Week A

Targeting a Single Core

A.1 Opening Remarks

A.1.1 Launch ☛ to edX

Homework A.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
☛ 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

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

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

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

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

A.1.2 Outline Week 1 ☛ to edX

A.1.3 What you will learn ☛ to edX

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

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

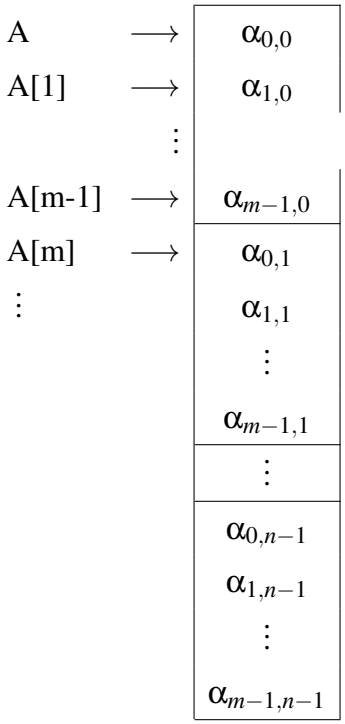
from the opener of this week. In memory, this may be stored in an array a by columns

$$\begin{array}{ll} \text{A} & \longrightarrow \\ \text{A[1]} & \longrightarrow \\ \text{A[2]} & \longrightarrow \\ \text{A[3]} & \longrightarrow \\ \vdots & \end{array} \begin{array}{|r|} \hline 1 \\ -1 \\ -2 \\ \hline -2 \\ 1 \\ 2 \\ \hline 2 \\ 3 \\ -1 \\ \hline \end{array}$$

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

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

Column-major ordering would store this in array a as



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 `ldA` 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:

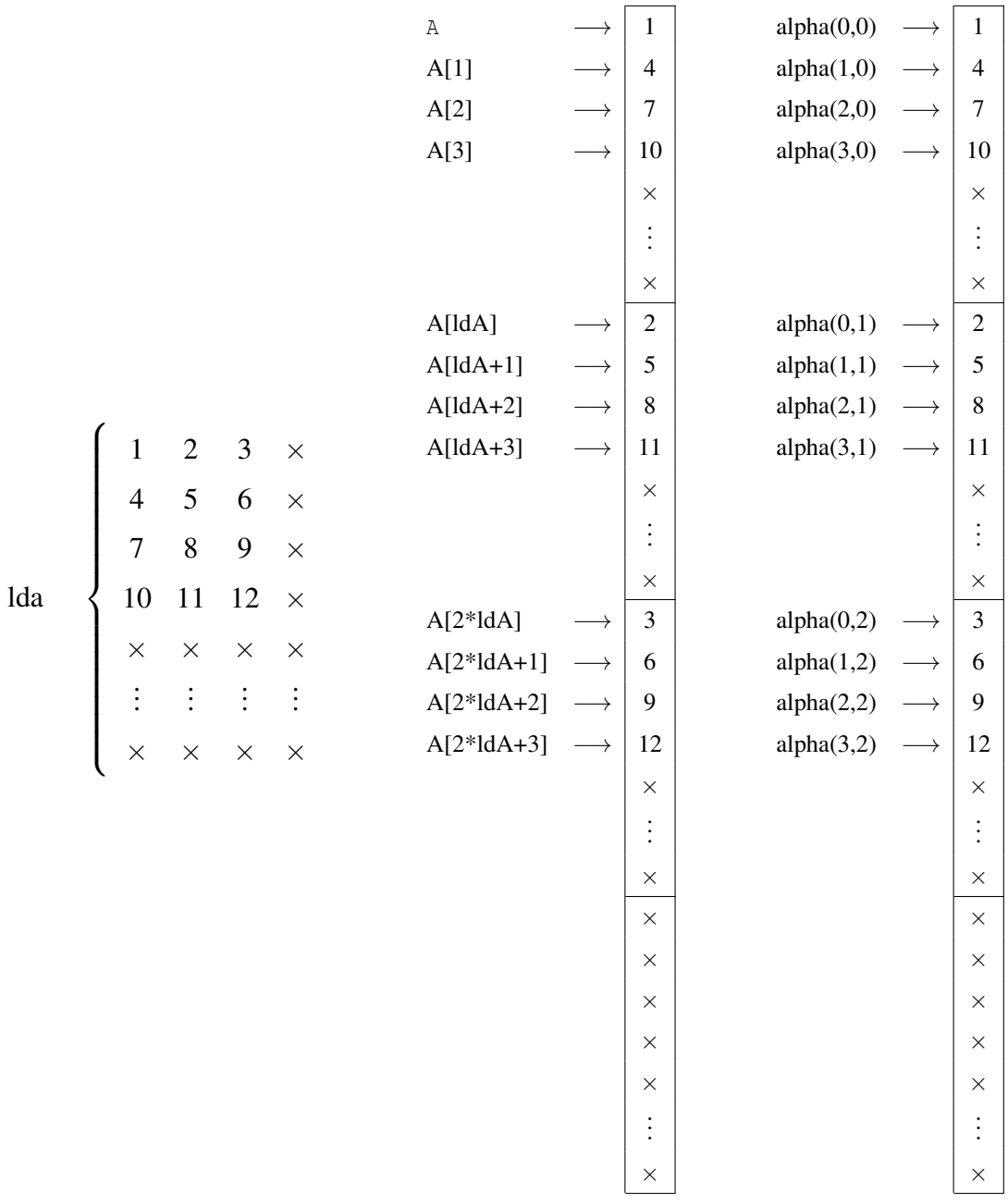
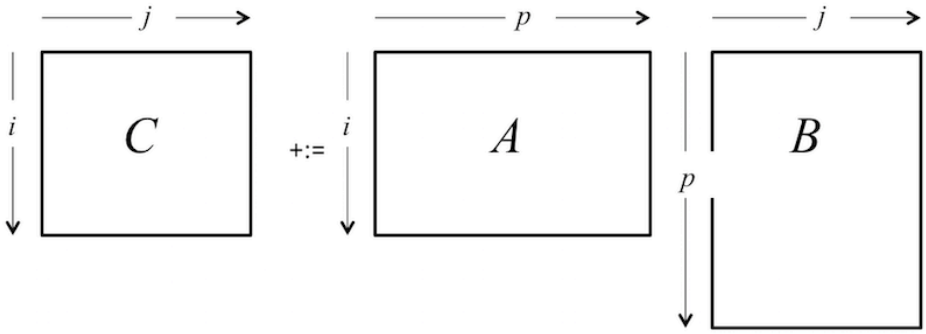


Figure A.1: Addressing a matrix embedded in an array with ldA rows. At the left we illustrate a 4×3 submatrix of a ldA $\times 4$ matrix. In the middle, we illustrate how this is mapped into an linear array a. In the right, we show how defining the C macro `#define alpha(i,j) A[ (j)*ldA + (i)]` allows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

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

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

```
#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 GemmIJP( int m, int n, int k,
        double *A, int ldA,
        double *B, int ldB,
        double *C, int ldC )
{
  int i, j, p;

  for ( i=0; i<m; i++ )
    for ( j=0; j<n; j++ )
      for ( p=0; p<k; p++ )
        gamma( i,j ) = alpha( i,p ) * beta( p,j ) + gamma( i,j );

  return;
}
```

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:

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

We then notice that

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

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

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

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

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

  end
end

.

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

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

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

  end
end

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

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*:

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

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

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

We then notice that

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

or, equivalently,

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

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

for $i := 0, \ldots, m-1$
  for $p := 0, \ldots, k-1$

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

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

  end
end

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

for $p := 0, \ldots, k-1$
  for $i := 0, \ldots, m-1$

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

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

  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:

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

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

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

and so forth.

A.3.3 The JPI and PJI orderings

Let's consider the ordering *JPI*:

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

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

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

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

so that

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

The algorithm captures this as

for $j := 0, \ldots, n-1$
  for $p := 0, \ldots, k-1$

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

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

  end
end

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

for $p := 0, \ldots, k-1$
  for $j := 0, \ldots, n-1$

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

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;
}
```

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

In this routine, `incx` and `incy` 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

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

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

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

  end
end

into the routine

```
#define A( i,j ) a[ (j)*lda + i ]   // map A( i,j ) to array a
#define B( i,j ) b[ (j)*ldb + i ]   // map B( i,j ) to array b
#define C( i,j ) c[ (j)*ldc + i ]   // map C( i,j ) to array c

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

void GemmIJP_Dots( int m, int n, int k,
        double *a, int lda,
        double *b, int ldb,
        double *c, int ldc )
{
  int i, j;

  for ( i=0; i<m; i++ )
    for ( j=0; j<n; j++ )
      Dots( k, &A( i,0 ), lda, &B( 0,j ), 1, &C( i,j ) );

  return;
}
```

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

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

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

$$\left(\begin{array}{c} \psi_0 \\ \hline \psi_1 \\ \hline \vdots \\ \hline \psi_{n-1} \end{array}\right) := \left(\begin{array}{c} \alpha\chi_0 + \psi_0 \\ \hline \alpha\chi_1 + \psi_1 \\ \hline \vdots \\ \hline \alpha\chi_{n-1} + \psi_{n-1} \end{array}\right)$$

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` , α 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`.

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

void Axpy( int n, double alpha, double *x, int incx, double *y, int incy )
{
  int i;

  for ( i=0; i<n; i++ )
    psi( i ) =                    ;

  return;
}
```

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 ANSWER

☛ DO EXERCISE ON edX

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

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

$\left.\begin{array}{l}\\\\\\\end{array}\right\}\ \widetilde{c}_i^T := \alpha_{i,p}\widetilde{b}_p^T + \widetilde{c}_i$

and illustrated by

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

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

Homework A.4.2.2 Complete the following routine

```
#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 Axpy( int n, double alpha, double *x, int incx, double *y, int incy )
   ;

void GemmIPJ_Axpy( int m, int n, int k,
       double *A, int ldA,
       double *B, int ldB,
       double *C, int ldC )
{
  int i, p;

  for ( i=0; i<m; i++ )
    for ( p=0; p<k; p++ )
       Axpy( n,    ,    ,    ,    ,    );

  return;
}
```

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 ANSWER

☛ DO 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

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

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

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

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

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

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

```
#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

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

void GemvJI_Axpy( int m, int n, double *A, int ldA, double *x, int incx,
                                double *y, int incy )
{
  int j;

  for ( j=0; j<n; j++ )
    Axpy( m, chi( j ), &alpha( 0,j ), 1, y, incy );

  return;
}
```

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

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

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

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

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

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

$\Big\}$ $\psi_i := \widetilde{a}_j^T x + \psi_i$

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

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

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

void GemvIJ_Dots( int m, int n, double *A, int ldA, double *x, int incx,
                                double *y, int incy )
{
  int i;

  for ( i=0; i<m; i++ )
    Dots( m, &alpha( i,0 ), ldA, x, incx, &psi( i ) );

  return;
}
```

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

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

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

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

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

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

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

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

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

} $a_j := \psi_i x + a_j$

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

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

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

void GerJI_Axpy( int m, int n, double *x, int incx, double *y, int incy,
                               double *A, int ldA )
{
  int j;

  for ( j=0; j<n; j++ )
    Axpy(    ,    ,    ,    ,    ,    );

  return;
}
```

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 ANSWER
☛ DO 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:

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

leading to the algorithm

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

$\Big\}$ $\widetilde{a}_i^T := \chi_i y^T + \widetilde{a}_i^T$

and implementation

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

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

void GerIJ_Axpy( int m, int n, double *x, int incx, double *y, int incy,
                               double *A, int ldA )
{
  int j;

  for ( j=0; j<n; j++ )
    Axpy( n, chi( i ),  y , incy, &alpha( i, 0 ), ldA );

  return;
}
```

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

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

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

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

for $p := 0, \ldots, k-1$

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

end
$\Big\}$ $c_j := Ab_j + c_j$

end

and

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

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

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

end
$\Big\}$ $c_j := Ab_j + c_j$

end

The second can then by translated into code as

```
#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 GemvIJ_Dots( int, int, double *, int, double *, int, double *, int );

void GemmJIP_Gemv( int m, int n, int k,
       double *A, int ldA,
       double *B, int ldB,
       double *C, int ldC )
{
  int j;

  for ( j=0; j<n; j++ )
    GemvIJ_Dots( m, k, A, ldA, &beta( 0,j ), 1, &gamma( 0,j ), 1 );

  return;
}
```

github/LAFF–On–PfHPC/Assignments/WeekA/C/GemmJIP_Gemv.c

Homework A.5.1.1 Complete the following routine

```
#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 GemvJI_Axpy( int, int, double *, int, double *, int, double *, int );

void GemmJPI_Gemv( int m, int n, int k,
       double *A, int ldA,
       double *B, int ldB,
       double *C, int ldC )
{
  int j;

  for ( j=0; j<n; j++ )
    GemvJI_Axpy(    ,    ,    ,    ,    ,    ,    ,    );

  return;
}
```

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 ANSWER
☛ DO 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

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

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

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

for $p := 0, \ldots, k-1$

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

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

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

$C := a_p\widetilde{b}_p^T + C$

end

and

for $p := 0, \ldots, k-1$

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

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

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

$C := a_p\widetilde{b}_p^T + C$

end

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

```
#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 GerJI_Axpy( int, int, double *, int, double *, int, double *, int );

void GemmPJI_Ger( int m, int n, int k,
      double *A, int ldA,
      double *B, int ldB,
      double *C, int ldC )
{
  int p;

  for ( p=0; p<k; p++ )
    GerJI_Axpy(    ,    ,    ,    ,    ,    ,    ,    );

  return;
}
```

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 ANSWER

☛ DO 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

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

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

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

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

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

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

end

end

and

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

for $p := 0, \ldots, k-1$

for $j := 0, \ldots, n-1$
$\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
end
$\left.\begin{array}{l}\\\\\end{array}\right\}$ $\widetilde{c}_i^T := \alpha_{i,p}\widetilde{b}_p^T + \widetilde{c}_i^T$ $\left.\begin{array}{l}\\\\\\\end{array}\right\}$ $\widetilde{c}_i^T := \widetilde{a}_i^T B + \widetilde{c}_i^T$

end

end

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:

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

- multiple rank-1 updates:

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

- multiple row times matrix multiplications:

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

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

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

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

Now, partition the matrices into submatrices:

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

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

with

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

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 4×4, A is $4 \times k$, and B is $k \times 4$. Now

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

$$+ := \left(\begin{array}{c|c|c} \alpha_{0,0} & \alpha_{0,1} & \cdots \\ \alpha_{1,0} & \alpha_{1,1} & \cdots \\ \alpha_{2,0} & \alpha_{2,1} & \cdots \\ \alpha_{3,0} & \alpha_{3,1} & \cdots \end{array}\right) \left(\begin{array}{cccc} \beta_{0,0} & \beta_{0,1} & \beta_{0,2} & \beta_{0,3} \\ \beta_{1,0} & \beta_{1,1} & \beta_{1,2} & \beta_{1,3} \\ \vdots & \vdots & \vdots & \vdots \end{array}\right)$$

$$= \left(\begin{array}{c|c|c|c} \alpha_{0,0}\beta_{0,0} + \alpha_{0,1}\beta_{1,0} + \cdots & \alpha_{0,0}\beta_{0,1} + \alpha_{0,1}\beta_{1,1} + \cdots & \alpha_{0,0}\beta_{0,2} + \alpha_{0,1}\beta_{1,2} + \cdots & \alpha_{0,0}\beta_{0,2} + \alpha_{0,1}\beta_{1,2} + \cdots \\ \alpha_{1,0}\beta_{0,0} + \alpha_{1,1}\beta_{1,0} + \cdots & \alpha_{1,0}\beta_{0,1} + \alpha_{1,1}\beta_{1,1} + \cdots & \alpha_{1,0}\beta_{0,2} + \alpha_{1,2}\beta_{1,2} + \cdots & \alpha_{1,0}\beta_{0,3} + \alpha_{1,3}\beta_{1,3} + \cdots \\ \alpha_{2,0}\beta_{0,0} + \alpha_{2,1}\beta_{1,0} + \cdots & \alpha_{2,0}\beta_{0,1} + \alpha_{2,1}\beta_{1,1} + \cdots & \alpha_{2,0}\beta_{0,2} + \alpha_{2,2}\beta_{1,2} + \cdots & \alpha_{2,0}\beta_{0,3} + \alpha_{2,3}\beta_{1,3} + \cdots \\ \alpha_{3,0}\beta_{0,0} + \alpha_{3,1}\beta_{1,0} + \cdots & \alpha_{3,0}\beta_{0,1} + \alpha_{3,1}\beta_{1,1} + \cdots & \alpha_{3,0}\beta_{0,2} + \alpha_{3,2}\beta_{1,2} + \cdots & \alpha_{3,0}\beta_{0,3} + \alpha_{3,3}\beta_{1,3} + \cdots \end{array}\right)$$

$$= \left(\begin{array}{c} \alpha_{0,0} \\ \alpha_{1,0} \\ \alpha_{2,0} \\ \alpha_{3,0} \end{array}\right) \left(\begin{array}{cccc} \beta_{0,0} & \beta_{0,1} & \beta_{0,2} & \beta_{0,3} \end{array}\right) + \left(\begin{array}{c} \alpha_{0,1} \\ \alpha_{1,1} \\ \alpha_{2,1} \\ \alpha_{3,1} \end{array}\right) \left(\begin{array}{cccc} \beta_{1,0} & \beta_{1,1} & \beta_{1,2} & \beta_{1,3} \end{array}\right) + \cdots$$

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

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

$$\left(\begin{array}{c|c|c|c} \gamma_{0,0} & \gamma_{0,1} & \gamma_{0,2} & \gamma_{0,3} \\ \gamma_{1,0} & \gamma_{1,1} & \gamma_{1,2} & \gamma_{1,3} \\ \gamma_{2,0} & \gamma_{2,1} & \gamma_{2,2} & \gamma_{2,3} \\ \gamma_{3,0} & \gamma_{3,1} & \gamma_{3,2} & \gamma_{3,3} \end{array}\right) += \left(\begin{array}{c} \alpha_{0,k} \\ \alpha_{1,k} \\ \alpha_{2,k} \\ \alpha_{3,k} \end{array}\right) \left(\begin{array}{cccc} \beta_{k,0} & \beta_{k,1} & \beta_{k,2} & \beta_{k,3} \end{array}\right)$$

endfor

or, equivalently to emphasize computations with vectors,

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

$$\left(\begin{array}{c|c|c|c} \gamma_{0,0} += \alpha_{0,p} \times \beta_{p,0} & \gamma_{0,1} += \alpha_{0,p} \times \beta_{p,1} & \gamma_{0,2} += \alpha_{0,p} \times \beta_{p,2} & \gamma_{0,3} += \alpha_{0,p} \times \beta_{p,3} \\ \gamma_{1,0} += \alpha_{1,p} \times \beta_{p,0} & \gamma_{1,1} += \alpha_{1,p} \times \beta_{p,1} & \gamma_{1,2} += \alpha_{1,p} \times \beta_{p,2} & \gamma_{1,3} += \alpha_{1,p} \times \beta_{p,3} \\ \gamma_{2,0} += \alpha_{2,p} \times \beta_{p,0} & \gamma_{2,1} += \alpha_{2,p} \times \beta_{p,1} & \gamma_{2,2} += \alpha_{2,p} \times \beta_{p,2} & \gamma_{2,3} += \alpha_{2,p} \times \beta_{p,3} \\ \gamma_{3,0} += \alpha_{3,p} \times \beta_{p,0} & \gamma_{3,1} += \alpha_{3,p} \times \beta_{p,1} & \gamma_{3,2} += \alpha_{3,p} \times \beta_{p,2} & \gamma_{3,3} += \alpha_{3,p} \times \beta_{p,3} \end{array}\right)$$

endfor

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 alpha_0123_p = _mm256_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 ) */
    gamma_0123_0 = _mm256_fmadd_pd( alpha_0123, beta_p_0, gamma_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 4×4.

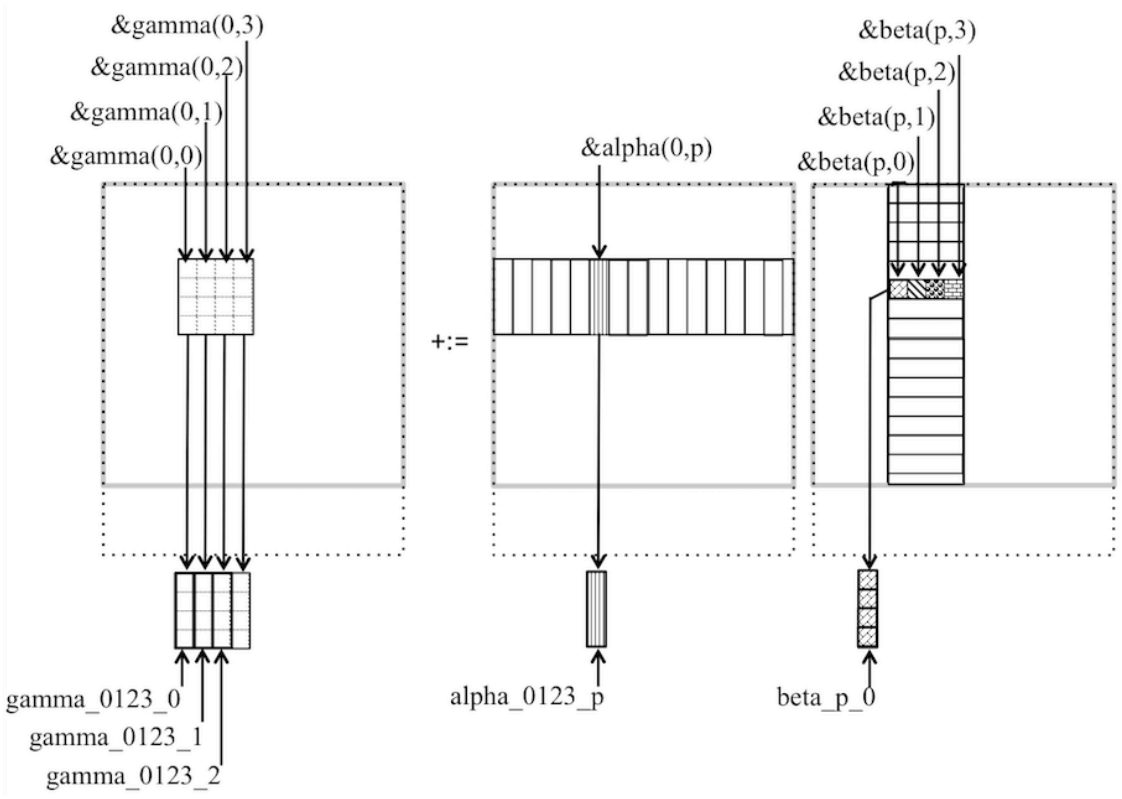


Figure A.3: Illustration of how the routine in Figure A.2 indexes into the matrices.

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

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

This is repeated for the other three columns of C.

The loop in Figure A.2 implements

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

$$\left(\begin{array}{c|c|c|c} \gamma_{0,0} += \alpha_{0,p} \times \beta_{p,0} & \gamma_{0,1} += \alpha_{0,p} \times \beta_{p,1} & \gamma_{0,2} += \alpha_{0,p} \times \beta_{p,2} & \gamma_{0,3} += \alpha_{0,p} \times \beta_{p,3} \\ \gamma_{1,0} += \alpha_{1,p} \times \beta_{p,0} & \gamma_{1,1} += \alpha_{1,p} \times \beta_{p,1} & \gamma_{1,2} += \alpha_{1,p} \times \beta_{p,2} & \gamma_{1,3} += \alpha_{1,p} \times \beta_{p,3} \\ \gamma_{2,0} += \alpha_{2,p} \times \beta_{p,0} & \gamma_{2,1} += \alpha_{2,p} \times \beta_{p,1} & \gamma_{2,2} += \alpha_{2,p} \times \beta_{p,2} & \gamma_{2,3} += \alpha_{2,p} \times \beta_{p,3} \\ \gamma_{3,0} += \alpha_{3,p} \times \beta_{p,0} & \gamma_{3,1} += \alpha_{3,p} \times \beta_{p,1} & \gamma_{3,2} += \alpha_{3,p} \times \beta_{p,2} & \gamma_{3,3} += \alpha_{3,p} \times \beta_{p,3} \end{array}\right)$$

endfor

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

`alpha_0123_p`, loading it with the contents of

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

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

```
gamma_0123_0 = _mm256_fmadd_pd( alpha_0123, beta_p_0, 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

```
_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 ANSWER
☛ DO 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 4×4, 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

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

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 $C_{i,j} := A_i B_j + C_{i,j}$:

for $j := 0, \ldots, N-1$

  for $i := 0, \ldots, M-1$

    $C_{i,j} := A_i B_i + C_{i,j}$ } Computed with `GemmInstrinsicKernel_m4xn4`

  end

end

This translates to the routine in Figure A.4.

```
#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 4×4 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 $m_r \times n_r$.

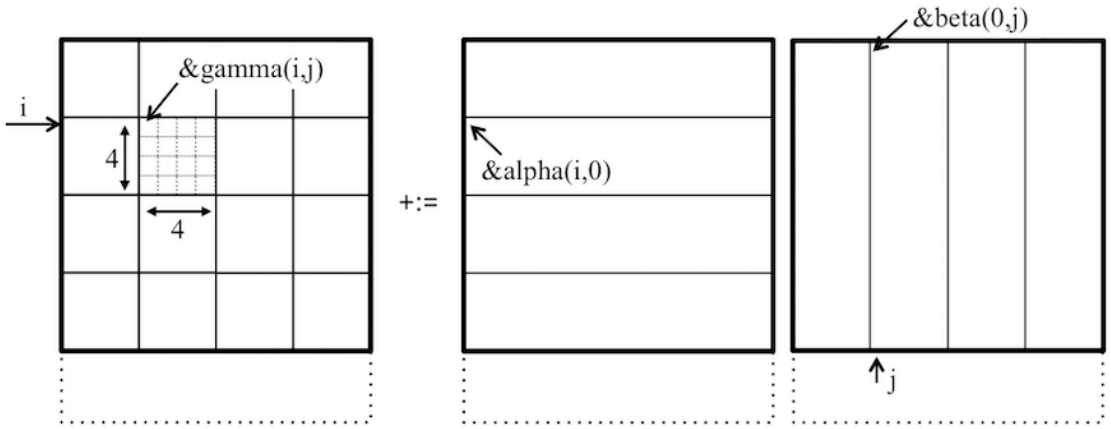


Figure A.5: Illustration of how the above routine indexes into the matrices.

Homework A.6.4.2 Assuming you have completed Homework **??**

```
#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 GerJI_Axpy( int, int, double *, int, double *, int, double *, int );

void GemmPJI_Ger( int m, int n, int k,
      double *A, int ldA,
      double *B, int ldB,
      double *C, int ldC )
{
  int p;

  for ( p=0; p<k; p++ )
    GerJI_Axpy(    ,    ,    ,    ,    ,    ,    ,    );

  return;
}
```

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 `TestGemmJI_Intrinsic.mlx`.

☛ SEE ANSWER

☛ DO EXERCISE ON edX

A.7 Enrichment

A.8 Wrapup

A.8.1 Additional exercises ☛ to edX

A.8.2 Summary ☛ to edX