

# **Targeting a Single Core**

# A.1 Opening Remarks

#### A.1.1 Launch • to edX

Homework A.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

\*\* DO EXERCISE ON edX

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

$$A = \left( egin{array}{ccccc} lpha_{0,0} & lpha_{0,1} & \cdots & lpha_{0,k-1} \ lpha_{1,0} & lpha_{1,1} & \cdots & lpha_{1,k-1} \ dots & dots & dots & dots \ lpha_{m-1,0} & lpha_{m-1,1} & \cdots & lpha_{m-1,k-1} \ \end{array} 
ight), B = \left( egin{array}{ccccc} eta_{0,0} & eta_{0,1} & \cdots & eta_{0,n-1} \ eta_{1,0} & eta_{1,1} & \cdots & eta_{1,n-1} \ dots & dots & dots & dots \ eta_{k-1,0} & eta_{k-1,1} & \cdots & eta_{k-1,n-1} \ \end{array} 
ight),$$

and

$$C=\left(egin{array}{cccc} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,n-1} \ \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,n-1} \ dots & dots & dots \ \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,n-1} \end{array}
ight).$$

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

defined as

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

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

```
\begin{aligned} & \textbf{for } i := 0, \dots, m-1 \\ & \textbf{for } j := 0, \dots, n-1 \\ & \textbf{for } p := 0, \dots, k-1 \\ & \gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j} \\ & \textbf{end} \\ & \textbf{end} \\ & \textbf{end} \end{aligned}
```

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

**Homework A.1.1.2** In directory LAFFPfHPC/Assignments/WeekA/C/ you will find a simple implementation that computes C := AB + C (GEMM) in GemmIJP.c. Compile, link, and execute it by follow the instructions in the MATLAB Live Script TestGemmIJP.mlx in that directory.

SEE ANSWER

**☞** DO EXERCISE ON edX

Λ -	12	Outline	Wook 1	Yha ot
<b>A</b> -		<b>₹</b>	VVEER	III HIIA

# A.2 Mapping Matrices to Memory

### A.2.1 Mapping matrices to memory

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

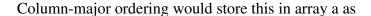
Consider the matrix

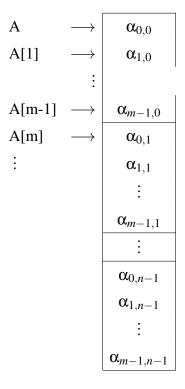
$$\begin{pmatrix}
1 & -2 & 2 \\
-1 & 1 & 3 \\
-2 & 2 & -1
\end{pmatrix}$$

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

which is known as *column-major ordering*. More generally, consider the matrix

$$\left(egin{array}{ccccc} lpha_{0,0} & lpha_{0,1} & \cdots & lpha_{0,k-1} \ lpha_{1,0} & lpha_{1,1} & \cdots & lpha_{1,k-1} \ dots & dots & dots \ lpha_{m-1,0} & lpha_{m-1,1} & \cdots & lpha_{m-1,k-1} \ \end{array}
ight).$$





Obviously, one could use the alternative known as *row-major ordering*. Since we won't use that, we will not further discuss it.

### A.2.2 The leading dimension

Very frequently we will work with a matrix that is a submatrix of a larger matrix. Consider Figure A.1. 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 will often 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 1dA holds the leading dimension parameter.

# A.3 Ordering the Loops

When we talk about loops for matrix-matrix multiplication, it helps to keep the following picture in mind, which illustrates which loop indices are used for what dimensions of the matrices:

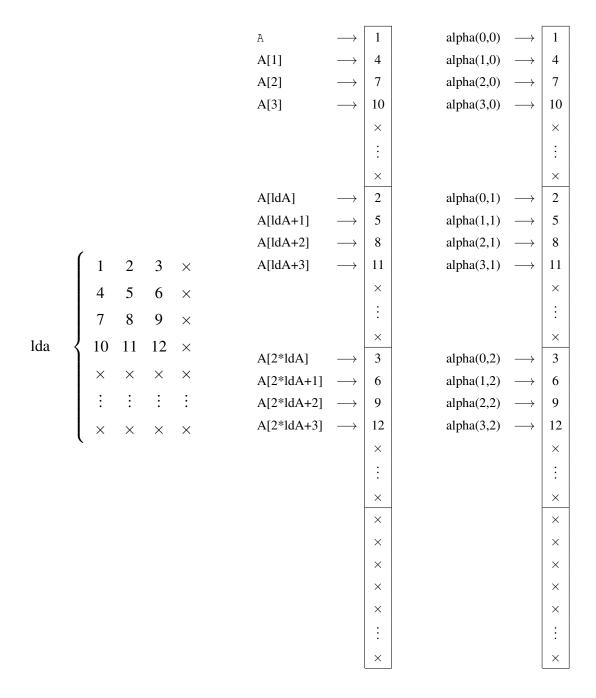
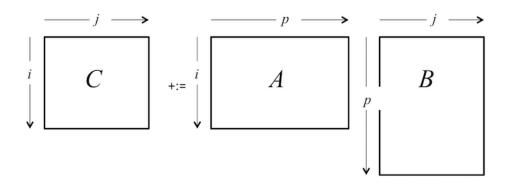


Figure A.1: Addressing a matrix embedded in an array with ldA rows. At the left we illustrate a  $4 \times 3$  submatrix of a ldA  $\times$  4 matrix. In the middle, we illustrate how this is mapped into an linear array a. In the right, we show how defining the C macro #define alpha(i, j) A[ (j)\*ldA + (i)] allows us to address the matrix in a more natural way.



We try to be consistent in this use.

### A.3.1 The IJP and JIP orderings

In the opening of this week, we discussed the algorithm for multiplying  $m \times k$  matrix A times  $k \times n$  matrix B, adding the result to  $m \times n$  matrix C:

```
\begin{aligned} & \textbf{for } i := 0, \dots, m-1 \\ & \textbf{for } j := 0, \dots, n-1 \\ & \textbf{for } p := 0, \dots, k-1 \\ & \gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j} \\ & \textbf{end} \\ & \textbf{end} \\ & \textbf{end} \end{aligned}
```

Given the discussion about how matrices are mapped to memory, we can translate this pseudo-code into the following routine coded in C:

github/LAFF-On-PfHPC/Assignments/WeekA/C/GemmIJP.c

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:

$$C = \left(\begin{array}{c|c|c} \underline{\gamma_{0,0}} & \gamma_{0,1} & \cdots & \gamma_{0,n-1} \\ \hline \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,n-1} \\ \hline \vdots & & \vdots & \\ \hline \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,n-1} \end{array}\right), \quad A = \left(\begin{array}{c|c} \overline{a_0^T} \\ \hline \overline{a_1^T} \\ \hline \vdots \\ \hline \overline{a_{m-1}^T} \end{array}\right), \quad \text{and } B = \left(\begin{array}{c|c} b_0 & b_1 & \cdots & b_{n-1} \end{array}\right).$$

We then notice that

This 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

$$\begin{aligned} &\textbf{for } i := 0, \dots, m-1 \\ &\textbf{for } j := 0, \dots, n-1 \\ &\textbf{for } p := 0, \dots, k-1 \\ &\gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j} \\ &\textbf{end} \end{aligned} \right\} \quad \gamma_{i,j} := \widetilde{a}_i^T b_j + \gamma_{i,j} \\ &\textbf{end} \\ &\textbf{end} \\ &\textbf{end} \end{aligned}$$

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:

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

which we call the JIP ordering of the loops.

### A.3.2 The IPJ and PIJ orderings

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

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

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

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

We then notice that

$$\left(\begin{array}{c|cccc} \overline{c_0^T} \\ \hline \overline{c_1^T} \\ \hline \vdots \\ \hline \overline{c_{m-1}^T} \end{array}\right) := \left(\begin{array}{c|ccccc} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,k-1} \\ \hline \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,k-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,k-1} \end{array}\right) \left(\begin{array}{c} \overline{b_0^T} \\ \hline \overline{b_1^T} \\ \hline \vdots \\ \hline \overline{b_{k-1}^T} \end{array}\right) + \left(\begin{array}{c} \overline{c_0^T} \\ \hline \overline{c_1^T} \\ \hline \vdots \\ \hline \overline{c_{m-1}^T} \end{array}\right)$$

**for** i := 0, ..., m-1

or, equivalently,

end

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

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

$$\left. \begin{array}{l} \textbf{for } p := 0, \dots, k-1 \\ \textbf{for } j := 0, \dots, n-1 \\ \gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j} \\ \textbf{end} \end{array} \right\} \quad \widetilde{c}_i^T := \alpha_{i,p} \widetilde{b}_p^T + \widetilde{c}_i \\ \textbf{end} \\ \textbf{end} \\ \textbf{One can switch the order of the outer two loops to get} \\ \textbf{for } p := 0, \dots, k-1 \\ \textbf{for } i := 0, \dots, m-1 \\ \gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j} \\ \textbf{end} \end{array} \right\} \quad \widetilde{c}_i^T := \alpha_{i,p} \widetilde{b}_p^T + \widetilde{c}_i^T \\ \textbf{end} \\ \textbf{end}$$

The outer loop in this second algorithm fixes the row of B that is used 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:

$$egin{pmatrix} \left( egin{array}{c} \overline{\widehat{c}_0^T} \ \hline \overline{\widehat{c}_1^T} \ \hline \vdots \ \hline \widehat{c}_{m-1}^T \end{array} 
ight) := egin{pmatrix} \overline{\widehat{c}_0^T} & + & lpha_{0,0} \widehat{b}_0^T \ \hline \overline{\widehat{c}_1^T} & + & lpha_{1,0} \widehat{b}_0^T \ \hline \hline \overline{\widehat{c}_{m-1}^T} & + & lpha_{m-1,0} \widehat{b}_0^T \ \hline \end{pmatrix}.$$

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

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

and so forth.

### A.3.3 The JPI and PJI orderings

Let's consider the ordering *JPI*:

```
\begin{aligned} & \textbf{for } j := 0, \dots, n-1 \\ & \textbf{for } p := 0, \dots, k-1 \\ & \textbf{for } i := 0, \dots, m-1 \\ & \gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j} \\ & \textbf{end} \\ & \textbf{end} \\ & \textbf{end} \end{aligned}
```

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

$$\left( \begin{array}{c|c|c|c} c_{0} & c_{1} & \cdots & c_{n-1} \end{array} \right) :=$$

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

so that

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

The algorithm captures this as

$$\begin{array}{l} \text{for } j:=0,\ldots,n-1\\ \text{for } p:=0,\ldots,k-1\\ \\ \text{for } i:=0,\ldots,m-1\\ \\ \gamma_{i,j}:=\alpha_{i,p}\beta_{p,j}+\gamma_{i,j} \\ \text{end} \end{array} \right\} \quad c_j:=\beta_{p,j}a_p+c_j\\ \\ \text{end} \\ \\ \text{end} \end{array}$$

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

$$\begin{array}{l} \text{for } p:=0,\ldots,k-1\\ \text{for } j:=0,\ldots,n-1\\ \\ \text{for } i:=0,\ldots,m-1\\ \\ \gamma_{i,j}:=\alpha_{i,p}\beta_{p,j}+\gamma_{i,j} \\ \text{end} \end{array} \right\} \quad c_j:=\beta_{p,j}a_p+c_j\\ \\ \text{end} \\ \\ \text{end} \end{array}$$

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:

$$\begin{pmatrix} c_0 \mid c_1 \mid \cdots \mid c_{n-1} \end{pmatrix} :=$$

$$\begin{pmatrix} c_0 + \beta_{0,0}a_0 & | c_1 + \beta_{0,1}a_0 & | \cdots \mid c_{n-1} + \beta_{0,n-1}a_0 \end{pmatrix}$$

In the second iteration of the outer loop it computes

$$\begin{pmatrix}
c_0 & c_1 & \cdots & c_{n-1} \\
c_0 + \beta_{0,0}a_0 + \beta_{1,0}a_1 & c_1 + \beta_{0,1}a_0 + \beta_{1,1}a_1 & \cdots & c_{n-1} + \beta_{0,n-1}a_0 + \beta_{1,n-1}a_1
\end{pmatrix}$$

and so forth.

### A.3.4 Compare and contrast

**Homework A.3.4.1** In directory WeekA/C/ use the implementation in GemmIJP.c to implement the other five loop oderings, naming the routines and files in the obvious way. Then use the Live Script TestGemmAll.mlx in that directory to test and time all implementations.

**SEE ANSWER** 

**☞** DO EXERCISE ON edX

Homework A.3.4.2 Which two orders of the loops yields the best performance? Why?

SEE ANSWER

**DO EXERCISE ON edX** 

# A.4 Building blocks

# A.4.1 Vector-vector operations: dot product

In our discussion in Section A.3, we encounted the operation

$$\gamma_{i,j} := \widetilde{a}_i^T b_j + \gamma_{i,j}.$$

This is an instance of the updating of a scalar by the result of a *dot product*, also known as an *inner product*, with the result added to a scalar:

$$\gamma := x^T y + \gamma$$
.

Since it adds to a scalar, we will refer to this operations as DOTS.

A routine that implements this operation may be implemented as

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

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

  return;
}</pre>
```

github/LAFF-On-PfHPC/Assignments/WeekA/C/Dots.c

In this routine, inex and iney indicate the *stride* with which one has to march through memory from one element of the vector to the next. Thus, one could translate the algorithm for computing C := AB + C given by

```
\begin{array}{l} \textbf{for } i:=0,\ldots,m-1\\ \textbf{for } j:=0,\ldots,n-1\\ \\ \textbf{for } p:=0,\ldots,k-1\\ \\ \gamma_{i,j}:=\alpha_{i,p}\beta_{p,j}+\gamma_{i,j} \end{array} \right\} \quad \gamma_{i,j}:=\widetilde{a}_i^Tb_j+\gamma_{i,j}\\ \\ \textbf{end}\\ \\ \textbf{end}\\ \\ \textbf{end} \end{array}
```

into the routine

github/LAFF-On-PfHPC/Assignments/WeekA/C/GemmIJP\_Dots.c

### A.4.2 Vector-vector operations: axpy

In our discussion in Section A.3, we also encountered the operations

$$\widetilde{c}_i^T := \alpha_{i,p} \widetilde{b}_p^T + \widetilde{c}_i^T$$

and

$$c_i := \beta_{p,j} a_p + c_j$$
.

These are examples of what is commonly called an AXPY operation: Alpha times X Plus Y:

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

which, for vectors of size n, updates

$$egin{aligned} \left( egin{aligned} \hline \psi_0 \ \hline \hline \psi_1 \ \hline \vdots \ \hline \psi_{n-1} \end{aligned} 
ight) := \left( egin{aligned} \hline lpha \chi_0 + \psi_0 \ \hline lpha \chi_1 + \psi_1 \ \hline \vdots \ \hline lpha \chi_{n-1} + \psi_{n-1} \end{aligned} 
ight) \end{aligned}$$

**Homework A.4.2.1** Complete the following routine for computing  $y := \alpha x + y$  where the size of the vectors is given in parameter n,  $\alpha$  is stored in parameter alpha, vector x is stored in array x with stride incx, and vector y is stored in array y with stride incy.

github/LAFF-On-PfHPC/Assignments/WeekA/C/Axpy.c

You can find the partial implementation in WeekA/C/Axpy.c. You can test the implementation with Live Script TestAxpy.mlx.

```
SEE ANSWERDO EXERCISE ON edX
```

In an implementation of the algorithm for computing C := AB + C given by

```
\left. \begin{array}{l} \textbf{for } i := 0, \dots, m-1 \\ \textbf{for } p := 0, \dots, k-1 \\ \textbf{for } j := 0, \dots, n-1 \\ \gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j} \\ \textbf{end} \end{array} \right\} \quad \widetilde{c}_i^T := \alpha_{i,p} \widetilde{b}_p^T + \widetilde{c}_i \\ \textbf{end} \\ \textbf{end} \\ \textbf{end} \end{array}
```

and illustrated by

one can now replace the loop indexed by j with a call to Axpy.

#### Homework A.4.2.2 Complete the following routine

github/LAFF-On-PfHPC/Assignments/WeekA/C/GemmIPJ\_Axpy.c

You can find the partial implementation in WeekA/C/GemmIPJ\_Axpy.c. You can test the implementation with Live Script TestGemmIPJ\_Axpy.mlx.

SEE ANSWERDO EXERCISE ON edX

# A.4.3 Matrix-vector multiplication via AXPY operations

Consider the matrix-vector multiplication

$$y := Ax + y$$
.

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

$$y := \left( a_0 \mid a_1 \mid \dots \mid a_{n-1} \right) \left( \frac{\frac{\chi_0}{\chi_1}}{\frac{\vdots}{\chi_{n-1}}} \right) + y$$
$$= \chi_0 a_0 + \chi_1 a_1 + \dots + \chi_{n-1} a_{n-1} + y.$$

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:

$$\left. \begin{array}{l} \textbf{for } j := 0, \dots, n-1 \\ \textbf{for } i := 0, \dots, m-1 \\ \psi_i := \alpha_{i,j} \chi_j + \psi_i \\ \textbf{end} \end{array} \right\} \quad y := \chi_j a_j + y \\ \textbf{end} \end{array}$$

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

github/LAFF-On-PfHPC/Assignments/WeekA/C/GemvJI\_Axpy.c

# A.4.4 Matrix-vector multiplication via dots operations

Again, consider the matrix-vector multiplication

$$y := Ax + y$$
.

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

$$\left(\frac{\psi_0}{\frac{\psi_1}{\vdots}}\right) := \left(\frac{\widetilde{a}_0^T}{\widetilde{a}_1^T}\right) x + \left(\frac{\psi_0}{\frac{\psi_1}{\vdots}}\right) = \left(\frac{\widetilde{a}_0^T x + \psi_0}{\widetilde{a}_1^T x + \psi_1}\right) \cdot \left(\frac{\widetilde{a}_0^T x + \psi_0}{\widetilde{a}_{m-1}^T x + \psi_{m-1}}\right).$$

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

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

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

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

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

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

github/LAFF-On-PfHPC/Assignments/WeekA/C/GemvIJ\_Dots.c

### A.4.5 Rank-1 update via AXPY operations

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

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

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

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

We notice that we could have partitioned A by columns and y by elements to find that

$$\begin{pmatrix} a_0 \mid a_1 \mid \cdots \mid a_{n-1} \end{pmatrix} := x \begin{pmatrix} \psi_0 \mid \psi_1 \mid \cdots \mid \psi_{n-1} \end{pmatrix} + \begin{pmatrix} a_0 \mid a_1 \mid \cdots \mid a_{n-1} \end{pmatrix}$$

$$= \begin{pmatrix} x\psi_0 + a_0 \mid x\psi_1 + a_1 \mid \cdots \mid x\psi_{n-1} + a_{n-1} \end{pmatrix}$$

$$= \begin{pmatrix} \psi_0 x + a_0 \mid \psi_1 x + a_1 \mid \cdots \mid \psi_{n-1} x + a_{n-1} \end{pmatrix}.$$

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

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

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

### Homework A.4.5.1 Complete the following routine

github/LAFF-On-PfHPC/Assignments/WeekA/C/GerJI\_Axpy.c

You can find the partial implementation in WeekA/C/GerJI\_Axpy.c. You can test the implementation with Live Script TestGerJI\_Axpy.mlx.

SEE ANSWERDO EXERCISE ON edX

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

$$\left( \frac{\widetilde{a}_0^T}{\widetilde{a}_1^T} \right) := \left( \frac{\chi_0}{\chi_1} \right) y^T + \left( \frac{\widetilde{a}_0^T}{\widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_1^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_0^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_0^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_0^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_0^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_0^T} \right) = \left( \frac{\chi_0 y^T + \widetilde{a}_0^T}{\chi_1 y^T + \widetilde{a}_0^T} \right)$$

leading to the algorithm

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

#### and implementation

github/LAFF-On-PfHPC/Assignments/WeekA/C/GerIJ\_Axpy.c

# A.5 Matrix-Matrix Multiplication, Again

### A.5.1 Matrix-matrix multiplication via matrix-vector multiplications

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

$$\begin{pmatrix} c_0 \mid c_1 \mid \cdots \mid c_{n-1} \end{pmatrix} := A \begin{pmatrix} b_0 \mid b_1 \mid \cdots \mid b_{n-1} \end{pmatrix} + \begin{pmatrix} c_0 \mid c_1 \mid \cdots \mid c_{n-1} \end{pmatrix} \\
= \begin{pmatrix} Ab_0 + c_0 \mid Ab_1 + c_1 \mid \cdots \mid Ab_{n-1} + c_{n-1} \end{pmatrix}$$

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

```
\left. \begin{array}{l} \textbf{for } j := 0, \dots, n-1 \\ \textbf{for } p := 0, \dots, k-1 \\ \textbf{for } i := 0, \dots, m-1 \\ \gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j} \end{array} \right\} \quad c_j := \beta_{p,j} a_p + c_j \\ \textbf{end} \\ \textbf{end} \\ \\ \textbf{end} \\ \\ \textbf{and} \\ \\ \textbf{for } j := 0, \dots, n-1 \\ \textbf{for } i := 0, \dots, m-1 \\ \textbf{for } p := 0, \dots, k-1 \\ \gamma_{i,j} := \alpha_{i,p} \beta_{p,j} + \gamma_{i,j} \\ \textbf{end} \\ \\ \end{array} \right\} \quad \widetilde{\gamma}_{i,j} := \widetilde{a}_i^T b_j + \gamma_{i,j} \\ \\ \textbf{end} \\ \\ \textbf{end} \\ \\ \textbf{end} \\ \\ \end{array} \right\} \quad \alpha_j := A b_j + c_j \\ \\ \alpha_j := A b_j + c_j \\ \\ \textbf{end} \\ \\ \textbf{end} \\ \\ \textbf{end} \\ \\ \end{aligned}
```

The second can then by translated into code as

github/LAFF-On-PfHPC/Assignments/WeekA/C/GemmJIP\_Gemv.c

#### **Homework A.5.1.1** Complete the following routine

github/LAFF-On-PfHPC/Assignments/WeekA/C/GemmJPI\_Gemv.c

You can find the partial implementation in WeekA/C/GemmJPI\_Gemv.c. You can test the implementation with Live Script TestGemmJPI\_Gemv.mlx.

SEE ANSWERDO EXERCISE ON edX

### A.5.2 Matrix-matrix multiplication via rank-1 updates

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

$$C := \left( a_0 \left| a_1 \right| \cdots \left| a_{k-1} \right) \left( \frac{\widetilde{b}_0^T}{\widetilde{b}_1^T} \right) + C$$

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

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

$$\begin{array}{l} \text{for } p:=0,\ldots,k-1 \\ \text{for } j:=0,\ldots,n-1 \\ \text{for } i:=0,\ldots,m-1 \\ \gamma_{i,j}:=\alpha_{i,p}\beta_{p,j}+\gamma_{i,j} \end{array} \right\} \quad c_j:=\beta_{p,j}a_p+c_j \\ \text{end} \\ \text{end} \\ \end{array} \right\} \quad C:=a_p\widetilde{b}_p^T+C$$

and

$$\begin{array}{l} \text{for } p:=0,\ldots,k-1 \\ \text{for } i:=0,\ldots,m-1 \\ \text{for } j:=0,\ldots,n-1 \\ \gamma_{i,j}:=\alpha_{i,p}\beta_{p,j}+\gamma_{i,j} \end{array} \right\} \quad \widetilde{c}_i^T:=\alpha_{i,p}\widetilde{b}_p^T+\widetilde{c}_i^T \\ \text{end} \\ \text{end} \\ \end{array} \right\} \quad C:=a_p\widetilde{b}_p^T+C$$
 
$$\begin{array}{l} \text{end} \\ \end{array}$$

**Homework A.5.2.1** Implement the PJI ordering of the loops using the routine GerJI.c by completing the following routine

github/LAFF-On-PfHPC/Assignments/WeekA/C/GemmPJI\_Ger.c

You can find the partial implementation in WeekA/C/GemmPJI\_Ger.c. You can test the implementation with Live Script TestGemmPJI\_Ger.mlx.

SEE ANSWERDO EXERCISE ON edX

# A.5.3 Matrix-matrix multiplication via row-times-matrix multiplications

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

$$\left(\frac{\overrightarrow{c}_{0}^{T}}{\overrightarrow{c}_{1}^{T}}\right) := \left(\frac{\overrightarrow{a}_{0}^{T}}{\overrightarrow{a}_{1}^{T}}\right) B + \left(\frac{\overrightarrow{c}_{0}^{T}}{\overrightarrow{c}_{1}^{T}}\right) B + \left(\frac{\overrightarrow{c}_{0}^{T}}{\overrightarrow{c}_{m-1}^{T}}\right) B + \left(\frac{\overrightarrow{c}_{0}^{T}}{\overrightarrow{c}_{m-1}^{T}}\right) B + \left(\frac{\overrightarrow{a}_{0}^{T} B + \overrightarrow{c}_{0}^{T}}{\overrightarrow{c}_{m-1}^{T}}\right) B + \left(\frac{\overrightarrow{a}_{0}^{T} B + \overrightarrow{c}_{0}^{T}}{\overrightarrow{c}_{0}^{T}}\right) B + \left(\frac{\overrightarrow{a}_{0}^{T} B + \overrightarrow{$$

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:

We don't discuss the "row times matrix multiplication"-based implementation further since it inherently accesses rows in the inner-most loop and hence will not attain good performance when column-major ordering is used to store matrices.

#### A.5.4 Discussion

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

• multiple matrix-vector multiplications:

$$\left(\begin{array}{c|c}c_0 & c_1 & \cdots & c_{n-1}\end{array}\right) = \left(\begin{array}{c|c}Ab_0 + c_0 & Ab_1 + c_1 & \cdots & Ab_{n-1} + c_{n-1}\end{array}\right)$$

• multiple rank-1 updates:

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

• multiple row times matrix multiplications:

$$\left( \begin{array}{c} \overline{c_0^T} \\ \overline{c_1^T} \\ \hline \vdots \\ \overline{c_{m-1}^T} \end{array} \right) = \left( \begin{array}{c} \overline{a_0^T}B + \overline{c_0^T} \\ \overline{a_1^T}B + \overline{c_1^T} \\ \hline \vdots \\ \overline{a_{m-1}^T}B + \overline{c_{m-1}^T} \end{array} \right)$$

So, for the outer loop there are three choices: *j*, *p*, or *i*. This may give the appearance that there are only three algorithms. However, the matrix-vector multiply and rank-1 update hide a double loop and the order of these two loops can be chosen, to bring us back to six choices for algorithms. Importantly, matrix-vector multiplication can be organized so that matrices are addressed by columns in the inner-most loop (the JI order for computing GEMV) as can the rank-1 update (the JI order for computing GER).

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

We have seen many examples of blocked matrix-matrix multiplication already, where matrices were partitioned by rows, columns, or individual entries.

# A.6 Optimization: A Start

### A.6.1 Blocked matrix-matrix multiplication

Key to understanding how we are going to optimize MMM is to understand blocked matrix-matrix multiplication. Consider C := AB + C. In terms of the elements of the matrices, this means that each  $\gamma_{i,j}$  is computed by

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

Now, partition the matrices into submatrices:

and

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

$$\bullet \ \sum_{i=0}^{M-1} m_i = m,$$

• 
$$\sum_{i=0}^{N-1} n_i = m$$
, and

$$\bullet \ \sum_{p=0}^{K-1} k_i = m.$$

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," matrix-matrix multiplication with partitioning matrices proceeds exactly as does matrix-matrix multiplication with the elements of the matrices *except* that the individual multiplications with the submatrices do not necessarily commute.

### A.6.2 Motivating example

We are going to focus on implementing a kernel for a MMM that computes C := AB + C where C is  $4 \times 4$ , A is  $4 \times k$ , and B is  $k \times 4$ . Now

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

$$= \begin{pmatrix} \alpha_{0,0} & \alpha_{1,0} & \alpha_{1$$

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

$$\begin{cases} \mathbf{for} \ p = 0, \dots, k-1 \\ \begin{pmatrix} \gamma_{0,0} & \gamma_{0,1} & \gamma_{0,2} & \gamma_{0,3} \\ \gamma_{1,0} & \gamma_{1,1} & \gamma_{1,2} & \gamma_{1,3} \\ \gamma_{2,0} & \gamma_{2,1} & \gamma_{2,2} & \gamma_{2,3} \\ \gamma_{3,0} & \gamma_{3,1} & \gamma_{3,2} & \gamma_{3,3} \end{pmatrix} + = \begin{pmatrix} \alpha_{0,k} \\ \alpha_{1,k} \\ \alpha_{2,k} \\ \alpha_{3,k} \end{pmatrix} \begin{pmatrix} \beta_{k,0} & \beta_{k,1} & \beta_{k,2} & \beta_{k,3} \end{pmatrix}$$

or, equivalently to emphasize computations with vectors,

#### **Vector registers**

#### A.6.3 Vector instructions

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 an FMA unit of the (processing) 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 (FPUs) that can simultaneously perform multiple FMA operations.

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 languages for performing vector instructions. We illustrate how the intrinsic operations are used by focusing on the motivating example, partially implemented in Figure A.2 and further illustrated in Figure A.3.

To use the intrinsics, we start by including the header file immintrin.h.

```
#include <immintrin.h>
```

The declaration

```
_{m256d gamma_0123_0} = _{mm256_loadu_pd( & gamma( 0,0 ) );}
```

creates gamma\_0123\_0 as a variable that references a vector register with four double precision numbers and loads it with the four such numbers that are stored starting at address &gamma(0, 0).

```
\#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
#include < immintrin.h>
void GemmIntrinsicsKernel_m4xn4( int k, double *A, int ldA,
                                       double *B, int ldB,
                                       double *C, int ldC )
 _{m256d} gamma_0123_0 = _{mm256}_loadu_pd( &gamma( 0,0 ) );
 _{m256d} gamma_0123_1 = _{mm256}_loadu_pd(&gamma(0,1));
 _{m256d} gamma_0123_2 = _{mm256}loadu_pd(&gamma(0,2));
  m256d gamma 0123 3 = mm256 loadu pd( &gamma( 0,3 ) );
 for ( int p=0; p<k; p++ ) {
   /* Declare a vector register to hold the current column of A and load
      it with the four elements of that column. */
   _{m256d} = _{mn256_{loadu_pd}} ( & alpha( 0,p ) );
   /* Declare a vector register to hold elements beta (p,0) and duplicate
      it in the register. */
   _{m256d} beta_p_0 = _{mm256}broadcast_sd( &beta( p, 0) );
   /* update the first column of C with the current column of A times
      beta (p,0) */
   qamma_0123_0 = _mm256_fmadd_pd( alpha_0123_p, beta_p_0, qamma_0123_0 );
   /* REPEAT for second, third, and fourth columns of C. Notice that the
      current column of A needs not be reloaded. */
 }
 /* Store the updated results */
 _mm256_storeu_pd( &gamma(0,0), gamma_0123_0 );
 _mm256_storeu_pd( &gamma(0,1), gamma_0123_1 );
 _mm256_storeu_pd( &gamma(0,2), gamma_0123_2 );
 _mm256_storeu_pd( &gamma(0,3), gamma_0123_3 );
 return;
```

github/LAFF-On-PfHPC/Assignments/WeekA/C/GemmIntrinsicsKernel\_m4xn4.c

Figure A.2: Partial routine for computing C := AB + C where C is  $4 \times 4$ .

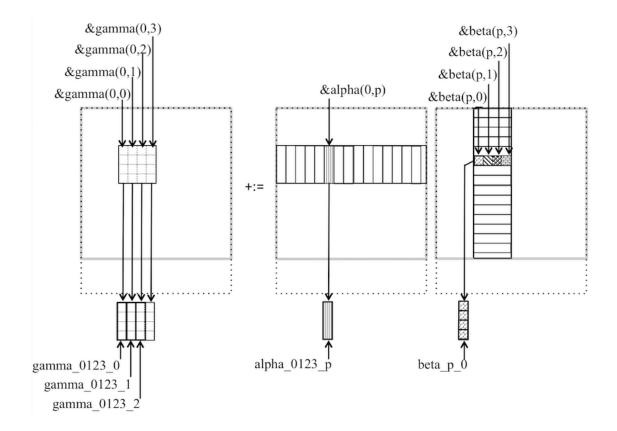


Figure A.3: Illustration of how the routine in Figure A.2 indexes into the matrices.

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

$$\left(\begin{array}{c} \gamma_{0,0} \\ \gamma_{1,0} \\ \gamma_{2,0} \\ \gamma_{3,0} \end{array}\right).$$

This is repeated for the other three columns of C.

The loop in Figure A.2 implements

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

leaving the result in the vector registers. Each iteration starts by declaring vector register variable

alpha\_0123\_p, loading it with the contents of

$$\left(egin{array}{c} lpha_{0,p} \ lpha_{1,p} \ lpha_{2,p} \ lpha_{3,p} \end{array}
ight).$$

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

```
_{m256d} beta_p_0 = _{mm256}broadcast_sd( &beta(p, 0) );
```

The command

then performs the computation

$$\left(egin{array}{l} \gamma_{0,0}+:=lpha_{0,p} imeseta_{p,0}\ \gamma_{1,0}+:=lpha_{1,p} imeseta_{p,0}\ \gamma_{2,0}+:=lpha_{2,p} imeseta_{p,0}\ \gamma_{3,0}+:=lpha_{3,p} imeseta_{p,0} \end{array}
ight).$$

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

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

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

```
_mm256_storeu_pd( &gamma(0,0), gamma_0123_0 );

_mm256_storeu_pd( &gamma(0,1), gamma_0123_1 );

_mm256_storeu_pd( &gamma(0,2), gamma_0123_2 );

_mm256_storeu_pd( &gamma(0,3), gamma_0123_3 );
```

**Homework A.6.3.1** Complete the routine in Figure A.2. You will test it once you complete Homework A.6.4.2.

```
SEE ANSWERDO EXERCISE ON edX
```

#### A.6.4 MMM in terms of the kernel

Now that we have a routine for computing C := AB + C where C is  $4 \times 4$ , we can take advantage of MMM with submatrices to implement the general case. In this discussion, we are going to assume that the subproblem for which we have a kernel handles the case where C is  $m_r \times n_r$  (where the r refers to "register blocking"). For simplicity, we will assume that the general problem has row and column sizes m and n that are multiples of  $m_r$  and  $n_r$ , respectively.

Partition

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

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 all  $C_{i,j} := A_iB_j + C_{i,j}$ :

for 
$$j:=0,\ldots,N-1$$
 for  $i:=0,\ldots,M-1$  
$$C_{i,j}:=A_iB_i+C_{i,j} \hspace{1cm} \} \hspace{1cm} \textbf{Computed with GemmInstrinsicKernel\_m4xn4}$$
 end

This translates to the routine in Figure A.4.

**Homework A.6.4.2** Assuming you have completed Homework A.6.3.1 test the routine in Figure A.4, which calls the routine from that homework, with Live Script TestGemmJI\_Intrinsic.mlx.

SEE ANSWERDO EXERCISE ON edX

### A.7 Enrichment

# A.8 Wrapup

end

A.8.1 Additional exercises to edX

A.8.2 Summary to edX

A.8. Wrapup 35

```
#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
#define MR 4
#define NR 4
void GemmJI_IntrinsicsKernel( int m, int n, int k,
                             double *A, int ldA,
                             double *B, int ldB,
                             double *C, int ldC )
 int i, j;
 for ( int j=0; j<n; j+=NR ) /* n is assumed to be a multiple of NR */
   for ( int i=0; i<m; i+=MR ) /* m is assumed to be a multiple of MR */
     GEMMIntrinsicsKernel_m4xn4
        ( k, &gamma( i,j ), ldC, &alpha( i,0 ), ldA, &beta( 0,j ), ldB );
 return:
```

github/LAFF-On-PfHPC/Assignments/WeekA/C/GemmJI\_IntrinsicsKernel.c

Figure A.4: Simple blocked algorithm that calls the kernel that updates a  $4 \times 4$  submatrix of C. In the routine we use MR and NR so that it more easily generalizes to call kernels that update submatrices of size  $m_r \times n_r$ .

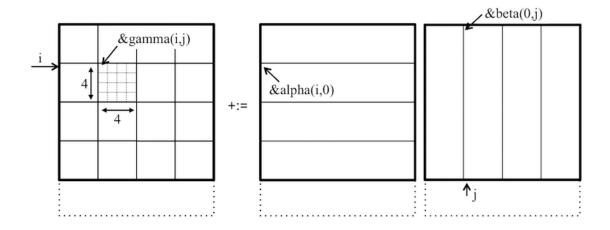


Figure A.5: Illustration of how the above routine indexes into the matrices.