performing these vector instructions. We illustrate how the intrinsic operations are used by incorporating them into an implementation of the micro-kernel for the case where $m_R \times n_R = 4 \times 4$. Thus, for the remainder of this unit, C is $m_R \times n_R = 4 \times 4$, A is $m_R \times k = 4 \times k$, and B is $k \times n_R = k \times 4$.

Now, $C+ := AB$ when C is 4×4 translates to the computation

$$\begin{pmatrix} \gamma_{0,0} & \gamma_{0,1} & \gamma_{0,2} & \gamma_{0,3} \\ \gamma_{1,0} & \gamma_{1,1} & \gamma_{1,2} & \gamma_{1,3} \\ \gamma_{2,0} & \gamma_{2,1} & \gamma_{2,2} & \gamma_{2,3} \\ \gamma_{3,0} & \gamma_{3,1} & \gamma_{3,2} & \gamma_{3,3} \end{pmatrix} + := \begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots \\ \alpha_{1,0} & \alpha_{1,1} & \cdots \\ \alpha_{2,0} & \alpha_{2,1} & \cdots \\ \alpha_{3,0} & \alpha_{3,1} & \cdots \end{pmatrix} \begin{pmatrix} \beta_{0,0} & \beta_{0,1} & \beta_{0,2} & \beta_{0,3} \\ \beta_{1,0} & \beta_{1,1} & \beta_{1,2} & \beta_{1,3} \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

$$\begin{aligned}
&= \begin{pmatrix} \alpha_{0,0}\beta_{0,0} + \alpha_{0,1}\beta_{1,0} + \cdots & \alpha_{0,0}\beta_{0,1} + \alpha_{0,1}\beta_{1,1} + \cdots & \alpha_{0,0}\beta_{0,2} + \alpha_{0,1}\beta_{1,2} + \cdots & \alpha_{0,0}\beta_{0,3} + \alpha_{0,1}\beta_{1,3} + \cdots \\ \alpha_{1,0}\beta_{0,0} + \alpha_{1,1}\beta_{1,0} + \cdots & \alpha_{1,0}\beta_{0,1} + \alpha_{1,1}\beta_{1,1} + \cdots & \alpha_{1,0}\beta_{0,2} + \alpha_{1,1}\beta_{1,2} + \cdots & \alpha_{1,0}\beta_{0,3} + \alpha_{1,1}\beta_{1,3} + \cdots \\ \alpha_{2,0}\beta_{0,0} + \alpha_{2,1}\beta_{1,0} + \cdots & \alpha_{2,0}\beta_{0,1} + \alpha_{2,1}\beta_{1,1} + \cdots & \alpha_{2,0}\beta_{0,2} + \alpha_{2,1}\beta_{1,2} + \cdots & \alpha_{2,0}\beta_{0,3} + \alpha_{2,1}\beta_{1,3} + \cdots \\ \alpha_{3,0}\beta_{0,0} + \alpha_{3,1}\beta_{1,0} + \cdots & \alpha_{3,0}\beta_{0,1} + \alpha_{3,1}\beta_{1,1} + \cdots & \alpha_{3,0}\beta_{0,2} + \alpha_{3,1}\beta_{1,2} + \cdots & \alpha_{3,0}\beta_{0,3} + \alpha_{3,1}\beta_{1,3} + \cdots \end{pmatrix} \\
&= \begin{pmatrix} \alpha_{0,0} \\ \alpha_{1,0} \\ \alpha_{2,0} \\ \alpha_{3,0} \end{pmatrix} \begin{pmatrix} \beta_{0,0} & \beta_{0,1} & \beta_{0,2} & \beta_{0,3} \end{pmatrix} + \begin{pmatrix} \alpha_{0,1} \\ \alpha_{1,1} \\ \alpha_{2,1} \\ \alpha_{3,1} \end{pmatrix} \begin{pmatrix} \beta_{1,0} & \beta_{1,1} & \beta_{1,2} & \beta_{1,3} \end{pmatrix} + \cdots
\end{aligned}$$

Thus, updating 4×4 matrix C can be implemented as a loop around rank-1 updates:

$$\begin{aligned}
&\textbf{for } p = 0, \dots, k-1 \\
&\quad \begin{pmatrix} \gamma_{0,0} & \gamma_{0,1} & \gamma_{0,2} & \gamma_{0,3} \\ \gamma_{1,0} & \gamma_{1,1} & \gamma_{1,2} & \gamma_{1,3} \\ \gamma_{2,0} & \gamma_{2,1} & \gamma_{2,2} & \gamma_{2,3} \\ \gamma_{3,0} & \gamma_{3,1} & \gamma_{3,2} & \gamma_{3,3} \end{pmatrix} + := \begin{pmatrix} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{pmatrix} \begin{pmatrix} \beta_{p,0} & \beta_{p,1} & \beta_{p,2} & \beta_{p,3} \end{pmatrix} \\
&\textbf{endfor}
\end{aligned}$$

or, equivalently to emphasize computations with vectors,

$$\begin{aligned}
&\textbf{for } p = 0, \dots, k-1 \\
&\quad \begin{pmatrix} \gamma_{0,0} + := \alpha_{0,p} \times \beta_{p,0} & \gamma_{0,1} + := \alpha_{0,p} \times \beta_{p,1} & \gamma_{0,2} + := \alpha_{0,p} \times \beta_{p,2} & \gamma_{0,3} + := \alpha_{0,p} \times \beta_{p,3} \\ \gamma_{1,0} + := \alpha_{1,p} \times \beta_{p,0} & \gamma_{1,1} + := \alpha_{1,p} \times \beta_{p,1} & \gamma_{1,2} + := \alpha_{1,p} \times \beta_{p,2} & \gamma_{1,3} + := \alpha_{1,p} \times \beta_{p,3} \\ \gamma_{2,0} + := \alpha_{2,p} \times \beta_{p,0} & \gamma_{2,1} + := \alpha_{2,p} \times \beta_{p,1} & \gamma_{2,2} + := \alpha_{2,p} \times \beta_{p,2} & \gamma_{2,3} + := \alpha_{2,p} \times \beta_{p,3} \\ \gamma_{3,0} + := \alpha_{3,p} \times \beta_{p,0} & \gamma_{3,1} + := \alpha_{3,p} \times \beta_{p,1} & \gamma_{3,2} + := \alpha_{3,p} \times \beta_{p,2} & \gamma_{3,3} + := \alpha_{3,p} \times \beta_{p,3} \end{pmatrix} \\
&\textbf{endfor}
\end{aligned}$$

This, once again, shows how MMM can be cast in terms of a sequence of rank-1 updates, and that a rank-1 update can be implemented in terms of AXPY operations with columns of C .

Now we translate this into code that employs vector instructions, illustrated in Figure 2.8 and given in Figures 2.6-2.7. To use the intrinsics, we start by including the header file `immintrin.h`.

25

}

The declaration

32

{

creates `gamma_0123_0` as a variable that references a vector register with four double precision numbers and loads it with the four numbers that are stored starting at address `&gamma(0, 0)`. In

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define alpha( i,j ) A[ (j)*ldA + (i) ] // map alpha( i,j ) to array A
5 #define beta( i,j ) B[ (j)*ldB + (i) ] // map beta( i,j ) to array B
6 #define gamma( i,j ) C[ (j)*ldC + (i) ] // map gamma( i,j ) to array C
7
8 #define MR 4
9 #define NR 4
10
11 void Gemm_IJ_4x4Kernel( int, int, int, double *, int, double *, int, double *, int );
12
13 void Gemm_4x4Kernel( int, double *, int, double *, int, double *, int );
14
15 void MyGemm( int m, int n, int k, double *A, int ldA,
16             double *B, int ldB, double *C, int ldC )
17 {
18     if ( m % MR != 0 || n % NR != 0 ){
19         printf( "m and n must be multiples of MR and NR, respectively \n" );
20         exit( 0 );
21     }
22
23     for ( int i=0; i<m; i+=MR ) /* m is assumed to be a multiple of MR */
24         for ( int j=0; j<n; j+=NR ) /* n is assumed to be a multiple of NR */
```

Assignments/Week2/C/Gemm_IJ_4x4Kernel.c

Figure 2.6: General framework for calling a kernel, instantiated for the case where $m_R = n_R = 4$. (Continued in Figure 2.7.)

```
26 }
27
28 #include<immintrin.h>
29
30 void Gemm_4x4Kernel( int k, double *A, int ldA, double *B, int ldB,
31 double *C, int ldC )
32 {
33     /* Declare vector registers to hold 4x4 C and load them */
34     __m256d gamma_0123_0 = _mm256_loadu_pd( &gamma( 0,0 ) );
35     __m256d gamma_0123_1 = _mm256_loadu_pd( &gamma( 0,1 ) );
36     __m256d gamma_0123_2 = _mm256_loadu_pd( &gamma( 0,2 ) );
37     __m256d gamma_0123_3 = _mm256_loadu_pd( &gamma( 0,3 ) );
38
39     for ( int p=0; p<k; p++ ){
40         /* Declare vector register for load/broadcasting beta( b,j ) */
41         __m256d beta_p_j;
42
43         /* Declare a vector register to hold the current column of A and load
44            it with the four elements of that column. */
45         __m256d alpha_0123_p = _mm256_loadu_pd( &alpha( 0,p ) );
46
47         /* Load/broadcast beta( p,0 ). */
48         beta_p_j = _mm256_broadcast_sd( &beta( p, 0 ) );
49
50         /* update the first column of C with the current column of A times
51            beta ( p,0 ) */
52         gamma_0123_0 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_0 );
53
54         /* REPEAT for second, third, and fourth columns of C. Notice that the
55            current column of A needs not be reloaded. */
56
57         /* Load/broadcast beta( p,1 ). */
58         beta_p_j = _mm256_broadcast_sd( &beta( p, 1 ) );
59
60         /* update the second column of C with the current column of A times
61            beta ( p,1 ) */
62         gamma_0123_1 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_1 );
63
64         /* Load/broadcast beta( p,2 ). */
65         beta_p_j = _mm256_broadcast_sd( &beta( p, 2 ) );
66
67         /* update the third column of C with the current column of A times
68            beta ( p,2 ) */
69         gamma_0123_2 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_2 );
70
71         /* Load/broadcast beta( p,3 ). */
72         beta_p_j = _mm256_broadcast_sd( &beta( p, 3 ) );
73
74         /* update the fourth column of C with the current column of A times
75            beta ( p,3 ) */
76         gamma_0123_3 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_3 );
77     }
78
79     /* Store the updated results */
80     _mm256_storeu_pd( &gamma(0,0), gamma_0123_0 );
81     _mm256_storeu_pd( &gamma(0,1), gamma_0123_1 );
82     _mm256_storeu_pd( &gamma(0,2), gamma_0123_2 );
83     _mm256_storeu_pd( &gamma(0,3), gamma_0123_3 );
```

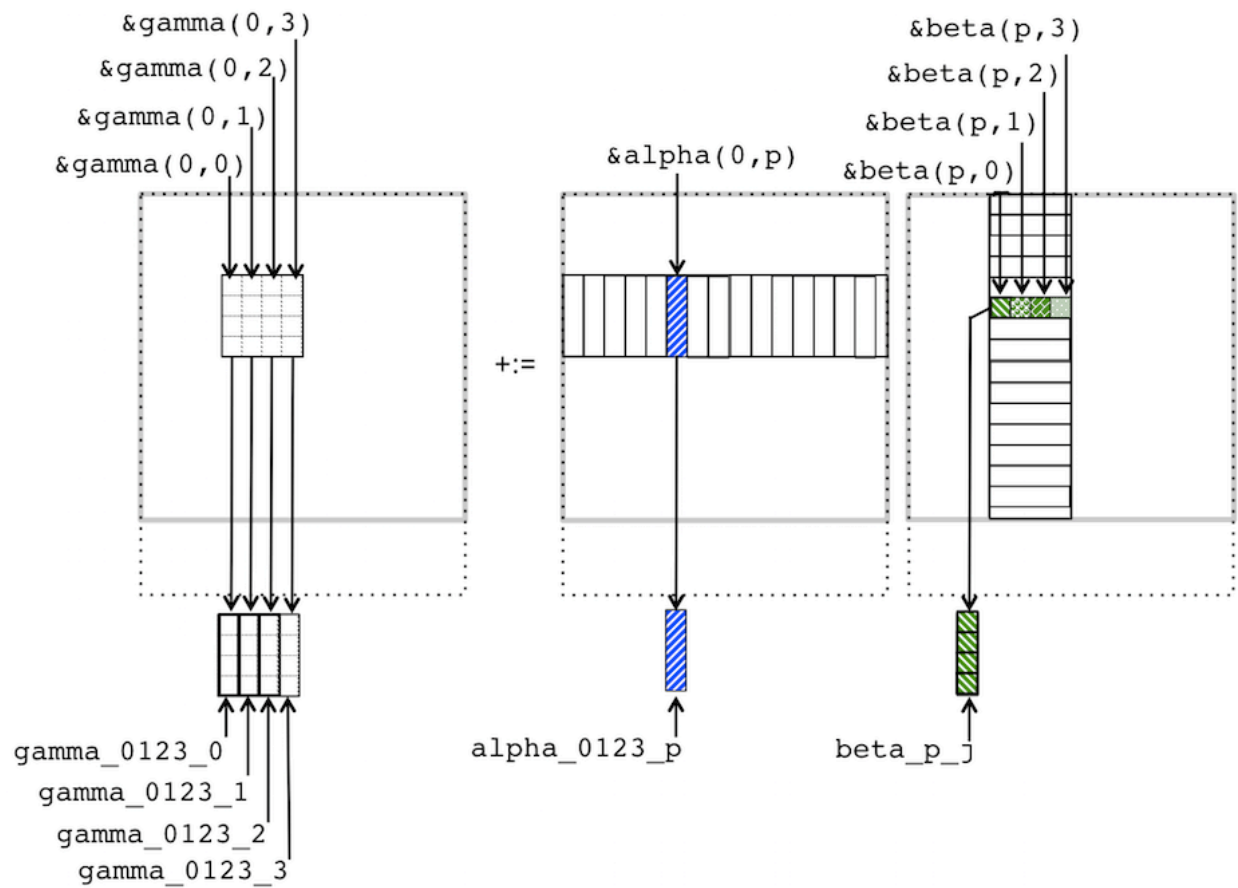


Figure 2.8: Illustration of how the routine in Figure 2.7 indexes into the matrices.

other words, it loads the vector register with the original values

$$\begin{pmatrix} \gamma_{0,0} \\ \gamma_{1,0} \\ \gamma_{2,0} \\ \gamma_{3,0} \end{pmatrix}.$$

This is repeated for the other three columns of C :

```

33  /* Declare vector registers to hold 4x4 C and load them */
34  __m256d gamma_0123_0 = _mm256_loadu_pd( &gamma( 0,0 ) );
35  __m256d gamma_0123_1 = _mm256_loadu_pd( &gamma( 0,1 ) );

```

The loop in Figure 2.7 implements

for $p = 0, \dots, k-1$

$\gamma_{0,0+} := \alpha_{0,p} \times \beta_{p,0}$	$\gamma_{0,1+} := \alpha_{0,p} \times \beta_{p,1}$	$\gamma_{0,2+} := \alpha_{0,p} \times \beta_{p,2}$	$\gamma_{0,3+} := \alpha_{0,p} \times \beta_{p,3}$
$\gamma_{1,0+} := \alpha_{1,p} \times \beta_{p,0}$	$\gamma_{1,1+} := \alpha_{1,p} \times \beta_{p,1}$	$\gamma_{1,2+} := \alpha_{1,p} \times \beta_{p,2}$	$\gamma_{1,3+} := \alpha_{1,p} \times \beta_{p,3}$
$\gamma_{2,0+} := \alpha_{2,p} \times \beta_{p,0}$	$\gamma_{2,1+} := \alpha_{2,p} \times \beta_{p,1}$	$\gamma_{2,2+} := \alpha_{2,p} \times \beta_{p,2}$	$\gamma_{2,3+} := \alpha_{2,p} \times \beta_{p,3}$
$\gamma_{3,0+} := \alpha_{3,p} \times \beta_{p,0}$	$\gamma_{3,1+} := \alpha_{3,p} \times \beta_{p,1}$	$\gamma_{3,2+} := \alpha_{3,p} \times \beta_{p,2}$	$\gamma_{3,3+} := \alpha_{3,p} \times \beta_{p,3}$

endfor

leaving the result in the vector registers. Each iteration starts by declaring vector register variable `alpha_0123_p` and loading it with the contents of

$$\begin{pmatrix} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{pmatrix}.$$

43 `/* Declare a vector register to hold the current column of A and load`

Next, $\beta_{p,0}$ is loaded into a vector register, broadcasting (duplicating) that value to each entry in that register:

The command

52 `gamma_0123_0 = _mm256_fmadd_pd(alpha_0123_p, beta_p_j, gamma_0123_0);`

then performs the computation

$$\begin{pmatrix} \gamma_{0,0+} := \alpha_{0,p} \times \beta_{p,0} \\ \gamma_{1,0+} := \alpha_{1,p} \times \beta_{p,0} \\ \gamma_{2,0+} := \alpha_{2,p} \times \beta_{p,0} \\ \gamma_{3,0+} := \alpha_{3,p} \times \beta_{p,0} \end{pmatrix}.$$

We leave it to the reader to add the commands that compute

$$\begin{pmatrix} \gamma_{0,1+} := \alpha_{0,p} \times \beta_{p,1} \\ \gamma_{1,1+} := \alpha_{1,p} \times \beta_{p,1} \\ \gamma_{2,1+} := \alpha_{2,p} \times \beta_{p,1} \\ \gamma_{3,1+} := \alpha_{3,p} \times \beta_{p,1} \end{pmatrix}, \quad \begin{pmatrix} \gamma_{0,2+} := \alpha_{0,p} \times \beta_{p,2} \\ \gamma_{1,2+} := \alpha_{1,p} \times \beta_{p,2} \\ \gamma_{2,2+} := \alpha_{2,p} \times \beta_{p,2} \\ \gamma_{3,2+} := \alpha_{3,p} \times \beta_{p,2} \end{pmatrix}, \quad \text{and} \quad \begin{pmatrix} \gamma_{0,3+} := \alpha_{0,p} \times \beta_{p,3} \\ \gamma_{1,3+} := \alpha_{1,p} \times \beta_{p,3} \\ \gamma_{2,3+} := \alpha_{2,p} \times \beta_{p,3} \\ \gamma_{3,3+} := \alpha_{3,p} \times \beta_{p,3} \end{pmatrix}.$$

Upon completion of the loop, the results are stored back into the original arrays with the commands

```
57  /* Load/broadcast beta( p,1 ). */
58  beta_p_j = _mm256_broadcast_sd( &beta( p, 1) );
59
60  /* update the second column of C with the current column of A times
```

Homework 2.3.2.1 In directory `Assignments/Week2/C/` complete the code in file [Gemm_IJ_4x4Kernel.c](#). View its performance with [data/Plot_Blocked_MMM.mlx](#).

[SEE ANSWER](#)

Homework 2.3.2.2 In directory `Assignments/Week2/C/` copy file [Gemm_IJ_4x4Kernel.c](#) to `Gemm_JI_4x4Kernel.c` and reverse the I and J loops. View its performance with [data/Plot_Blocked_MMM.mlx](#).

[SEE ANSWER](#)

The form of parallelism illustrated here is often referred to as Single Instruction, Multiple Data (SIMD) parallelism. The same (FMA) instruction is performed with all data in the vector registers.

On Robert's laptop, we are starting to see some progress towards high performance, as illustrated in Figure 2.9.

2.4 Optimizing the Microkernel

2.4.1 Reuse of data in registers

In Section 2.3.2, we introduced the fact that modern architectures have vector registers and then showed how to (somewhat) optimize the update of a 4×4 submatrix of C .

Some observations:

- Four vector registers are used for the submatrix of C . Once loaded, those values remain in the registers for the duration of the computation, until they are finally written out once they have been completely updated. It is important to keep values of C in registers that long, because values of C are read as well as written, unlike values from A and B , which are read but not written.
- The block is 4×4 . In general, the block is $m_R \times n_R$, where in this case $m_R = n_R = 4$. If we assume we will use R_C registers for elements of the submatrix of C , what should m_R and n_R be, given the constraint that $m_R \times n_R \leq R_C$?
- Given that some registers need to be used for elements of A and B , how many registers should be used for elements of each of the matrices? In our implementation in the last section, we needed one vector register for four elements of A and one vector register in which one element of B is broadcast (duplicated). Thus, $16 + 4 + 1$ elements of matrices were being stored in 6 out of 16 available vector registers.

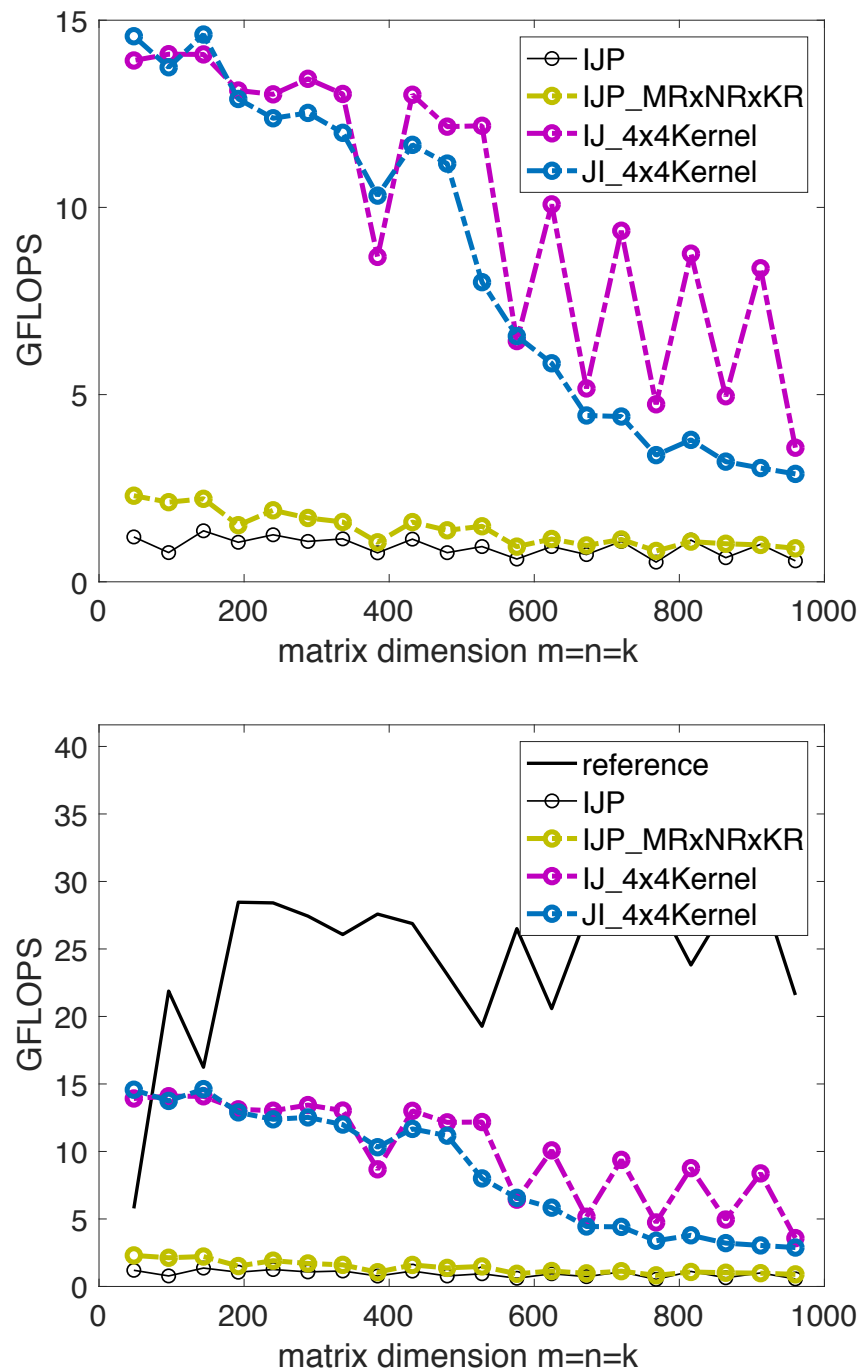


Figure 2.9: Performance of a double loop around a micro-kernel implemented with vector instructions, for $m_R = n_R = 4$.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define alpha( i,j ) A[ (j)*ldA + (i) ] // map alpha( i,j ) to array A
5 #define beta( i,j ) B[ (j)*ldB + (i) ] // map beta( i,j ) to array B
6 #define gamma( i,j ) C[ (j)*ldC + (i) ] // map gamma( i,j ) to array C
7
8 #define MR 4
9 #define NR 4
10
11 void Gemm_IJ_4x4Kernel( int, int, int, double *, int, double *, int, double *, int );
12
13 void Gemm_4x4Kernel( int, double *, int, double *, int, double *, int );
14
15 void MyGemm( int m, int n, int k, double *A, int ldA,
16             double *B, int ldB, double *C, int ldC )
17 {
18     if ( m % MR != 0 || n % NR != 0 ){
19         printf( "m and n must be multiples of MR and NR, respectively \n" );
20         exit( 0 );
21     }
22
23     for ( int i=0; i<m; i+=MR ) /* m is assumed to be a multiple of MR */
24         for ( int j=0; j<n; j+=NR ) /* n is assumed to be a multiple of NR */

```

Assignments/Week2/C/Gemm_IJ_4x4Kernel.c

Figure 2.10: General framework for calling a kernel, instantiated for the case where $m_R = n_R = 4$. (Continued in Figure 2.11.)

What we notice is that, ignoring the cost of loading and storing elements of C , loading four doubles from matrix A and broadcasting four doubles from matrix B is amortized over 32 flops. The ratio is 32 flops for 8 loads, or 4 flops/load. Here for now we ignore the fact that loading the four elements of A is less costly than broadcasting the four elements of B .

```

26 }
27
28 #include<immintrin.h>
29
30 void Gemm_4x4Kernel( int k, double *A, int ldA, double *B, int ldB,
31 double *C, int ldC )
32 {
33     /* Declare vector registers to hold 4x4 C and load them */
34     __m256d gamma_0123_0 = _mm256_loadu_pd( &gamma( 0,0 ) );
35     __m256d gamma_0123_1 = _mm256_loadu_pd( &gamma( 0,1 ) );
36     __m256d gamma_0123_2 = _mm256_loadu_pd( &gamma( 0,2 ) );
37     __m256d gamma_0123_3 = _mm256_loadu_pd( &gamma( 0,3 ) );
38
39     for ( int p=0; p<k; p++ ){
40         /* Declare vector register for load/broadcasting beta( b,j ) */
41         __m256d beta_p_j;
42
43         /* Declare a vector register to hold the current column of A and load
44            it with the four elements of that column. */
45         __m256d alpha_0123_p = _mm256_loadu_pd( &alpha( 0,p ) );
46
47         /* Load/broadcast beta( p,0 ). */
48         beta_p_j = _mm256_broadcast_sd( &beta( p, 0 ) );
49
50         /* update the first column of C with the current column of A times
51            beta ( p,0 ) */
52         gamma_0123_0 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_0 );
53
54         /* REPEAT for second, third, and fourth columns of C. Notice that the
55            current column of A needs not be reloaded. */
56
57         /* Load/broadcast beta( p,1 ). */
58         beta_p_j = _mm256_broadcast_sd( &beta( p, 1 ) );
59
60         /* update the second column of C with the current column of A times
61            beta ( p,1 ) */
62         gamma_0123_1 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_1 );
63
64         /* Load/broadcast beta( p,2 ). */
65         beta_p_j = _mm256_broadcast_sd( &beta( p, 2 ) );
66
67         /* update the third column of C with the current column of A times
68            beta ( p,2 ) */
69         gamma_0123_2 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_2 );
70
71         /* Load/broadcast beta( p,3 ). */
72         beta_p_j = _mm256_broadcast_sd( &beta( p, 3 ) );
73
74         /* update the fourth column of C with the current column of A times
75            beta ( p,3 ) */
76         gamma_0123_3 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_3 );
77     }
78
79     /* Store the updated results */
80     _mm256_storeu_pd( &gamma(0,0), gamma_0123_0 );
81     _mm256_storeu_pd( &gamma(0,1), gamma_0123_1 );
82     _mm256_storeu_pd( &gamma(0,2), gamma_0123_2 );
83     _mm256_storeu_pd( &gamma(0,3), gamma_0123_3 );

```

2.4.2 More options

Homework 2.4.2.1 Modify [Gemm_JI_4x4Kernel.c](#) to implement the case where $m_R = 8$ and $n_R = 4$, storing the result in `Gemm_JI_8x4Kernel.c`. Don't forget to update MR and NR! You can test the result by typing

```
make JI_8x4Kernel
```

in that directory and view the resulting performance by appropriately modifying [data/Plot_MRxNR_Kernels.mlx](#).

[SEE ANSWER](#)

Homework 2.4.2.2 How many vector registers are needed for the implementation in Homework 2.4.2.1?

Ignoring the cost of loading the registers with the 8×4 submatrix of C , analyze the ratio of flops to loads for the implementation in Homework 2.4.2.1.

[SEE ANSWER](#)

Homework 2.4.2.3 We have considered $m_R \times n_R = 4 \times 4$ and $m_R \times n_R = 8 \times 4$, where elements of A are loaded without duplication into vector registers (and hence m_R must be a multiple of 4), and elements of B are loaded/broadcast. Extending this approach to loading A and B , complete the entries in the following table:

$m_R \times n_R$	# vector registers	flops/load
4×1		/ =
4×2		/ =
4×4	6	$32 / 8 = 4$
4×12		/ =
4×14		/ =
8×4		/ =
8×6		/ =
12×4		/ =
16×2		/ =
16×3		/ =

[SEE ANSWER](#)

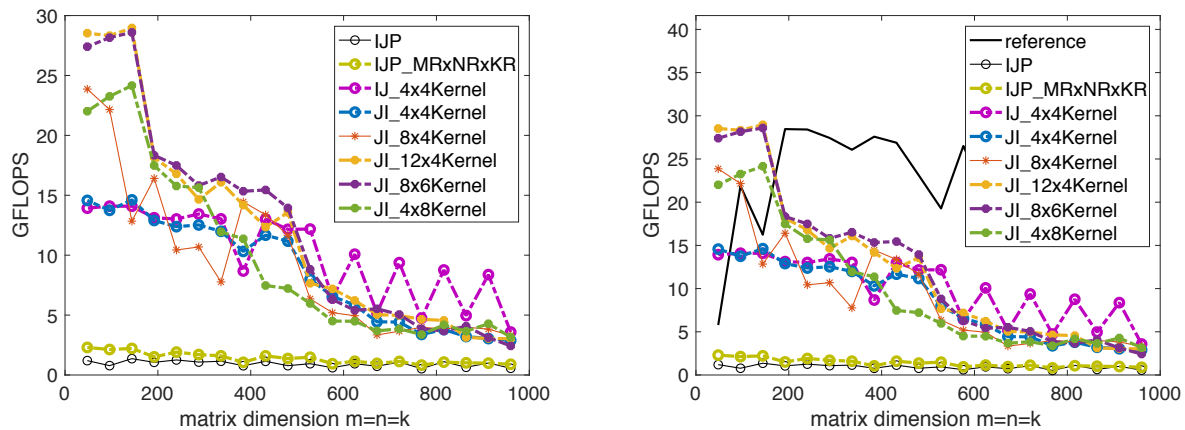


Figure 2.12: Performance of implementations for various choices of $m_R \times n_R$.

Homework 2.4.2.4 At this point, you have already implemented the following kernels:

`Gemm_JI_4x4Kernel.c` and `Gemm_JI_8x4Kernel.c`.

Implement as many of the more promising of the kernels you analyzed in the last homework as you like. They can be executed by typing

```
make test_JI_?x?Kernel
```

where the ?'s are replaced with the obvious choices of m_R and n_R . Don't forget to update MR and NR! The resulting performance can again be viewed with Live Script in [data/Plot_MRxNR_Kernels.mlx](#).

[SEE ANSWER](#)

In Figure 2.12, we show the performance of MMM cast in terms of various micro-kernel calls on Robert's laptop.

2.4.3 Amortizing data movement

As we have already discussed, one of the fundamental principles behind high performance is that when data is moved from one layer of memory to another, one needs to amortize the cost of that data movement over many computations. In our discussion so far, we have only discussed two layers of memory: (vector) registers and a cache.

Let us focus on the current approach:

- Four vector registers are loaded with a $m_R \times n_R$ submatrix of C . The cost of loading and unloading this data is negligible if k size is large.

- $m_R/4$ vector register is loaded with the current column of A , with m_R elements. In our picture, we call this register `alpha_0123_p`. The cost of this load is amortized over $2m_R \times n_R$ floating point operations.
- One vector register is loaded with broadcast elements of the current row of B . The loading of the n_R of the elements is also amortized over $2m_R \times n_R$ floating point operations.

To summarize, ignoring the cost of loading the submatrix of C , this would then require

$$m_R + n_R$$

floating point number loads from slow memory, m_R elements of A and n_R elements of B , which are amortized over $2 \times m_R \times n_R$ flops, for each rank-1 update.

If loading a double as part of vector load costs the same as the loading and broadcasting of a double, then the problem of finding the optimal choices of m_R and n_R can be stated as finding the m_R and n_R that optimize

$$\frac{2m_R n_R}{m_R + n_R}$$

under the constraint that

$$m_R n_R + m_R + 1 = r_C,$$

where r_C equals the number of doubles that can be stored in vector registers.

Finding the optimal solution is somewhat nasty, even if m_R and n_R are allowed to be real valued rather than integers. If we recognize that $m_R + 1$ is small relative to $m_R n_R$, and therefore ignore that those need to be kept in registers, we instead end up with having to find the m_R and n_R that optimize

$$\frac{2m_R n_R}{m_R + n_R}$$

under the constraint that

$$m_R n_R = r_C.$$

This is essentially a standard problem in calculus, used to find the sides of the rectangle with length x and width y that maximizes the area of the rectangle, xy , under the constraint that the length of the perimeter is constant. The answer for that problem is that the rectangle should be a square. Thus, we want the submatrix of C that is kept in registers to be “squarish.”

In practice, we have seen that the shape of the block of C kept in registers is noticeably not square. The reason for this is that, all other issues being equal, loading a double as part of a vector load requires fewer cycles than loading a double as part of a load/broadcast.

2.4.4 Discussion

Notice that we have analyzed that we must amortize the loading of elements of A and B over, at best, a few flops per such element. The “latency” from main memory (the time it takes for a double precision number to be fetched from main memory) is equal to the time it takes to perform on the order of 100 flops. Thus, “feeding the beast” from main memory spells disaster. Fortunately, modern architectures are equipped with multiple layers of cache memories that have a lower latency.

In Figure 2.12, we saw that for small matrices that fit in the L-2 cache, performance is very good, especially for the right choices of $m_R \times n_R$. The performance drops when problem sizes spill out of the L2 cache and drops again when they spill out of the L3 cache.

2.5 When Optimal Means Optimal

2.5.1 Reasoning about optimality

Early in our careers, we learned that if you say that an implementation is optimal, you better prove that it is optimal.

In our empirical studies, graphing the measured performance, we can compare our achieved results to the theoretical peak. Obviously, if we achieved theoretical peak performance, then we would know that the implementation is optimal. The problem is that we rarely achieve the theoretical peak performance as computed so far (multiplying the clock rate by the number of floating point operations that can be performed per clock cycle).

In order to claim optimality, one must carefully model an architecture and compute, through analysis, the exact limit of what it theoretical can achieve. Then, one can check achieved performance against the theoretical limit, and make a claim. Usually, this is also not practical.

In practice, one creates a model of computation for a simplified architecture. With that, one then computes a theoretical limit on performance. The next step is to show that the theoretical limit can be (nearly) achieved by an algorithm that executes on that simplified architecture. This then says something about the optimality of the algorithm under idealized circumstances. By finally comparing and contrasting the simplified architecture with an actual architecture, and the algorithm that targets the simplified architecture with an actual algorithm designed for the actual architecture, one can reason about the optimality, or lack thereof, of the practical algorithm.

In this section, we will illustrate this process with MMM.

2.5.2 A simple model

Let us give a simple model of computation that matches what we have assumed so far when programming matrix-matrix multiplication:

- We wish to compute $C := AB + C$ where C , A , and C are $m \times n$, $m \times k$, and $k \times n$, respectively.
 - The computation is cast in terms of FMAs.
 - Our machine has two layers of memory: fast memory (registers) and slow memory (main memory).
 - Initially, data reside in main memory.
 - To compute a FMA, all three operands must be in fast memory.
 - Fast memory can hold at most S floats.
-

- Slow memory is large enough that its size is not relevant to this analysis.
- Computation cannot be overlapped with data movement.

Notice that this model matches pretty well how we have viewed our processor so far.

2.5.3 Minimizing data movement

We have seen that matrix-matrix multiplication requires $m \times n \times k$ FMA operations, or $2mnk$ flops. Executing floating point operations constitutes useful computation. Moving data between slow memory and fast memory is overhead since we assume it cannot be overlapped. Hence, under our simplified model, if an algorithm only performs the minimum number of flops (namely $2mnk$), minimizes the time spent moving data between memory layers, and we at any given time are either performing useful computation (flops) or moving data, then we can argue that (under our model) the algorithm is optimal.

We now focus the argument by reasoning about a lower bound on the number of data that must be moved from fast memory to slow memory. We will build the argument with a sequence of observations.

Consider the loop

```

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

```

One can view the computations

$$\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}$$

as a cube of points in 3D, (i, j, k) for $0 \leq i < m$, $0 \leq j < n$, $0 \leq p < k$. The set of all possible algorithms that execute each such update only once can be viewed as an arbitrary ordering on that set of points. We can think of this as indexing the set of all such triples with i_r, j_r, p_r , $0 \leq r < m \times n \times k$:

$$(i_r, j_r, p_r).$$

so that the algorithm that computes $C := AB + C$ can then be written as

```

for  $r := 0, \dots, mnk-1$ 
   $\gamma_{i_r, j_r} := \alpha_{i_r, p_r} \beta_{p_r, j_r} + \gamma_{i_r, j_r}$ 
end

```

Obviously, this puts certain restrictions on i_r , j_r , and p_r . Articulating those exactly is not important right now.

We now partition the ordered set $0, \dots, mnk - 1$ into ordered contiguous subranges (phases) each of which requires $S + M$ distinct elements from A , B , and C ¹, except for the last phase, which will contain fewer². Recall that S equals the number of floats that fit in fast memory. A typical phase will start with S elements in fast memory, and will in addition read M elements from slow memory (except for the final phase). Such a typical phase will start at $r = R$ and consists of F triples, (i_R, j_R, p_R) through $(i_{R+F-1}, j_{R+F-1}, p_{R+F-1})$. Let us denote the set of these triples by \mathbf{D} . These represent F FMAs being performed in our algorithm. Even the first phase needs to read *at least* M elements since it will require $S + M$ elements to be read.

The key question now becomes what the upper bound on the number of FMAs is that can be performed with $S + M$ elements. Let us denote this bound by F_{\max} . If we know F_{\max} as a function of $S + M$, then we know that at least $\frac{mnk}{F_{\max}} - 1$ phases of FMAs need to be executed, where each of those phases requires at least S reads from slow memory. The total number of reads requires for *any* algorithm is thus at least

$$\left(\frac{mnk}{F_{\max}} - 1 \right) M. \quad (2.1)$$

To find F_{\max} we make a few observations:

- A typical triple $(i_r, j_r, p_r) \in \mathbf{D}$ represents a FMA that requires one element from each of matrices C , A , and B : $\gamma_{i,j}$, $\alpha_{i,p}$, and $\beta_{p,j}$.
- The set of all elements from C , γ_{i_r, j_r} , that are needed for the computations represented by the triples in \mathbf{D} are indexed with the tuples $\mathbf{C}_{\mathbf{D}} = \{(i_r, j_r) \mid R \leq r < R + F\}$. If we think of the triples in \mathbf{D} as points in 3D, then $\mathbf{C}_{\mathbf{D}}$ is the projection of those points onto the i, j plane. Its size, $|\mathbf{C}_{\mathbf{D}}|$, tells us how many elements of C must at some point be in fast memory during that phase of computations.
- The set of all elements from A , α_{i_r, p_r} , that are needed for the computations represented by the triples in \mathbf{D} are indexed with the tuples $\mathbf{A}_{\mathbf{D}} = \{(i_r, p_r) \mid R \leq r < R + F\}$. If we think of the triples (i_r, j_r, p_r) as points in 3D, then $\mathbf{A}_{\mathbf{D}}$ is the projection of those points onto the i, p plane. Its size, $|\mathbf{A}_{\mathbf{D}}|$, tells us how many elements of A must at some point be in fast memory during that phase of computations.
- The set of all elements from B , β_{p_r, j_r} , that are needed for the computations represented by the triples in \mathbf{D} are indexed with the tuples $\mathbf{B}_{\mathbf{D}} = \{(p_r, j_r) \mid R \leq r < R + F\}$. If we think of the triples (i_r, j_r, p_r) as points in 3D, then $\mathbf{B}_{\mathbf{D}}$ is the projection of those points onto the p, j plane. Its size, $|\mathbf{B}_{\mathbf{D}}|$, tells us how many elements of B must at some point be in fast memory during that phase of computations.

Now, there is an result known as the *discrete Loomis-Whitney inequality* that tells us that in our situation $|\mathbf{D}| \leq \sqrt{|\mathbf{C}_{\mathbf{D}}||\mathbf{A}_{\mathbf{D}}||\mathbf{B}_{\mathbf{D}}|}$. In other words, $F_{\max} \leq \sqrt{|\mathbf{C}_{\mathbf{D}}||\mathbf{A}_{\mathbf{D}}||\mathbf{B}_{\mathbf{D}}|}$. The name of the game

¹E.g., S elements from A , $M/3$ elements of B , and $2M/3$ elements of C .

²Strictly speaking, it is a bit more complicated than splitting the range of the iterations, since the last FMA may require anywhere from 0 to 3 new elements to be loaded from slow memory. Fixing this is a matter of thinking of the loads that are required as separate from the computation (as our model does) and then splitting the operations (loads, FMAs, and stores) into phases rather than the range. This does not change our analysis.

now becomes to find the largest value F_{\max} that satisfies

$$\text{maximize } F_{\max} \text{ such that } \begin{cases} F_{\max} \leq \sqrt{|\mathbf{C}_D||\mathbf{A}_D||\mathbf{B}_D|} \\ |\mathbf{C}_D| > 0, |\mathbf{A}_D| > 0, |\mathbf{B}_D| > 0 \\ |\mathbf{C}_D| + |\mathbf{A}_D| + |\mathbf{B}_D| = S + M. \end{cases}$$

An application known as Lagrange multipliers yields the solution

$$|\mathbf{C}_D| = |\mathbf{A}_D| = |\mathbf{B}_D| = \frac{S+M}{3} \quad \text{and} \quad F_{\max} = \frac{(S+M)\sqrt{S+M}}{3\sqrt{3}}.$$

With that largest F_{\max} we can then establish a lower bound on the number of memory reads given by Equation (2.1):

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

Now, M is a free variable. To come up with the sharpest (best) lower bound, we want the largest lower bound. It turns out that, using techniques from calculus, one can show that $M = 2S$ maximizes the lower bound. Thus, the best lower bound our analysis yields is given by

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

2.5.4 A nearly optimal algorithm

We now discuss a (nearly) optimal algorithm for our simplified architecture. Recall that we assume fast memory can hold S elements. For simplicity, assume S is a perfect square. Partition

$$C = \begin{pmatrix} C_{0,0} & C_{0,1} & \cdots \\ C_{1,0} & C_{1,1} & \cdots \\ \vdots & \vdots & \end{pmatrix}, \quad A = \begin{pmatrix} A_0 \\ A_1 \\ \vdots \end{pmatrix}, \quad \text{and} \quad B = \begin{pmatrix} B_0 & B_1 & \cdots \end{pmatrix}.$$

where $C_{i,j}$ is $m_R \times n_R$, A_i is $m_R \times k$, and B_j is $k \times n_R$. Here we choose $m_R \times n_R = (\sqrt{S} - 1) \times \sqrt{S}$ so that fast memory can hold one submatrix $C_{i,j}$, one column of A_i , and one element of B_j : $m_R \times n_R + m_R + 1 = (\sqrt{S} - 1) \times \sqrt{S} + \sqrt{S} - 1 + 1 = S$.

When computing $C := AB + C$, we recognize that $C_{i,j} := A_i B_j + C_{i,j}$. Now, let's further partition

$$A_i = \begin{pmatrix} a_{i,0} & a_{i,1} & \cdots \end{pmatrix} \quad \text{and} \quad B_j = \begin{pmatrix} b_{0,j}^T \\ b_{1,j}^T \\ \vdots \end{pmatrix}.$$

We now recognize that $C_{i,j} := A_i B_j + C_{i,j}$ can be computed as

$$C_{i,j} := a_{i,0} b_{0,j}^T + a_{i,1} b_{1,j}^T + \cdots,$$

the by now very familiar sequence of rank-1 updates that makes up the microkernel discussed in Section ???. The following loop exposes the computation $C := AB + C$, including the loads and stores from and to slow memory:

```

for  $j := 0, \dots, N - 1$ 
  for  $i := 0, \dots, M - 1$ 
    load  $C_{i,j}$  into fast memory
    for  $p := 0, \dots, k - 1$ 
      load  $a_{i,p}$  and  $b_{p,j}^T$  into fast memory
       $C_{i,j} := a_{i,p}b_{p,j}^T + C_{i,j}$ 
    end
    store  $C_{i,j}$  to slow memory
  end
end

```

For simplicity, here $M = m/m_r$ and $N = n/n_r$.

On the surface, this seems to require $C_{i,j}$, $a_{i,p}$, and $b_{p,j}^T$ to be in fast memory at the same time, placing $m_R \times n_R + m_R + n_r = S + \sqrt{S} - 1$ floats in fast memory. However, we have seen before that the rank-1 update $C_{i,j} := a_{i,p}b_{p,j}^T + C_{i,j}$ can be implemented as a loop around AXPY operations, so that the elements of $b_{p,j}^T$ only need to be in fast memory one at a time.

Let us now analyze the number of memory operations incurred by this algorithm:

- Loading and storing all $C_{i,j}$ incurs mn loads and mn stores, for a total of $2mn$ memory operations. (Each such block is loaded once and stored once, meaning every element of C is loaded once and stored once.)
- Loading all $a_{i,p}$ requires

$$MNkm_R = (Mm_R)Nk = m \frac{n}{n_R} k = \frac{mnk}{\sqrt{S}}$$

memory operations.

- Loading all $b_{p,j}^T$ requires

$$MNkn_R = M(Nn_R)k = \frac{m}{m_R} nk = \frac{mnk}{\sqrt{S} - 1}$$

memory operations.

The total number of memory operations is hence

$$2mn + \frac{mnk}{\sqrt{S}} + \frac{mnk}{\sqrt{S} - 1} = 2\frac{mnk}{\sqrt{S}} + 2mn + \frac{mnk}{S - \sqrt{S}}.$$

We can now compare this to the lower bound from the last unit:

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

The cost of reading and writing elements of C , $2mn$, contributes a lower order term, as does $\frac{mnk}{S-\sqrt{S}}$ if S (the size of fast memory) is reasonably large. Thus, the proposed algorithm is nearly optimal with regards to the amount of data that is moved between slow memory and fast memory.

2.5.5 Discussion

What we notice is that the algorithm presented in the last unit is quite similar to the algorithm that in the end delivered good performance in Section 2.3. Both utilize most of fast memory (registers in Section 2.3) with a submatrix of C . Both organize the computation in terms of a kernel that performs rank-1 updates of that submatrix of C .

What is different is that the theory indicates that the number of memory operations are minimized if the block of C is chosen to be square. In Section 2.3, the best performance was observed with a kernel that chose the submatrix of C in registers to be 12×4 . The reason is that loading elements of A into registers (four at a time) is cheaper (on a per-element basis) than broadcasting elements of B . While in Unit 2.4.4 we try to minimize memory operations, in Section 2.3 we are trying to minimize time spent in those memory operations.

2.6 Enrichment

2.7 Wrapup

Pushing the Limits

3.1 Opener

3.1.1 Launch

The current plan is to discuss the memory hierarchy and how MMM allows data movement to be amortized. Maybe the following can be the opener for Week 3:

The inconvenient truth is that floating point computations can be performed very fast while bringing data in from main memory is relatively slow. How slow? On a typical architecture it takes two orders of magnitude more time to bring a floating point number in from main memory than it takes to compute with it.

The reason why main memory is slow is relatively simple: there is not enough room on a chip for the large memories that we are accustomed to, and hence they are off chip. The mere distance creates a latency for retrieving the data. This could then be offset by retrieving a lot of data simultaneously. Unfortunately there are inherent bandwidth limitations: there are only so many pins that can connect the central processing unit with main memory.

To overcome this limitation, a modern processor has a hierarchy of memories. We have already encountered the two extremes: registers and main memory. In between, there are smaller but faster *cache memories*. These cache memories are on-chip and hence do not carry the same latency as does main memory and also can achieve greater bandwidth. The hierarchical nature of these memories is often depicted as a pyramid as illustrated in Figure 3.1. To put things in perspective: We have discussed that the Haswell processor has sixteen vector processors that can store 64 double precision floating point numbers (doubles). Close to the CPU it has a 32Kbytes level-1 cache (L1 cache) that can thus store 4,096 doubles. Somewhat further it has a 256Kbytes level-2 cache (L2 cache) that can store 32,768 doubles. Further away yet, but still on chip, it has a 8Mbytes level-3 cache (L3 cache) that can hold 524,288 doubles.

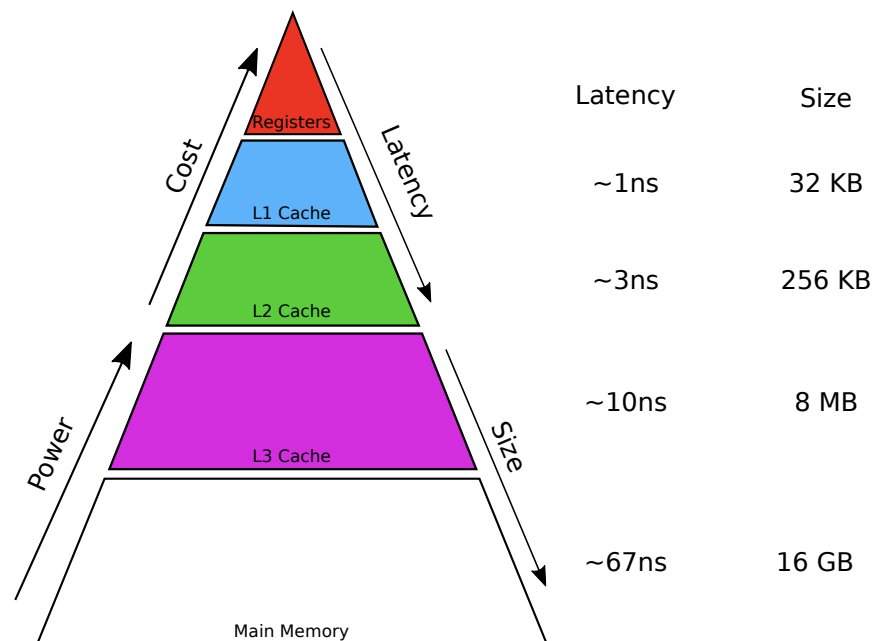


Figure 3.1: Illustration of the memory hierarchy.

In further discussion, we will pretend that one can place data in a specific cache and keep it there for the duration of computations. In fact, caches retain data using some replacement policy that evicts data that has not been recently used. By carefully ordering computations, we can encourage data to remain in cache, which is what happens in practice.

Homework 3.1.1.1 Since we compute with (sub)matrices, it is useful to have some idea of how big of a matrix one can fit in each of the layers of cache. Assume each element in the matrix is a double precision number, which requires 8 bytes. Assume each element in the matrix is a double precision number, which requires 8 bytes.

Layer	Size	Largest $n \times n$ matrix
registers	16×4 doubles	
L1 cache	32 Kbytes	
L2 cache	256 Kbytes	
L3 cache	8 Mbytes	

(The cache sizes given here are for a Intel Haswell processor.)

 [SEE ANSWER](#)

3.1.2 Outline Week 2

3.1. Opener	81
3.1.1. Launch	81
3.1.2. Outline Week 2	84
3.1.3. What you will learn	85
3.2. Leveraging the Caches	86
3.2.1. Adding cache memory into the mix	86
3.2.2. Streaming submatrices of C and B	89
3.2.3. Blocking for multiple caches	93
3.3. Contiguous Memory Access is Still Important	97
3.3.1. Stride matters	97
3.3.2. Packing	98
3.4. Further Tricks of the Trade	102
3.4.1. Avoiding repeated memory allocations	102
3.4.2. Different $m_R \times n_R$ choices	105
3.4.3. Alignment	105
3.4.4. Prefetching	106
3.4.5. Loop unrolling	106
3.4.6. Different m_C, n_C and/or k_C choices	107
3.4.7. Using in-line assembly code	107
3.5. Enrichment	107
3.6. Wrapup	107
3.6.1. Additional exercises	107

3.1.3 What you will learn

3.2 Leveraging the Caches

3.2.1 Adding cache memory into the mix

We now refine our model of the processor slightly, adding one cache memory into the mix.

- Our processor has only one core.
- That core has three levels of memory: registers, a cache memory, and main memory.
- Moving data between the cache and registers takes time $\beta_{C \leftrightarrow R}$ per double while moving it between main memory and the cache takes time $\beta_{M \leftrightarrow C}$
- The registers can hold 64 doubles.
- The cache memory can hold one or more smallish matrices.
- Performing a flop with data in registers takes time γ_R .
- Data movement and computation cannot overlap.

The idea now is to figure out how to block the matrices into submatrices and then compute while these submatrices are in cache to avoid having to access memory more than necessary.

A naive approach partitions C , A , and B into (roughly) square blocks:

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

and

$$B = \left(\begin{array}{c|c|c|c} B_{0,0} & B_{0,1} & \cdots & B_{0,N-1} \\ \hline B_{1,0} & B_{1,1} & \cdots & B_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline B_{K-1,0} & B_{K,1} & \cdots & B_{K-1,N-1} \end{array} \right),$$

where $C_{i,j}$ is $m_C \times n_C$, $A_{i,p}$ is $m_C \times k_C$, and $B_{p,j}$ is $k_C \times n_C$. Then

$$C_{i,j} := \sum_{p=0}^{K-1} A_{i,p} B_{p,j} + C_{i,j},$$

which can be written as the triple-nested loop

```
for i := 0, ..., M - 1
  for j := 0, ..., N - 1
    for p := 0, ..., K - 1
       $C_{i,j} := A_{i,p}B_{p,j} + B_{i,j}$ 
    end
  end
end
```

which is one of $3! = 6$ possible loop orderings.

If we choose m_C , n_C , and k_C such that $C_{i,j}$, $A_{i,p}$, and $B_{p,j}$ all fit in the cache, then we meet our conditions. We can then compute $C_{i,j} := A_{i,p}B_{p,j} + C_{i,j}$ by “bringing these blocks into cache” and computing with them before writing out the result, as before. The difference here is that while one can explicitly load registers, the movement of data into caches is merely encouraged by careful ordering of the computation, since replacement of data in cache is handled by the hardware, which has some cache replacement policy similar to “least recently used” data gets evicted.

Homework 3.2.1.1 For reasons that will become clearer later, we are going to assume that “the cache” in our discussion is the L2 cache, which for the Intel Haswell processor is of size 256Kbytes. If we assume all three (square) matrices fit in that cache during the computation, what size should they be?

In our later discussions 12 becomes a magic number (from the 12×4 micro-kernel) and therefore we should pick the size of the block to be the largest multiple of 12 less than that number. What is it?

Later we will further tune this parameter.

 [SEE ANSWER](#)

Homework 3.2.1.2 In Figure 3.2, we give an IJP loop ordering around the computation of $C_{i,j} := A_{i,p}B_{p,j} + C_{i,j}$, which itself is implemented by `Gemm_JI_4x4Kernel`. It can be found in [Week3/C/Gemm_IJP_JI_4x4Kernel.c](#) and executed by typing

```
make IJP_JI_4x4Kernel
```

in that directory. The resulting performance can be viewed with Live Script [data/Plot_XYZ_JI_MRxNRKernel.mlx](#).

 [SEE ANSWER](#)

```

19 #define MC 96
20 #define NC 96
21 #define KC 96
22
23 void MyGemm( int m, int n, int k, double *A, int ldA,
24             double *B, int ldB, double *C, int ldC )
25 {
26     if ( m % MR != 0 || MC % MR != 0 ){
27         printf( "m and MC must be multiples of MR\n" );
28         exit( 0 );
29     }
30     if ( n % NR != 0 || NC % NR != 0 ){
31         printf( "n and NC must be multiples of NR\n" );
32         exit( 0 );
33     }
34
35     for ( int i=0; i<m; i+=MC ) {
36         int ib = min( MC, m-i );          /* Last block may not be a full block */
37         for ( int j=0; j<n; j+=NC ) {
38             int jb = min( NC, n-j );      /* Last block may not be a full block */
39             for ( int p=0; p<k; p+=KC ) {
40                 int pb = min( KC, k-p );  /* Last block may not be a full block */
41                 Gemm_JI_4x4Kernel
42                     ( ib, jb, pb, &alpha( i,p ), ldA, &beta( p,j ), ldB, &gamma( i,j ), ldC );
43             }
44         }
45     }
46 }
47
48 void Gemm_JI_4x4Kernel( int m, int n, int k, double *A, int ldA,
49                        double *B, int ldB, double *C, int ldC )
50 {
51     for ( int j=0; j<n; j+=NR ) /* n is assumed to be a multiple of NR */
52     for ( int i=0; i<m; i+=MR ) /* m is assumed to be a multiple of MR */
53         Gemm_4x4Kernel( k, &alpha( i,0 ), ldA, &beta( 0,j ), ldB, &gamma( i,j ), ldC );
54 }

```

Assignments/Week3/C/Gemm_IJP_JI_4x4Kernel.c

Figure 3.2: Triple loop around $C_{i,j} := A_{i,p}B_{p,j} + C_{i,j}$.

Homework 3.2.1.3 Copy [Gemm_IJP_JI_4x4Kernel.c](#) into `Gemm_IJP_JI_12x4Kernel.c` and modify it to call your implementation of the 12×4 kernel. Collect performance data by executing

```
make IJP_JI_12x4Kernel
```

in that directory. The resulting performance can be viewed with Live Script [data/Plot_XYZ_JI_MRxNRKernel.mlx](#).

[SEE ANSWER](#)

Homework 3.2.1.4 Create all six loop orderings by copying [Gemm_IJP_JI_12x4Kernel.c](#) into

```
Gemm_IPJ_JI_12x4Kernel.c
Gemm_JIP_JI_12x4Kernel.c
Gemm_JPI_JI_12x4Kernel.c
Gemm_PIJ_JI_12x4Kernel.c
Gemm_PJI_JI_12x4Kernel.c
```

reordering the loops as indicated by the XYZ in `Gemm_XYZ_JI_12x4Kernel.c`. Collect performance data by executing

```
make XYZ_JI_12x4Kernel
```

for $XYZ \in \{IPJ, PIJ, PJI, PJI, PJI\}$.

(If you don't want to do them all, implement at least `Gemm_PIJ_JI_12x4Kernel.c`.) The resulting performance can be viewed with Live Script [data/Plot_XYZ_JI_MRxNRKernel.mlx](#).

[SEE ANSWER](#)

The performance attained by our implementation is shown in Figure ?? (top-right).

3.2.2 Streaming submatrices of C and B

We illustrate the execution of `Gemm_JI_4x4Kernel` with one set of three submatrices $C_{i,j}$, $A_{i,p}$, and $B_{p,j}$ in Figure 3.3 for the case where $m_R = n_R = 4$ and $m_C = n_C = k_C = 4 \times m_R$. What we notice is that each the $m_R \times n_R$ submatrices of $C_{i,j}$ is not reused again once it has been updated. This means that at any given time only one such matrix needs to be in the cache memory (as well as registers). Similarly, a panel of $B_{p,j}$ is not reused and hence only one such panel needs to be in cache. It is submatrix $A_{i,p}$ that must remain in cache during the entire computation. By not keeping the entire submatrices $C_{i,j}$ and $B_{p,j}$ in the cache memory, the size of the submatrix $A_{i,p}$ can be increased, which can be expected to improve performance. We will refer to this as “streaming” “micro-blocks” of $C_{i,j}$ and “micro-panels” of $B_{p,j}$, while keeping all of block $A_{i,p}$ in cache.

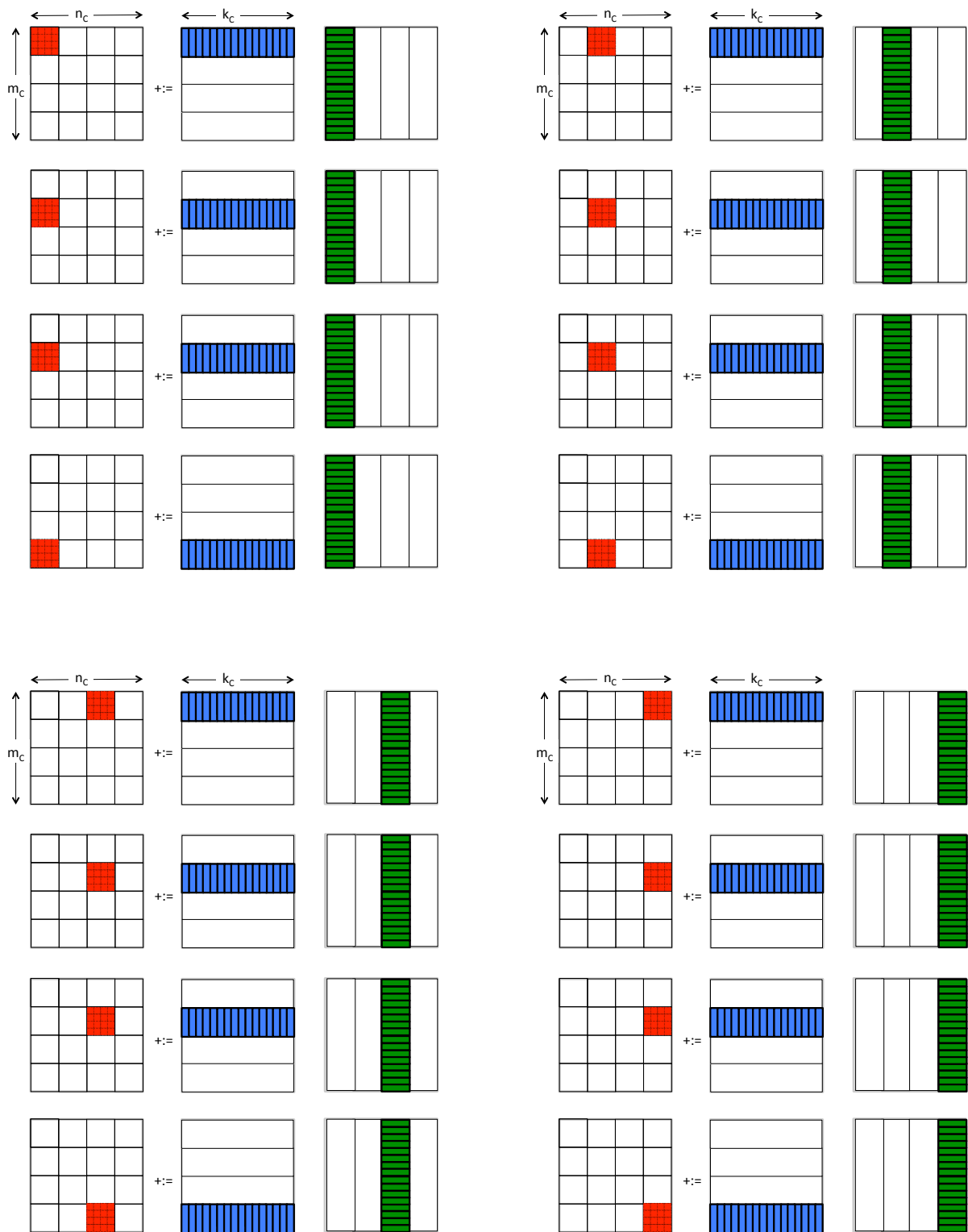



Figure 3.3: Illustration of computation in the cache with one block from each of A , B , and C .

We could have chosen a different order of the loops, and then we may have concluded that submatrix $B_{p,j}$ needs to stay in cache while streaming a micro-blocks of C and small panels of A . Obviously, there is a symmetry here and hence we will just focus on the case where $A_{i,p}$ is kept in cache.

The second observation is that now that $C_{i,j}$ and $B_{p,j}$ are being streamed, there is no need for those submatrices to be square. Furthermore, for `Gemm_PIJ_JI_12x4Kernel.c` and `Gemm_IPJ_JI_12x4Kernel.c` the larger n_C , the more effectively the cost of bringing $A_{i,p}$ into cache is amortized over computation: we should pick $n_C = n$. (Later, we will reintroduce n_C .) Another way of think of this is that if we choose the PIJ loop around the JI ordering around the micro-kernel (in other words, if we consider the `Gemm_PIJ_JI_12x4Kernel` implementation), then we notice that the inner loop of PIJ (indexed with J) matches the outer loop of the JI double loop, and hence those two loops can be combined, leaving us with an implementation that is a double loop (PI) around the double loop JI.

Homework 3.2.2.1 Copy  `Gemm_PIJ_JI_12x4Kernel.c` into `Gemm_PI_JI_12x4Kernel.c` and remove the loop indexed with j . Collect performance data by executing

```
make PI_JI_12x4Kernel
```

in that directory. The resulting performance can be viewed with Live Script  [data/Plot_XY_JI_MRxNRKernel.mlx](#).

 [SEE ANSWER](#)

Homework 3.2.2.2 Execute

```
make PI_JI_MCxKC
```

and view the performance results with Live Script

 [data/Plot_MC_KC_Performance.mlx](#).

This experiment tries many different choices for m_C and k_C , and presents them as a scatter graph so that the optimal choice can be determined and visualized. (It will take a long time to execute. Go take a nap!)

 [SEE ANSWER](#)

The result of this last homework on Robert's laptop is shown in Figure 3.4.

To summarize, our insights suggest

1. Bring an $m_C \times k_C$ submatrix of A into the cache, at a cost of $m_C \times k_C$ memops. (It is the order of the loops and instructions that encourages the submatrix of A to stay in cache.) This cost is amortized over $2m_C n_C k_C$ flops for a ratio of $2m_C n_C k_C / (m_C k_C) = 2n$. The larger n , the better.
2. The cost of reading an $k_C \times n_R$ submatrix of B is amortized over $2m_C n_R k_C$ flops, for a ratio of $2m_C n_R k_C / (2m_C n_R) = m_C$. Obviously, the greater m_C , the better.

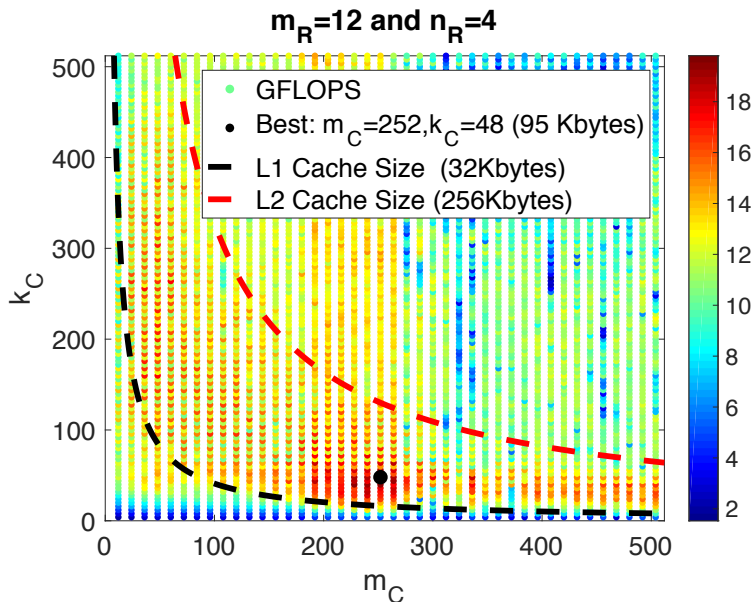


Figure 3.4: Performance when $m_R = 12$ and $n_R = 4$ and m_C and k_C are varied. The best performance is empirically determined by searching the results. We notice that \tilde{A} is sized to fit in the L2 cache. As we apply additional optimizations, the optimal choice for m_C and k_C will change. Notice that on Robert's laptop, the optimal choice varies greatly from one experiment to the next. You may observe the same level of variability.

3. The cost of reading and writing an $m_R \times n_R$ submatrix of C is now amortized over $2m_R n_R k_C$ flops, for a ratio of $2m_R n_R k_C / (2m_R n_R) = k_C$. Obviously, the greater k_C , the better.

Items 2 and 3 suggest that $m_C \times k_C$ submatrix $A_{i,p}$ be roughly square. (This comes with a caveat. See video!)

If we revisit the performance data plotted in Figure 3.4, we notice that matrix $A_{i,p}$ fits in part of the L2 cache, but is too big for the L1 cache. What this means is that the bandwidth between the L2 and registers is enough to allow these blocks of A to reside in the L2 cache. Since the L2 cache is larger than the L1 cache, this benefits performance, since the m_C and k_C can be larger.

In our future discussions, we will use the following terminology:

- A $m_R \times n_R$ submatrix of C that is begin updated we will call a *micro-tile*.
- The $m_R \times k_C$ submatrix of A and $k_C \times n_R$ submatrix of B we will call *micro-panels*.
- The routine that updates a micro-tile by multiplying two micro-panels we will call the *micro-kernel*.

3.2.3 Blocking for multiple caches

The last section motivated Figure 3.5, which describes one way for blocking for multiple levels of cache. While some details still remain, this brings us very close to how matrix-matrix multiplication is implemented in practice in libraries that are widely used. The approach illustrated there was first suggested by Kazushige Goto [?] and is referred to as the Goto approach (to matrix-matrix multiplication) or GotoBLAS approach, since he incorporated it in an implementation of the Basic Linear Algebra Subprograms (BLAS) by that name.

Each nested box represents a single loop that partitions one of the three dimensions (m , n , or k). The submatrices $C_{i,j}$, $A_{i,p}$, and $B_{p,j}$ are those already discussed in Section 3.2.1.

- The roughly square matrix $A_{i,p}$ is “placed” in the L2 cache. If data from $A_{i,p}$ can be fetched from the L2 cache fast enough, it is better to place it there since there is an advantage to making m_C and k_C larger and the L2 cache is larger than the L1 cache.
- Our analysis in Section 3.2.1 also suggests that k_C be chosen to be large. Since the submatrix $B_{p,j}$ is reused multiple times for many iterations of the “fourth loop around the micro-kernel” it may be beneficial to choose n_C so that $k_C \times n_C$ submatrix $B_{p,j}$ stays in the L3 cache. Later we will see there are other benefits to this.
- In this scheme, the $m_R \times n_R$ micro-block of $C_{i,j}$ end up in registers and the $k_C \times n_R$ “micro-panel” of $B_{p,j}$ ends up in the L1 cache.

Data in the L1 cache typically are also in the L2 and L3 caches. The update of the $m_R \times n_R$ micro-block of C also brings that in to the various caches. Thus, the described situation is similar to what we described in Section 3.2.1, where “the cache” discussed there corresponds to the L2 cache here. As our discussion proceeds, in Week 3, we will try to make this all more precise.

Homework 3.2.3.1 Which of the implementations

- `Gemm_IJP_JI_12x4Kernel.c`
- `Gemm_JPI_JI_12x4Kernel.c`
- `Gemm_PJI_JI_12x4Kernel.c`

best captures the loop structure illustrated in Figure 3.5?

 [SEE ANSWER](#)

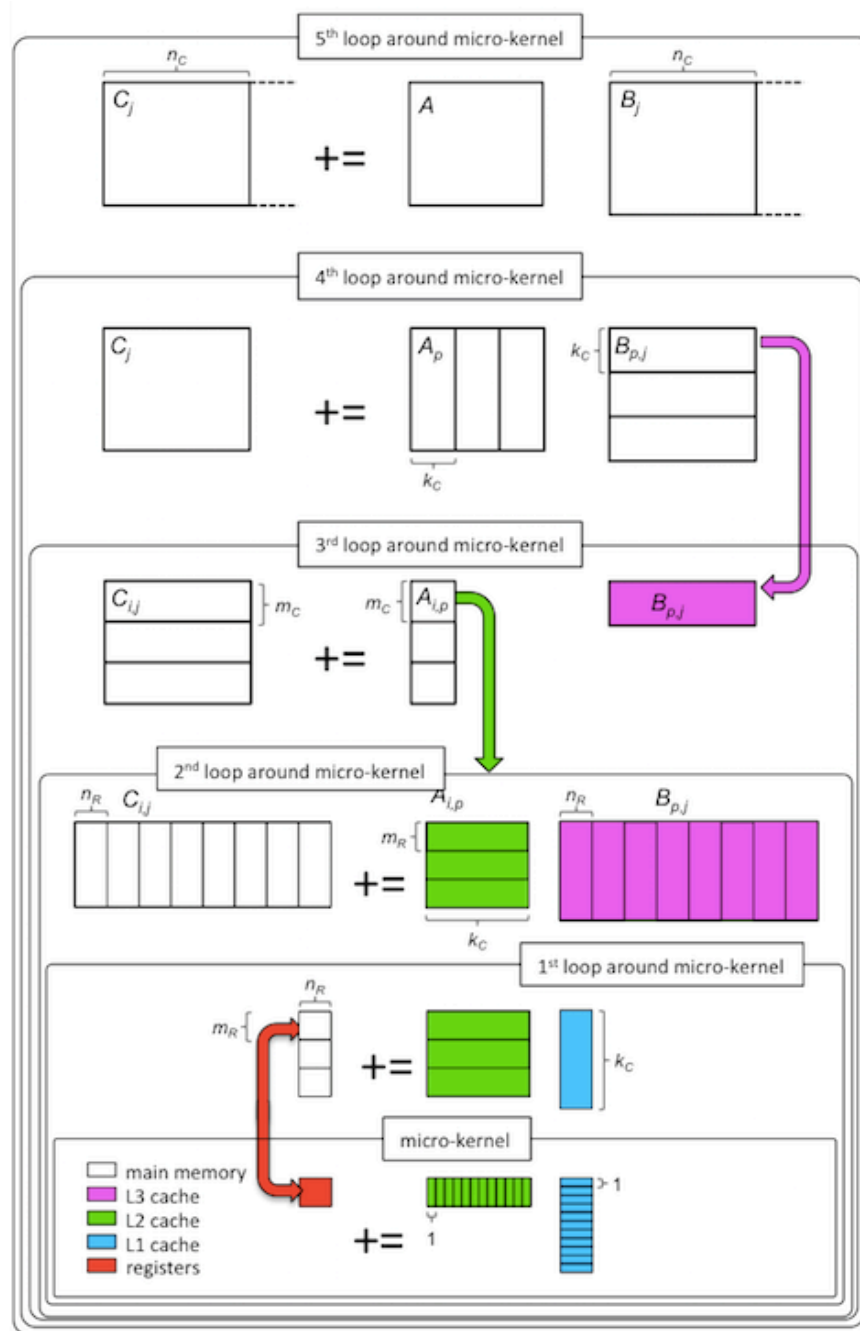


Figure 3.5: Blocking for multiple levels of cache. Figure adapted from [?].

```

39     double *B, int ldB, double *C, int ldC )
40 {
41     for ( int j=0; j<n; j+=NC ) {
42         int jb = min( NC, n-j );    /* Last loop may not involve a full block */
43         LoopFour( m, jb, k, A, ldA, &beta( , ), ldB, &gamma( , ), ldC );
44     }
45 }
46
47 void LoopFour( int m, int n, int k, double *A, int ldA,
48               double *B, int ldB, double *C, int ldC )
49 {
50     for ( int p=0; p<k; p+=KC ) {
51         int pb = min( KC, k-p );    /* Last loop may not involve a full block */
52         LoopThree( m, n, pb, &alpha( , ), ldA, &beta( , ), ldB, C, ldC );
53     }
54 }
55
56 void LoopThree( int m, int n, int k, double *A, int ldA,
57                double *B, int ldB, double *C, int ldC )
58 {
59     for ( int i=0; i<m; i+=MC ) {
60         int ib = min( MC, m-i );    /* Last loop may not involve a full block */
61         LoopTwo( ib, n, k, &alpha( , ), ldA, B, ldB, &gamma( , ), ldC );
62     }
63 }
64
65 void LoopTwo( int m, int n, int k, double *A, int ldA,
66              double *B, int ldB, double *C, int ldC )
67 {
68     for ( int j=0; j<n; j+=NR ) {
69         int jb = min( NR, n-j );
70         LoopOne( m, jb, k, A, ldA, &beta( , ), ldB, &gamma( , ), ldC );
71     }
72 }
73
74 void LoopOne( int m, int n, int k, double *A, int ldA,
75              double *B, int ldB, double *C, int ldC )
76 {
77     for ( int i=0; i<m; i+=MR ) {
78         int ib = min( MR, m-i );
79         Gemm_4x4Kernel( k, &alpha( , ), ldA, B, ldB, &gamma( , ), ldC );
80     }
81 }

```

Assignments/Week3/C/Gemm_Five_Loops_4x4Kernel.c

Figure 3.6: Blocking for multiple levels of cache.

Homework 3.2.3.2 We always advocate, when it does not substantially impede performance, to instantiate an algorithm in code in a way that closely resembles how one explains it. In this spirit, the algorithm described in Figure 3.5 can be coded by making each loop (box) in the figure a separate routine. An outline of how this might be accomplished is given in Figure 3.6 and file [Gemm_Five_Loops_4x4Kernel.c](#). Complete the code and execute it with

```
make Five_Loops_4x4Kernel
```

Then, view the performance with Live Script [data/Plot_Five_Loops.mlx](#).

[SEE ANSWER](#)

Homework 3.2.3.3 Copy [Gemm_Five_Loops_4x4Kernel.c](#) into `Gemm_Five_Loops_12x4Kernel.c` and modify it to use the “ 12×4 ” kernel. Complete the code and execute it with

```
make Five_Loops_12x4Kernel
```

Then, view the performance with Live Script [data/Plot_Five_Loops.mlx](#).

(To do a fair comparison, you may want to look at the block sizes chosen for `Gemm_Five_Loops_12x4Kernel.c` and use those in other implementations as well.)

[SEE ANSWER](#)

3.3 Contiguous Memory Access is Still Important

3.3.1 Stride matters

Homework 3.3.1.4 One would think that once one introduces blocking, the higher performance observed for small matrix sizes will be maintained as the problem size increases. This is not what you observed: there is still a steady decrease in performance. To investigate what is going on, copy the driver routine in `driver.c` into `driver_ldim.c` and in that new file change the line


```
ldA = ldB = ldC = size;
```

to

```
ldA = ldB = ldC = last;
```

This sets the leading dimension of matrices *A*, *B*, and *C* to the largest problem size to be tested, for all experiments. Collect performance data by executing

```
make FiveLoops_ldim
```

in that directory. The resulting performance can be viewed with Live Script  [data/Plot_FiveLoops.mlx](#).

 [SEE ANSWER](#)

What is going on here? Modern architectures like the Haswell processor have a feature called “hardware prefetching” that speculatively preloads data into caches based on observed access patterns so that this loading is hopefully overlapped with computation. Processors also organize memory in terms of pages, for reasons that go beyond the scope of this course. For our purposes, a page is a contiguous block of memory of 4 Kbytes (4096 bytes or 512 doubles). Prefetching only occurs within a page. Since the leading dimension is now large, when the computation moves from column to column in a matrix data is not prefetched. You may want to learn more on memory pages by visiting [https://en.wikipedia.org/wiki/Page_\(computer_memory\)](https://en.wikipedia.org/wiki/Page_(computer_memory)) on Wikipedia.

The bottom line: early in the course we discussed that marching contiguously through memory is a good thing. While we claim that we block for the caches, in practice the data is not contiguous and hence we can’t really control how the data is brought into caches and where it exists at a particular time in the computation.

3.3.2 Packing

Homework 3.3.2.1 In the setting of the five loops, the micro-kernel updates $C := AB + C$ where C is $m_R \times n_R$, A is $m_R \times k_C$, and B is $k_C \times n_R$. How many matrix elements of C have to be brought into registers to perform this computation? How many flops are performed with these elements? How many flops are performed with each element of C ? We seem to have settled on a $m_R \times n_R = 12 \times 4$ microtile. Let's use $k_C = 128$. What is the ratio for these parameters?

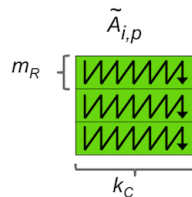
[SEE ANSWER](#)

The next step towards optimizing the micro-kernel recognizes that computing with contiguous data (accessing data with a “stride one” access pattern) improves performance. The fact that the $m_R \times n_R$ micro-tile of C is not in contiguous memory is not particularly important. The cost of bringing it into the vector registers from some layer in the memory is mostly inconsequential because a lot of computation is performed before it is written back out. It is the repeated accessing of the elements of A and B that can benefit from stride one access.

Two successive rank-1 updates of the micro-tile can be given by

$$\begin{pmatrix} \gamma_{0,0} & \gamma_{0,1} & \gamma_{0,2} & \gamma_{0,3} \\ \gamma_{1,0} & \gamma_{1,1} & \gamma_{1,2} & \gamma_{1,3} \\ \gamma_{2,0} & \gamma_{2,1} & \gamma_{2,2} & \gamma_{2,3} \\ \gamma_{3,0} & \gamma_{3,1} & \gamma_{3,2} & \gamma_{3,3} \end{pmatrix} + := \begin{pmatrix} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{pmatrix} \begin{pmatrix} \beta_{p,0} & \beta_{p,1} & \beta_{p,2} & \beta_{p,3} \end{pmatrix} + \begin{pmatrix} \alpha_{0,p+1} \\ \alpha_{1,p+1} \\ \alpha_{2,p+1} \\ \alpha_{3,p+1} \end{pmatrix} \begin{pmatrix} \beta_{p+1,0} & \beta_{p+1,1} & \beta_{p+1,2} & \beta_{p+1,3} \end{pmatrix}.$$

Since A and B are stored with column-major order, the four elements of $\alpha_{[0:3],p}$ are contiguous in memory, but they are (generally) not contiguously stored with $\alpha_{[0:3],p+1}$. Elements $\beta_{p,[0:3]}$ are (generally) also not contiguous. The access pattern during the computation by the micro-kernel would be much more favorable if the A involved in the micro-kernel was packed in column-major order with leading dimension m_R :



and B was packed in row-major order with leading dimension n_R :

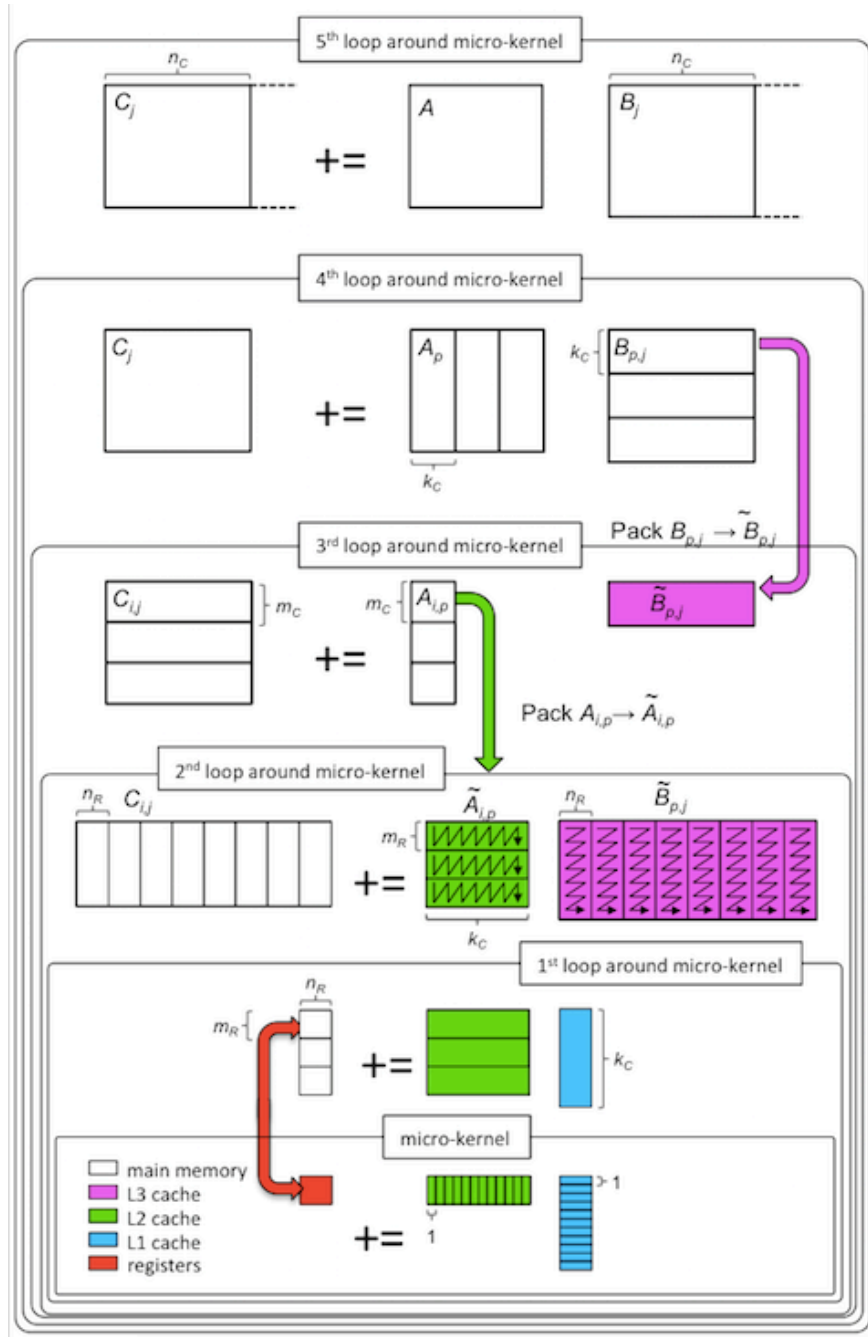


Figure 3.7: Blocking for multiple levels of cache, with packing. Picture adapted from []

```
void PackMicroPanelA_MRxC( int m, int k, double *A, int ldA, double *Atilde )
/* Pack a micro-panel of A into buffer pointed to by Atilde.
   This is an unoptimized implementation for general MR and KC. */
{
    /* March through A in column-major order, packing into Atilde as we go. */

    if ( m == MR ) /* Full row size micro-panel.*/
        for ( int p=0; p<k; p++ )
            for ( int i=0; i<MR; i++ )
                *Atilde++ = alpha( i, p );
    else /* Not a full row size micro-panel. We pad with zeroes. */
        for ( int p=0; p<k; p++ ) {
            for ( int i=0; i<m; i++ )
                *Atilde++ = alpha( i, p );
            for ( int i=m; i<MR; i++ )
                *Atilde++ = 0.0;
        }
}

void PackBlockA_MCxC( int m, int k, double *A, int ldA, double *Atilde )
/* Pack a MC x KC block of A. MC is assumed to be a multiple of MR. The block is packed
   into Atilde a micro-panel at a time. If necessary, the micro-panel is padded with rows
   of zeroes. */
{
    for ( int i=0; i<m; i+=MR ){
        int ib = min( MR, m-i );

        PackMicroPanelA_MRxC( ib, k, &alpha( i, 0 ), ldA, Atilde );
        Atilde += MR * k;
    }
}
```

Figure 3.8: A reference implementation of routines for packing $A_{i,p}$.


```

void PackMicroPanelB_KCNR( int k, int n, double *B, int ldB, double *Btilde )
/* Pack a micro-panel of B into buffer pointed to by Btilde.
   This is an unoptimized implementation for general KC and NR. */
{
    /* March through B in row-major order, packing into Btilde as we go. */

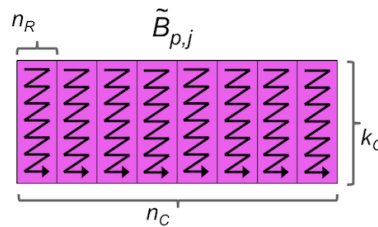
    if ( n == NR ) /* Full column width micro-panel.*/
        for ( int p=0; p<k; p++ )
            for ( int j=0; j<NR; j++ )
                *Btilde++ = beta( p, j );
    else /* Not a full row size micro-panel. We pad with zeroes. */
        for ( int p=0; p<k; p++ ) {
            for ( int j=0; j<n; j++ )
                *Btilde++ = beta( p, j );
            for ( int j=n; j<NR; j++ )
                *Btilde++ = 0.0;
        }
}

void PackPanelB_KCNC( int k, int n, double *B, int ldB, double *Btilde )
/* Pack a KC x NC panel of B. NC is assumed to be a multiple of NR. The
   block is packed into Btilde a micro-panel at a time. If necessary, the
   micro-panel is padded with columns of zeroes. */
{
    for ( int j=0; j<n; j+= NR ){
        int jb = min( NR, n-j );

        PackMicroPanelB_KCNR( k, jb, &beta( 0, j ), ldB, Btilde );
        Btilde += k * jb;
    }
}

```

Figure 3.9: A reference implementation of routines for packing $B_{p,j}$.



If this packing were performed at a strategic point in the computation, so that the packing is amortized over many computations, then a benefit might result. These observations are captured in Figure 3.7 and translated into an implementation in Figure 3.10. Reference implementations of packing routines can be found in Figures 3.8 and 3.9. While these implementations can be optimized, the fact is that the cost when packing is in the data movement between main memory and faster memory. As a result, optimizing the packing has relatively little effect.

How to modify the five loops to incorporate packing is illustrated in Figure 3.10. A micro-kernel to compute with the packed data when $m_R \times n_R = 4 \times 4$ is illustrated in Figure 3.11.

Homework 3.3.2.2 Examine the file `Gemm_Five_Loops_Pack_4x4Kernel.c`. Collect performance data with

```
make Five_Loops_Pack_4x4Kernel
```

and view the resulting performance with Live Script `Plot_Five_Loops.mlx`.

[SEE ANSWER](#)

Homework 3.3.2.3 Copy the file `Gemm_Five_Loops_Pack_4x4Kernel.c` into file `Gemm_Five_Loops_Pack_12x4Kernel.c`. Modify that file so that it uses $m_R \times n_R = 12 \times 4$. Test the result with

```
make Five_Loops_Pack_12x4Kernel
```

and view the resulting performance with Live Script `Plot_Five_Loops.mlx`.

[SEE ANSWER](#)

3.4 Further Tricks of the Trade

We now give a laundry list of further optimizations that can be tried.

3.4.1 Avoiding repeated memory allocations

In the final implementation that incorporates packing, at various levels workspace is allocated and deallocated multiple times. This can be avoided by allocating workspace immediately upon entering `LoopFive`.

```

40 void LoopFive( int m, int n, int k, double *A, int ldA,
41               double *B, int ldB, double *C, int ldC )
42 {
43     for ( int j=0; j<n; j+=NC ) {
44         int jb = min( NC, n-j );    /* Last loop may not involve a full block */
45         LoopFour( m, jb, k, A, ldA, &beta( 0,j ), ldB, &gamma( 0,j ), ldC );
46     }
47 }
48
49 void LoopFour( int m, int n, int k, double *A, int ldA, double *B, int ldB,
50               double *C, int ldC )
51 {
52     double *Btilde = ( double * ) malloc( KC * NC * sizeof( double ) );
53
54     for ( int p=0; p<k; p+=KC ) {
55         int pb = min( KC, k-p );    /* Last loop may not involve a full block */
56         PackPanelB_KCxNC( pb, n, &beta( p, 0 ), ldB, Btilde );
57         LoopThree( m, n, pb, &alpha( 0, p ), ldA, Btilde, C, ldC );
58     }
59
60     free( Btilde );
61 }
62
63 void LoopThree( int m, int n, int k, double *A, int ldA, double *Btilde, double *C, int ldC )
64 {
65     double *Atilde = ( double * ) malloc( MC * KC * sizeof( double ) );
66
67     for ( int i=0; i<m; i+=MC ) {
68         int ib = min( MC, m-i );    /* Last loop may not involve a full block */
69         PackBlockA_MCxCxKC( ib, k, &alpha( i, 0 ), ldA, Atilde );
70         LoopTwo( ib, n, k, Atilde, Btilde, &gamma( i,0 ), ldC );
71     }
72
73     free( Atilde );
74 }
75
76 void LoopTwo( int m, int n, int k, double *Atilde, double *Btilde, double *C, int ldC )
77 {
78     for ( int j=0; j<n; j+=NR ) {
79         int jb = min( NR, n-j );
80         LoopOne( m, jb, k, Atilde, &Btilde[ j*k ], &gamma( 0,j ), ldC );
81     }
82 }
83
84 void LoopOne( int m, int n, int k, double *Atilde, double *MicroPanelB, double *C, int ldC )
85 {
86     for ( int i=0; i<m; i+=MR ) {
87         int ib = min( MR, m-i );
88         Gemm_4x4Kernel_Packed( k, &Atilde[ i*k ], MicroPanelB, &gamma( i,0 ), ldC );
89     }
90 }

```

Assignments/Week3/C/Gemm.Five.Loops.Pack_4x4Kernel.c

Figure 3.10: Blocking for multiple levels of cache, with packing.

```
157 void Gemm_4x4Kernel_Packed( int k,  
158     double *BlockA, double *PanelB, double *C, int ldC )  
159 {  
160     __m256d gamma_0123_0 = _mm256_loadu_pd( &gamma( 0,0 ) );  
161     __m256d gamma_0123_1 = _mm256_loadu_pd( &gamma( 0,1 ) );  
162     __m256d gamma_0123_2 = _mm256_loadu_pd( &gamma( 0,2 ) );  
163     __m256d gamma_0123_3 = _mm256_loadu_pd( &gamma( 0,3 ) );  
164  
165     __m256d beta_p_j;  
166  
167     for ( int p=0; p<k; p++ ){  
168         /* load alpha( 0:3, p ) */  
169         __m256d alpha_0123_p = _mm256_loadu_pd( BlockA );  
170  
171         /* load beta( p, 0 ); update gamma( 0:3, 0 ) */  
172         beta_p_j = _mm256_broadcast_sd( PanelB );  
173         gamma_0123_0 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_0 );  
174  
175         /* load beta( p, 1 ); update gamma( 0:3, 1 ) */  
176         beta_p_j = _mm256_broadcast_sd( PanelB+1 );  
177         gamma_0123_1 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_1 );  
178  
179         /* load beta( p, 2 ); update gamma( 0:3, 2 ) */  
180         beta_p_j = _mm256_broadcast_sd( PanelB+2 );  
181         gamma_0123_2 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_2 );  
182  
183         /* load beta( p, 3 ); update gamma( 0:3, 3 ) */  
184         beta_p_j = _mm256_broadcast_sd( PanelB+3 );  
185         gamma_0123_3 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_3 );  
186  
187         BlockA += MR;  
188         PanelB += NR;  
189     }  
190  
191     /* Store the updated results. This should be done more carefully since  
192        there may be an incomplete micro-tile. */  
193     _mm256_storeu_pd( &gamma(0,0), gamma_0123_0 );  
194     _mm256_storeu_pd( &gamma(0,1), gamma_0123_1 );  
195     _mm256_storeu_pd( &gamma(0,2), gamma_0123_2 );  
196     _mm256_storeu_pd( &gamma(0,3), gamma_0123_3 );  
197 }
```

Assignments/Week3/C/Gemm_Five_Loops_Pack_4x4Kernel.c

Figure 3.11: Blocking for multiple levels of cache, with packing.

3.4.2 Different $m_R \times n_R$ choices

As mentioned, there are many choices for $m_R \times n_R$. On the Haswell architecture, we focused on 12×4 . There is some evidence that 8×6 works better.

Homework 3.4.2.1 Redo the last few exercises to create `Gemm_Five_Loops_8x6Kernel.c`, `Gemm_Five_Loops_Pack_8x6Kernel.c`. Modify the Makefile and Live Script `Plot_Five_Loops.mlx` as appropriate.

 [SEE ANSWER](#)

We have focused on choices for $m_R \times n_R$ where m_R is a multiple of four. Before packing was added, this was because one loads into registers from contiguous memory and for this reason loading from A , four elements at a time, while broadcasting elements from B was natural. After packing was introduced, one could also contemplate loading from B and broadcasting elements of A , which then also makes kernels like 6×8 possible. The problem is that this would mean performing a vector instruction with elements that are in rows of C , and hence when loading elements of C one would need to perform a transpose operation, and also when storing the elements back to memory. While in practice we have observed that a 6×8 kernel edges out a 8×6 kernel, we leave this optimization to those who are particularly industrious.

However, there is a trick so that you don't have to change the micro-kernel at all. If

$$C := AB + C$$

then

$$C^T := (AB)^T + C^T = B^T A^T + C^T.$$

Now, if a matrix X is stored by columns, then matrix X^T is stored by rows. Now, $C^T := B^T A^T + C^T$ is equivalent to, assuming C , A , and B are stored with column-major order, computing $C := BA + C$ assuming that C , A , and B are stored by rows. If one now uses a 6×8 kernel where A is broadcast and B is loaded, then the result of the micro-kernel computation can naturally update the appropriate elements of C . The only trick now is that when one packs blocks of $B_{i,p}$ into $\tilde{B}_{i,p}$ (targeting the L2 cache) and row panels of $A_{p,j}$ into $\tilde{A}_{p,j}$ (targeting the L3 cache), the packing routines have to take into account that these matrices are stored by rows.

3.4.3 Alignment

There are alternative vector intrinsic routines that load vector registers under the assumption that the data being loaded is “32 byte aligned.” This means that the address that specifies the four doubles being loaded is a multiple of 32.

Conveniently, loads of elements of A and B are from buffers into which the data was packed. By creating that buffer to be 32 byte aligned, we can ensure that all these loads are 32 byte aligned. Intel's intrinsic library has a special memory allocation routine specifically for this purpose.

3.4.4 Prefetching

Elements of $A_{i,p}$ in theory remain in the L2 cache. Loading them into registers may be faster if elements of a next micro-panel of A are prefetched into the L1 cache while computing with the current micro-panel, overlapping that movement with computation. There are intrinsic instructions for that. Some choices of m_R and n_R lend themselves better to exploiting this.

Prefetching (or, rather, preloading) into registers is not a simple matter: The $m_r \times n_r = 12 \times 4$ kernel already uses all sixteen vector registers. The fundamental tools for preloading are the prefetch instructions:

```
// prefetch cache line that includes the address pnttr to L1 cache
_mm_prefetch( pnttr, 0 )
```

```
// prefetch cache line that includes the address pnttr to L2 cache
_mm_prefetch( pnttr, 1 )
```

```
// prefetch cache line that includes the address pnttr to L2 or L3 cache
_mm_prefetch( pnttr, 2 )
```

It is important to replace the allocation routines `malloc` and `free` with `_mm_malloc` and `_mm_free`, respectively, to ensure that data is properly aligned in memory.

Some things to try. You probably want to do this for one core first, before you move on to multiple cores.

- Before computing the update of the $m_r \times n_r$ micro-tile with the micro-kernel, prefetch the next micro-tile to the L3 cache.
- As you compute with the current micro-panel of \tilde{A} , prefetch the next micro-panel of \tilde{A} into the appropriate cache level.
- As you compute with the current micro-panel of \tilde{B} , prefetch the next micro-panel of \tilde{B} into the appropriate cache level. Notice that for the entire execution of Loop 1, the same micro-panel of \tilde{B} is used. So, you want to be selective as to what you prefetch.

3.4.5 Loop unrolling

There is some overhead that comes from having a loop in the micro-kernel. That overhead is in part due to loop indexing and branching overhead. The loop also can get in the way of the explicitly reordering of operations towards some benefit. So, one could turn the loop into a long sequence of instructions. A compromise is to “unroll” it more modestly, decreasing the number of iterations while increasing the number of operations performed in each iteration. Our experience is that unrolling, in combination with using prefetching, confuses the compiler. We have observed that the compiler moves all the the loads to the top of the loop, which then counters the benefits of the prefetching.

3.4.6 Different m_C , n_C and/or k_C choices

There may be a benefit to playing around with the various blocking parameters. Better yet, you may be able to compute what the best choices are. A good paper to read regarding this is

☛ **Analytical Modeling is Enough for High Performance BLIS**

Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti
ACM Transactions on Mathematical Software, Volume 43 Issue 2, September 2016

If you don't have access to ACM's publications, you can read an **early draft**.

3.4.7 Using in-line assembly code

Even more control over where what happens from using in-lined assembly code. This will allow prefetching and how registers are used to be made more explicit. (The compiler often changes the order of the operations with intrinsics are used in C.)

3.5 Enrichment

3.6 Wrapup

3.6.1 Additional exercises

Repeat for single precision!

Repeat for another architecture!

Repeat for other MMM-like operations!

Placing a block of B in the L2 cache

There is a symmetric way of blocking the various matrices that then places a block of B in the L2 cache. This has the effect of accessing C by rows, if C were accessed by columns before, and vice versa. For some choices of $m_R \times n_R$, this may show a benefit.

Week 4

Multithreaded Parallelism

4.1 Opener

4.1.1 Launch

4.1.2 Outline Week 2

4.1. Opener	109
4.1.1. Launch	109
4.1.2. Outline Week 2	110
4.1.3. What you will learn	111
4.2. OpenMP	112
4.2.1. Of cores and threads	112
4.2.2. Basics	112
4.2.3. Hello World!	113
4.3. Multithreading Matrix-Matrix Multiplication	115
4.3.1. Parallelizing the second loop around the micro-kernel	115
4.3.2. Parallelizing the first loop around the micro-kernel	117
4.3.3. Parallelizing the third loop around the micro-kernel	118
4.4. Parallelizing More	120
4.4.1. Speedup, efficiency, etc.	120
4.4.2. Ahmdahl's law	121
4.4.3. Parallelizing the packing	122
4.5. Enrichment	123
4.6. Wrapup	123

4.1.3 What you will learn

4.2 OpenMP

4.2.1 Of cores and threads

From Wikipedia:

“A multi-core processor is a single computing component with two or more independent processing units called cores, which read and execute program instructions.”

The idea is: What will give you your solution faster than using one core? Use multiple cores!

A thread is a *thread of execution*.

- It is an stream of program instructions.
- All these instructions may execute on a single core, multiple cores, or they may move from core to core.
- Multiple threads can execute on different cores or on the same core.

4.2.2 Basics

A good place to read up on OpenMP is again, where else, Wikipedia: <https://en.wikipedia.org/wiki/OpenMP>

OpenMP is a standardized API (Application Programming Interface) for creating multiple threads of execution from a single program. It consists of

- Directives to the compiler (pragmas):

```
#pragma omp parallel
```

- A library of routines:

```
omp_get_max_threads()  
omp_get_num_threads()
```

that can, for example, be used to inquire about the execution environment.

- Environment parameters that can be set

```
$ export OMP_NUM_THREADS=4
```

4.2.3 Hello World!

We will illustrate some of the basics of OpenMP via the old standby, the "Hello World!" program.

Homework 4.2.3.1 In Week4/C/ look at the contents of file HelloWorld.c. Compile it with the command

```
gcc -o HelloWorld.x HelloWorld.c
```

and execute it with

```
./HelloWorld.x
```

 [SEE ANSWER](#)

Homework 4.2.3.2 Copy the file HelloWorld.c to HelloWorld1.c. Modify it to add the OpenMP header file:

```
#include "omp.h"
```

Compile it with the command

```
gcc -o HelloWorld1.x HelloWorld1.c
```

and execute it with

```
./HelloWorld1.x
```

What do you notice?

Next, on the command line, set the environment parameter

```
export OMP_NUM_THREADS=4
```

and execute again with

```
./HelloWorld1.x
```

What do you notice?

Finally, recompile and execute with

```
gcc -o HelloWorld1.x HelloWorld1.c -fopenmp
./HelloWorld1.x
```

Pay attention to the `-fopenmp`, which links the OpenMP library. What do you notice?

 [SEE ANSWER](#)

You are now running identical programs on four threads, each of which is printing out a message. The problem is that they don't collaborate on a useful computation. We'll get to that later.

Next, we introduce three routines with which we can extract information about the environment

in which the program executes and information about a specific thread of execution:

- `omp_get_max_threads()` returns the maximum number of threads that are available for computation.
- `omp_get_num_threads()` equals the number of threads in the current *team*: The total number of threads that are available may be broken up into teams that perform separate tasks.
- `omp_get_thread_num()` returns the index that uniquely identifies the thread that calls this function, among the threads in the current team. This index ranges from 0 to one less than the number returned by `omp_get_num_threads()`. In other words, the numbering of the threads starts at zero.

Homework 4.2.3.3 Copy the file `HelloWorld1.c` to `HelloWorld2.c`. Modify the body of the main routine to

```
int nthreads = omp_get_num_threads();
int tid = omp_get_thread_num();
int maxthreads = omp_get_max_threads();

printf( "Hello World! from %d of %d max_threads = %d \n\n",
        tid, nthreads, maxthreads );
```

Compile it and execute it:

```
gcc -o HelloWorld2.x HelloWorld2.c
./HelloWorld2.x
```

What do you notice?

 [SEE ANSWER](#)

In the last exercise, there are four threads available for execution (since `OMP_NUM_THREADS` equals 4), but only one thread is executing (the team only has one thread). The index of that only thread is 1.

Homework 4.2.3.4 Copy the file HelloWorld2.c to HelloWorld3.c. Add the compiler directive

```
#pragma omp parallel
```

immediately before the print statement. Compile it and execute it:

```
gcc -o HelloWorld3.x HelloWorld3.c
./HelloWorld3.x
```

What do you notice?

Next, replace the entire body of main with

```
#pragma omp parallel
{
    int nthreads = omp_get_num_threads();
    int tid = omp_get_thread_num();
    int maxthreads = omp_get_max_threads();

    printf( "Hello World! from %d of %d max_threads = %d \n\n", tid, nthreads\
, maxthreads );
}
```

Compile and execute again. What do you notice?

[SEE ANSWER](#)

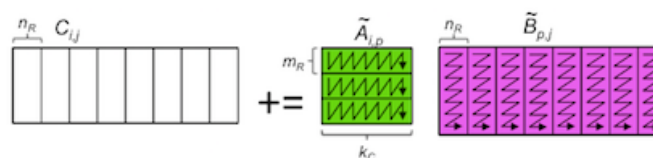
This last exercise illustrates the difference between the total set of threads that is available and the team of threads that a given thread is a member of, and the index of the thread within that team. It also illustrates that you need to be careful about the scope of a “parallel section.”

Often, what work a specific thread performance is determined by its index within a team.

4.3 Multithreading Matrix-Matrix Multiplication

4.3.1 Parallelizing the second loop around the micro-kernel

Now let’s dive right in and parallelize one of the loops. We flipped a coin and it is the second loop around the micro-kernel that we are going to discuss first:



implemented by

```
void LoopTwo( int m, int n, int k, double *Atilde, double *Btilde, double *C, int ldC )
{
    for ( int j=0; j<n; j+=NR ) {
        int jb = min( NR, n-j );
        LoopOne( m, jb, k, Atilde, &Btilde[ j*k ], &gamma( 0,j ), ldC );
    }
}
```

What we want to express is that each threads will perform a subset of multiplications of the submatrix $\tilde{A}_{i,p}$ times the micro-panels of $\tilde{B}_{p,j}$. Each should perform a roughly equal part of the necessary computation. This is indicated by inserting a *pragma*, a directive to the compiler:

```
#pragma omp parallel for
for ( int j=0; j<n; j+=NR ) {
    int jb = min( NR, n-j );
    LoopOne( m, jb, k, Atilde, &Btilde[ j*k ], &gamma( 0,j ), ldC );
}
```

To compile this, we must add `-fopenmp` to the `CFLAGS` in the Makefile. Next, at execution time, you will want to indicate how many threads to use during the execution. First, to find out how many cores are available, you may want to execute

```
$ lscpu
```

which, on the machine where I tried it, yields

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                20
On-line CPU(s) list:   0-19
Thread(s) per core:    2
Core(s) per socket:    10
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz
Stepping:              2
CPU MHz:               1326.273
CPU max MHz:           3000.0000
CPU min MHz:           1200.0000
BogoMIPS:              4594.96
Virtualization:        VT-x
```



```
L1d cache:          32K
L1i cache:          32K
L2 cache:           256K
L3 cache:           25600K
NUMA node0 CPU(s):  0-19
```

and some additional information. What this tells us is that there are 10 cores, with two threads possible per core (this is called *hyperthreading*). On a mac you may want to try executing

```
$ sysctl -n hw.physicalcpu
```

instead. If you are using the BASH shell for Linux¹, you may then want to indicate, for example, that you want to use four threads:

```
$ export OMP_NUM_THREADS=4
```

After you have added the directive to your code, compile it and execute.

Homework 4.3.1.1 So far, you have optimized your choice of MMM in file `GemmFiveLoops_??x??`.c where `??x??` reflects the register blocking for your micro-kernel. At this time, your implementation targets a single core. For the remainder of the exercises, you will start with the simplest implementation that includes packing, for the case where $m_r \times n_r = 12 \times 4$. Now, in directory `Week4/C`,

- Start with `Gemm_Parallel_Loop2_12x4.c`, in which you will find our implementation for a single core with $m_r \times n_r = 12 \times 4$.
- Modify it as described in this unit to parallelize Loop 2 around the micro-kernel.
- Set the number of threads to some number between 1 and the number of CPUs in the target processor.
- Execute `make Parallel_Loop2_12x4`.
- View the resulting performance with `data/ShowPerformance.mlx`.
- Be sure to check if you got the right answer!
- How does performance improve relative to the number of threads you indicated should be used?

 [SEE ANSWER](#)

4.3.2 Parallelizing the first loop around the micro-kernel

One can similarly optimize the first loop around the micro-kernel:

¹If you use a C-shell, the command is `setenv OMP_NUM_THREADS 4`.

Homework 4.3.2.2 In directory `Week4/C`,

- Copy `Gemm_Parallel_Loop2_12x4.c` into `Gemm_Parallel_Loop1_12x4.c`.
- Modify it so that only the first loop around the micro-kernel is parallelized.
- Set the number of threads to some number between 1 and the number of CPUs in the target processor.
- Execute `make Parallel_Loop1_12x4`.
- View the resulting performance with `data/ShowPerformance.mlx`, uncommenting the appropriate lines. (You should be able to do this so that you see previous performance curves as well.)
- Be sure to check if you got the right answer!
- How does the performance improve relative to the number of threads being used?

 [SEE ANSWER](#)

4.3.3 Parallelizing the third loop around the micro-kernel

One can similarly optimize the third loop around the micro-kernel:

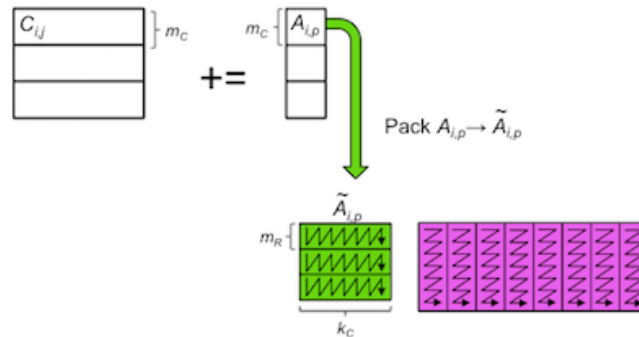
Homework 4.3.3.3 In directory `Week4/C`,

- Copy `Gemm_Parallel_Loop2_12x4.c` into `Gemm_Parallel_Loop3_12x4.c`.
- Modify it so that only the third loop around the micro-kernel is parallelized.
- Set the number of threads to some number between 1 and the number of CPUs in the target processor.
- Execute `make Parallel_Loop3_12x4`.
- View the resulting performance with `ShowPerformance.mlx`, uncommenting the appropriate lines.
- Be sure to check if you got the right answer! This actually is trickier than you might at first think!

 [SEE ANSWER](#)

Observations.

- Parallelizing the third loop is a bit trickier. Likely, you started by inserting the `#pragma` statement and got the wrong answer. The problem is that you are now parallelizing



implemented by

```
void LoopThree( int m, int n, int k, double *A, int ldA, double *Atilde,
               double *Btilde, double *C, int ldC )
{
    for ( int i=0; i<m; i+=MC ) {
        int ib = min( MC, m-i );    /* Last loop may not involve a full block */
        PackBlockA_MCxKC( ib, k, &alpha( i, 0 ), ldA, Atilde );

        LoopTwo( ib, n, k, Atilde, Btilde, &gamma( i, 0 ), ldC );
    }
}
```

The problem with a naive parallelization is that all threads pack their block of A into the same buffer $Atilde$. There are two solutions to this:

- Quick and dirty: allocate and free a new buffer in this loop, so that each iteration, and hence each thread, gets a different buffer.
- Cleaner: In Loop 5 around the micro-kernel, we currently allocate one buffer for $Atilde$:

```
double *Atilde = ( double * ) malloc( MC * KC * sizeof( double ) );
```

What one could do is to allocate a larger buffer here, enough for as many blocks \tilde{A} as there are threads, and then in the third loop around the micro-kernel one can index into this larger buffer to give each thread its own space. For this, one needs to know about a few OpenMP inquiry routines:

- * `omp_get_num_threads()` returns the number of threads being used in a parallel section.
- * `omp_get_thread_num()` returns the thread number of the thread that calls it.

Now, this is all a bit tricky, because `omp_get_num_threads()` returns the number of threads being used only in a parallel section. So, to allocate enough space, one would try to change the code in Loop 5 to

```
double *Atilde;

#pragma omp parallel
{
    if ( omp_get_thread_num() == 0 )
        malloc( MC * NC * sizeof( double ) * omp_get_num_threads() );
}
```

Here the curly brackets indicate the scope of the parallel section.

The call to `free` must then be correspondingly modified so that only thread 0 releases the buffer.

Now, in Loop 2, one can index into this larger buffer.

4.4 Parallelizing More

4.4.1 Speedup, efficiency, etc.

Amdahl's law is a simple observation regarding how much speedup can be expected when one parallelizes a code. First some definitions.

Sequential time. The *sequential time* for an operation is given by

$$T(n),$$

where n is a parameter that captures the size of the problem being solved. For example, if we only time problems with square matrices, then n would be the size of the matrices. $T(n)$ is equal the time required to compute the operation on a single core.

Parallel time. The *parallel time* for an operation and a given implementation using t threads is given by

$$T_t(n).$$

Speedup. The *speedup* for an operation and a given implementation using t threads is given by

$$T(n)/T_t(n).$$

It measures how much speedup is attained by using t threads. Generally speaking, it is assumed that

$$S_t(n) = T(n)/T_t(n) \leq t.$$

This is not always the case: sometimes t threads collectively have resources (e.g., more cache memory) that allows speedup to exceed t . Think of the case where you make a bed with one person versus two people: You can often finish the job more than twice as fast because two people do not need to walk back and forth from one side of the bed to the other.

Efficiency. The *efficiency* for an operation and a given implementation using t threads is given by

$$S_t(n)/t.$$

It measures how efficiently the t threads are being used.

GFLOPS. We like to report GFLOPS (billions of floating point operations per second). Notice that this gives us an easy way of judging efficiency: If we know what the theoretical peak of a core is, in terms of GFLOPS, then we can easily judge how efficiently we are using the processor relative to the theoretical peak. How do we compute the theoretical peak? We can look up the number of cycles a core can perform per second and the number of floating point operations it can perform per cycle. This can then be converted to a peak rate of computation for a single core (in billions of floating point operations per second). If there are multiple cores, we just multiply this peak by the number of cores.

Of course, this is all complicated by the fact that as more cores are utilized, each core often executes at a lower clock speed so as to keep the processor from overheating.

4.4.2 Ahmdahl's law

Ahmdahl's law is a simple observation about the limits on parallel speed and efficiency. Consider an operation that requires sequential time

$$T$$

for computation. What if fraction f of this time is **not** parallelized, and the other fraction $(1 - f)$ is parallelized with t threads. Then the total execution time is

$$T_t \geq (1 - f)T/t + fT.$$

This, in turn, means that the speedup is bounded by

$$S \leq \frac{T}{(1 - f)T/t + fT}.$$

Now, even if the parallelization with t threads is such that the fraction that is being parallelized takes no time at all ($(1 - f)T/t = 0$), then we still find that

$$S \leq \frac{T}{(1 - f)T/t + fT} \leq \frac{T}{fT} = \frac{1}{f}.$$

What we notice is that the maximal speedup is bounded by the inverse of the fraction of the execution time that is not parallelized. So, if 20% of the code is not optimized, then no matter how

many threads you use, you can at best compute the result five times faster. This means that one must think about parallelizing all significant parts of a code.

Ahmdahl's law says that if does not parallelize the part of a sequential code in which fraction f time is spent, then the speedup attained regardless of how many threads are used is bounded by $1/f$.

The point is: So far, we have focused on parallelizing the computational part of matrix-matrix multiplication. We should also parallelize the packing, since that is a nontrivial part of the total execution time.

4.4.3 Parallelizing the packing

Some observations:

- If you choose to parallelize Loop 3 around the micro-kernel, then the packing into \tilde{A} is already parallelized, since each thread packs its own such block. So, you will want to parallelize the packing of \tilde{B} .
- If you choose to parallelize Loop 2 around the micro-kernel, you will also want to parallelize the packing of \tilde{A} .
- Be careful: we purposely wrote the routines that pack \tilde{A} and \tilde{B} so that a naive parallelization will give the wrong answer. Analyze carefully what happens when multiple threads execute the loop...

Homework 4.4.3.1 In directory Week4/C,

- Copy `Gemm_Parallel_Loop2_12x4.c` into `Gemm_Parallel_Loop2_Parallel_Pack_12x4.c`.
- Parallelize the packing of \tilde{B} and/or \tilde{A} .
- Set the number of threads to some number between 1 and the number of CPUs in the target processor.
- Execute `make Parallel_Loop2_Parallel_Pack_12x4`.
- View the resulting performance with `ShowPerformance.mlx`, uncommenting the appropriate lines.
- Be sure to check if you got the right answer! This actually is trickier than you might at first think!

 [SEE ANSWER](#)

4.5 Enrichment

4.6 Wrapup

