Optimizing Serial Code Performance with Cache-aware Programming and BLAS

T. Yang. UCSB CS140, Winter 2026

UCSB

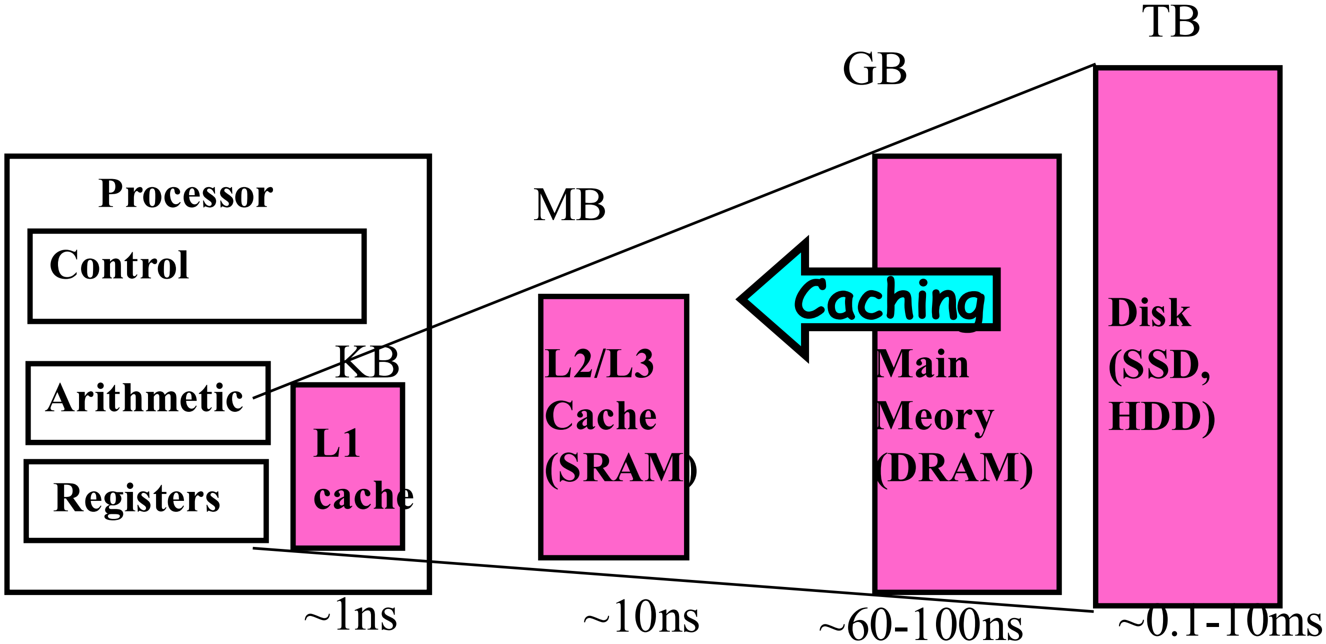
Topics

High performance computing on single cores

- SIMD vectorization on Intel/AMD CPUs
  – Covered in parallel architecture lecture
- Cache-aware optimization
- BLAS

Memory Hierarchy in Computer Systems

- Large performance impact when accessing data in different levels of memory hierarchy
- Cache-aware programming through program transformation is critical to maximize code efficiency



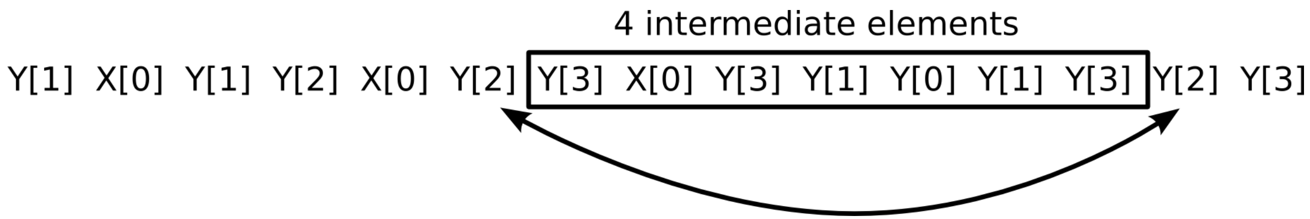
UCSB

Cache-Aware Programming: Temporal Locality

- **Exploit temporal locality in program**
  - Reuse an item that was previously accessed
- **Ex 1:** Y[2] is revisited continuously

```
For i=1 to n
   y[2]=y[2]+3
```

- **Ex 2 with access sequence:** Y[2] is revisited after a few instructions later

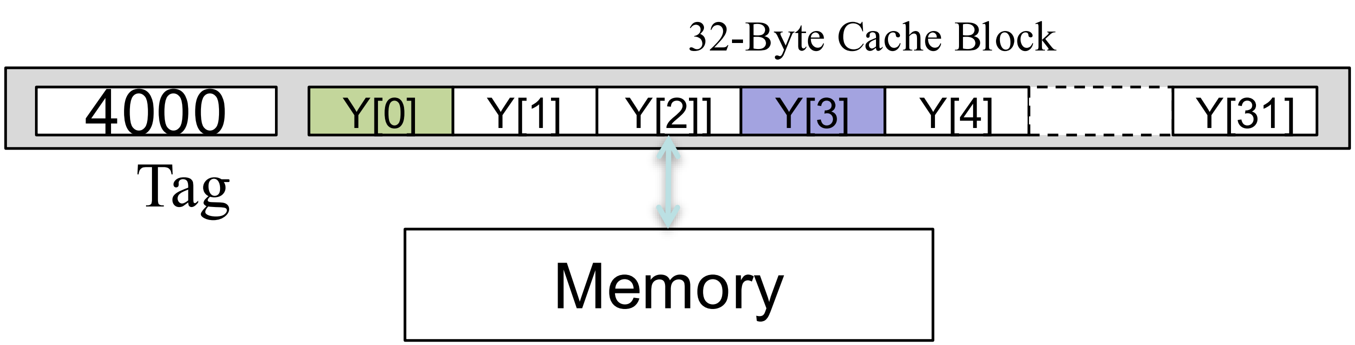


Cache-aware Programming: Spatial Locality

- Take advantage of better bandwidth by getting a chunk of memory to cache and use whole or part of chunk
- Exploit spatial locality in program
  - Access things nearby previous accesses

For i=1 to n
    y[i]=y[i]+3

Fetching Y[1] benefits next access of Y[2]



UCSB

Exploit spatial data locality in 2D array with a simple cache

- **Each cache block has 64 bytes. Cache has 128 bytes**
- **Program structure**
  - char D[64][64];
  - Each row is stored in one cache line block
  - **Program 1**

```
for (j = 0; j <64; j++)
    for (i = 0; i < 64; i++)
        D[i][j] = 0;
```

  - **Program 2**

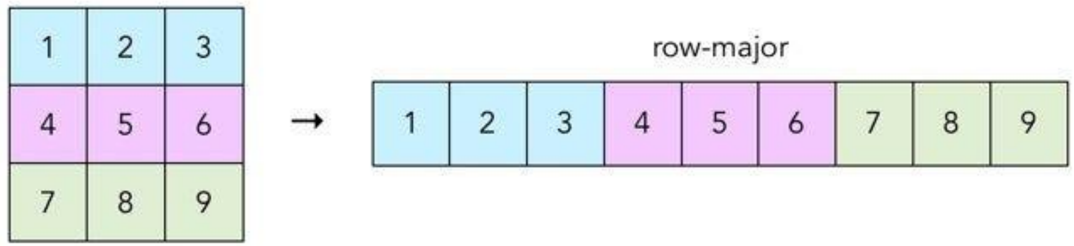
```
for (i = 0; i < 64; i++)
    for (j = 0; j < 64; j++)
        D[i][j] = 0;
```

64*64 data byte access → What is cache miss rate?

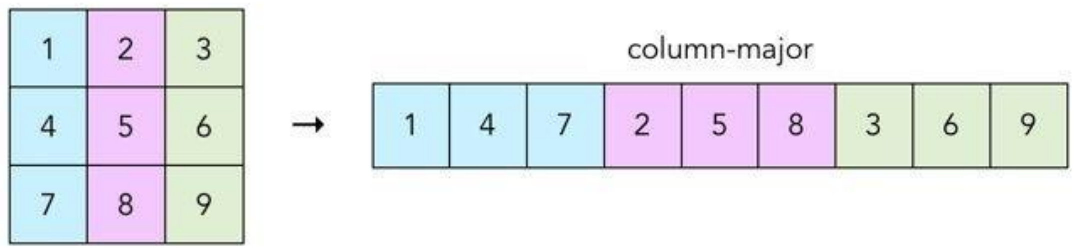
UCSB

Array layout in memory

- Default layout in C/C++ : Row major



- Alternative layout (e.g. BLAS library) column major



- A 2D matrix is 1D in memory addresses
- Use 1D array to implement 2D 3x3 array with row major

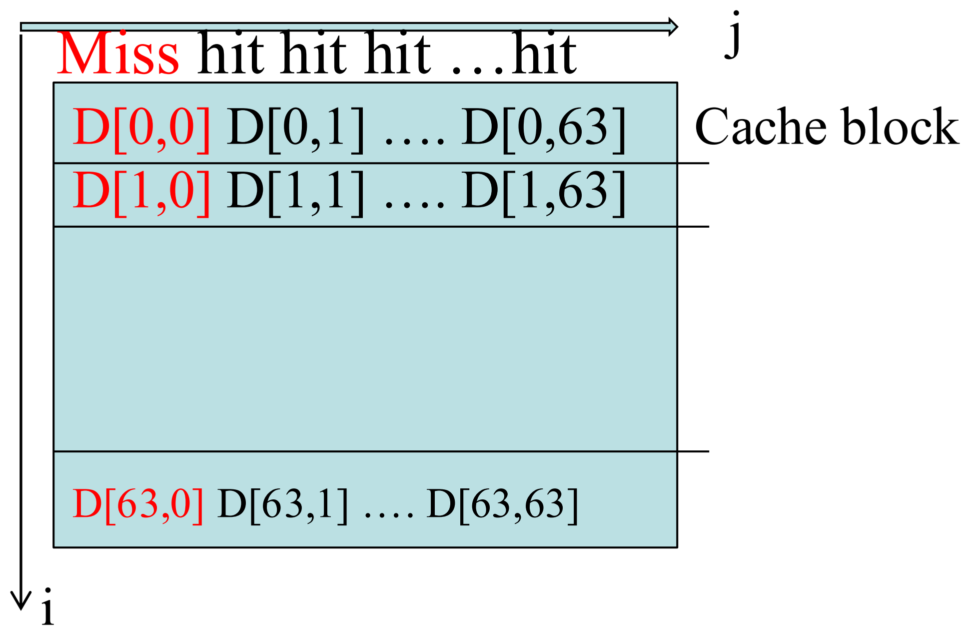
```
for(x = 0; x < 3; x++){
 for(y = 0; y < 3; y++) {
    array[3*x+y]=0;  //   Column major: array[x+3y]=0;
}
```

Data Access Pattern and Cache Miss

```
for (i = 0; i <64; i++)
   for (j = 0; j < 64; j++)
      D[i][j] = 0;
```

Each row is stored in one cache line block

1 cache miss in one inner loop iteration



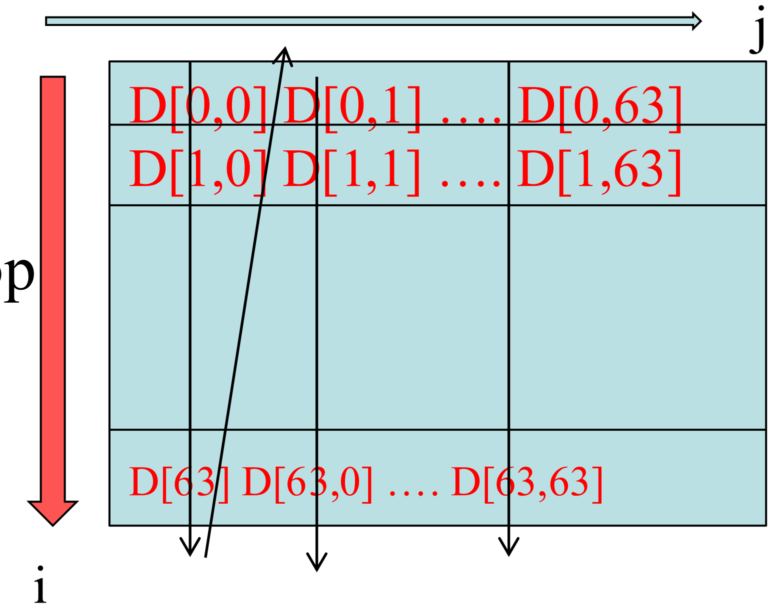
64 cache miss out of 64*64 access.

There is spatial locality. Fetched cache block is used 64 times before swapping out (consecutive data access within the inner loop

Data Locality and Cache Miss

```
• for (j = 0; j < 64; j++)
      for (i = 0; i < 64; i++)
          D[i][j] = 0;
```

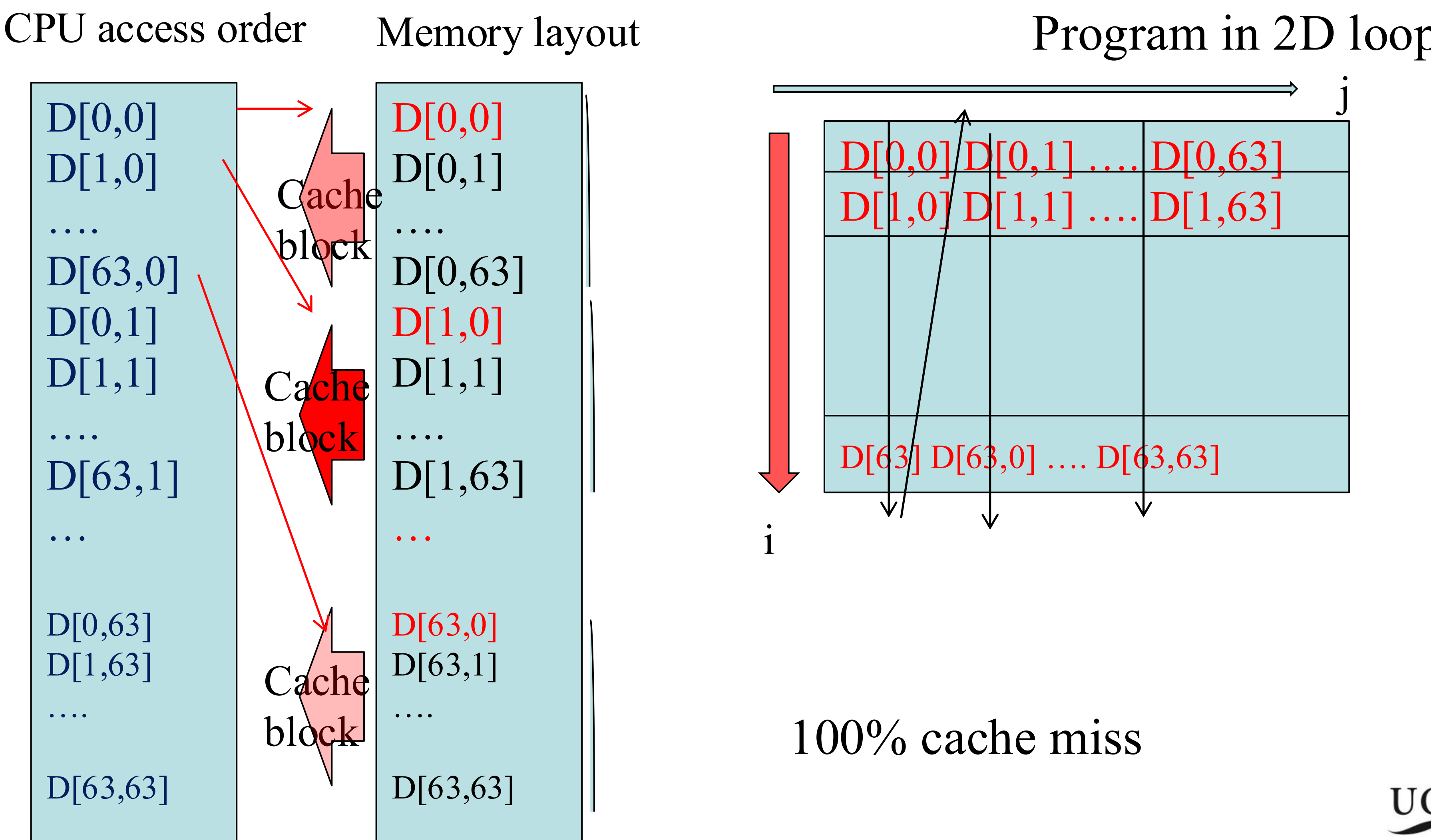
64 cache miss
in one inner loop
iteration



100% cache miss
There is no spatial locality. Fetched block is only used once before swapping out.

UCSB

Memory layout and data access by block



100% cache miss

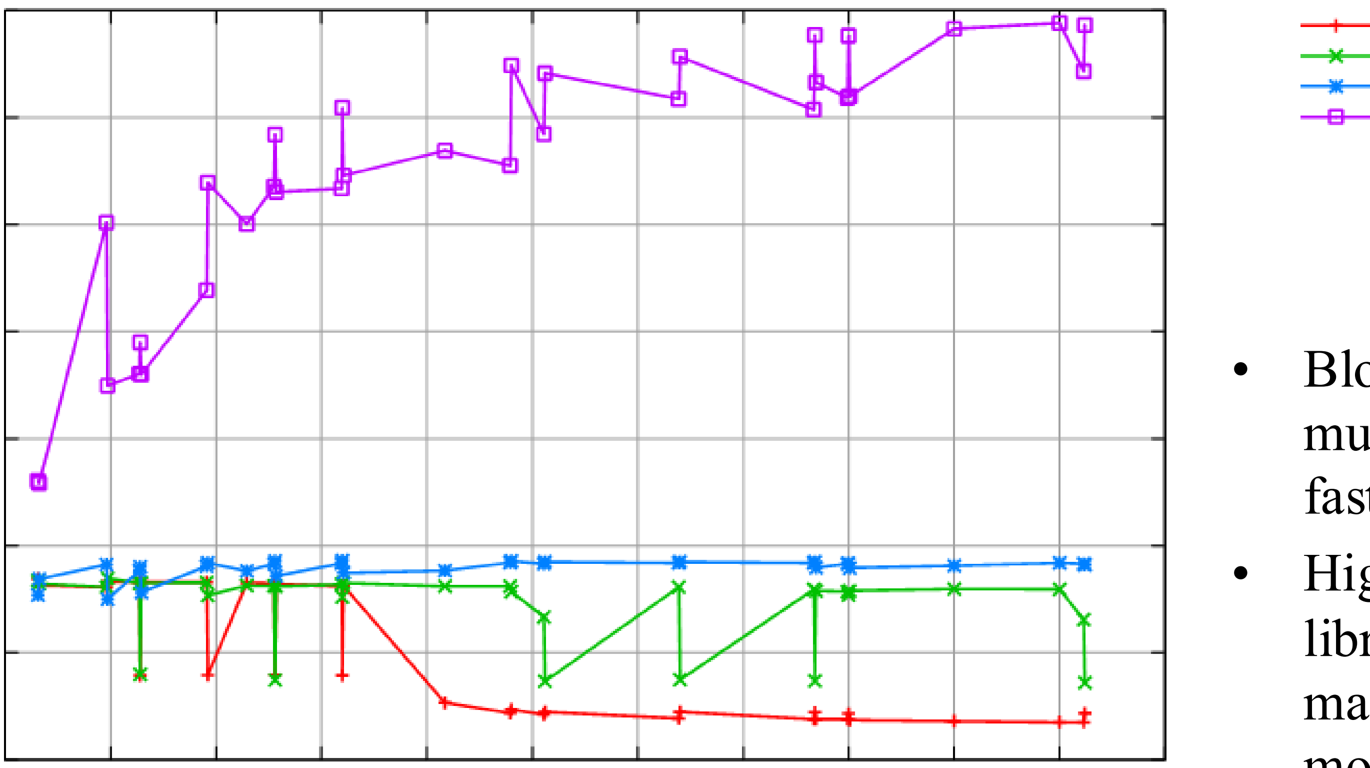
UCSB

Performance of Serial Matrix Multiply with Different Optimizations in FLOPS

Naïve 3 nested loop ~350MFLOPS

Green = simple blocking Upto 1700MFLOPS

DSB = Hand optimized code by David Bindel@Cornell



~6800MFLOPS

- Blocked matrix multiply: 2-4.8x faster naïve version
- High performance library in vendor machines with more optimization: 10-19x faster

FLOPS= #operations/time

Use a Simple Model of Memory to Explain and Optimize

- **Assume just 2 levels in the hierarchy: fast cache and slow memory**
- **All data initially in slow memory**
  - m = number of data elements moved between fast and memory
  - t_m = time of each element access from memory
  - f = number of arithmetic operations
  - t_f = time per arithmetic operation << t_m
  - q = f / m average number of flops per memory element access
- **Minimum possible time = f* t_f when all data in fast cache**
- **Actual time = computation cost + data fetch cost**

  $= f * t_f + m * t_m = f * t_f * (1 + t_m/t_f\ /q)$
- **Larger q → actual time closer to minimum f * t_f**

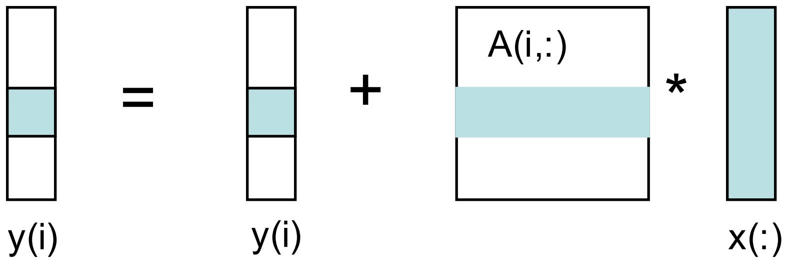
Computational Intensity: Key to algorithm efficiency

Machine Balance: Key to machine efficiency

Analysis for matrix-vector multiplication

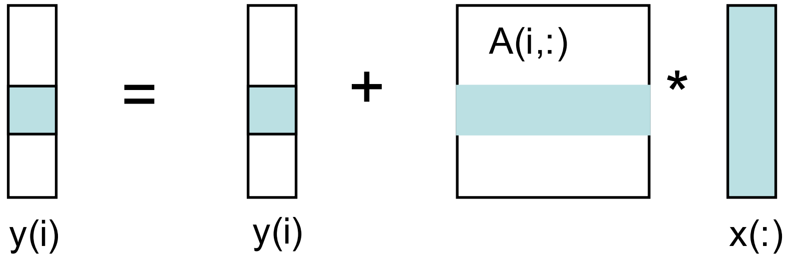
{Implements y = y + A*x}

```
for i = 1 to n
        for j = 1 to n
                y(i) = y(i) + A(i,j)*x(j)
```



Add memory-cache data movement

```
{Read vector x(1:n) into cache}
{Read vector y(1:n) into cache}
for i = 1 to n
        {Read row i of A into cache}
        for j = 1 to n
                y(i) = y(i) + A(i,j)*x(j)
{Write y(1:n) back to slow memory}
```

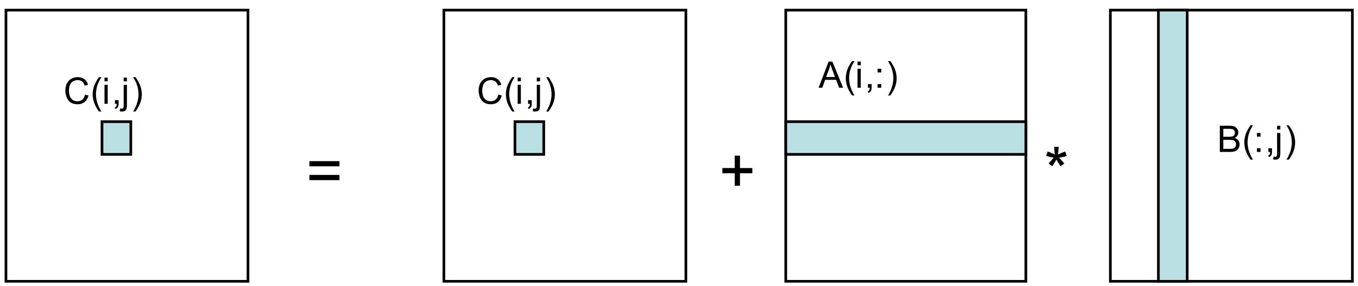


- m = number of slow memory refs = $3n + n^2$
- f = number of arithmetic operations = $2n^2$
- q = f / m ≈ 2   Low computational intensity
- **Running time** = $f * t_f + m * t_m$
- **FLOPS rate**=f/ Time =$1 / (t_f + t_m/q) = 1 / (t_f + t_m/2)$
- Matrix-vector multiplication limited by slow memory speed

Naïve Implementation for Matrix-Matrix Multiplication

```
{Implements C = C + A*B}
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

Inner loop is matrix-vector multiplication operations

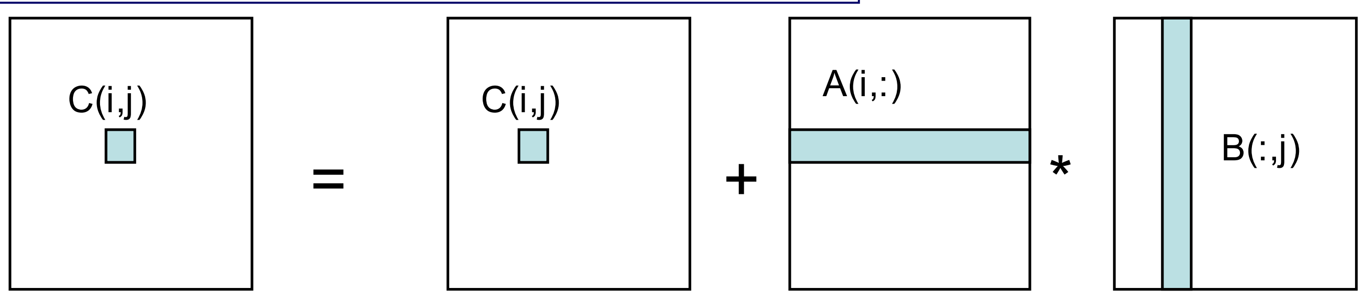


- Algorithm has $2*n^3$ operations and operates on $3*n^2$ words of memory
- Computational intensity q *potentially* as large as $2*n^3 / 3*n^2 = O(n)$
- But actual answer is not. $q \approx 2$ for large n, same as matrix-vector multiplication

Naïve Matrix Multiply with Memory-Cache Movement

```
{Implements C = C + A*B}
for i = 1 to n
  {Read row i of A into cache}
   for j = 1 to n
      {Read C(i,j) into cache}
      {Read column j of B into cache}
      for k = 1 to n
          C(i,j) = C(i,j) + A(i,k) * B(k,j)
      {Write C(i,j) back to slow memory}
```

Keep Row i of A in cache. Assume optimized cache replacement



Naïve Matrix Multiply

```
{Implements C = C + A*B}
for i = 1 to n
  {Read row i of A into cache}
  for j = 1 to n
     {Read C(i,j) into cache}
     {Read column j of B into cache}
     for k = 1 to n
        C(i,j) = C(i,j) + A(i,k) * B(k,j)
     {Write C(i,j) back to memory}
```

# of slow memory ops:

$m = n^3$ to read each column of B n times

$+ n^2$ to read each row of A once

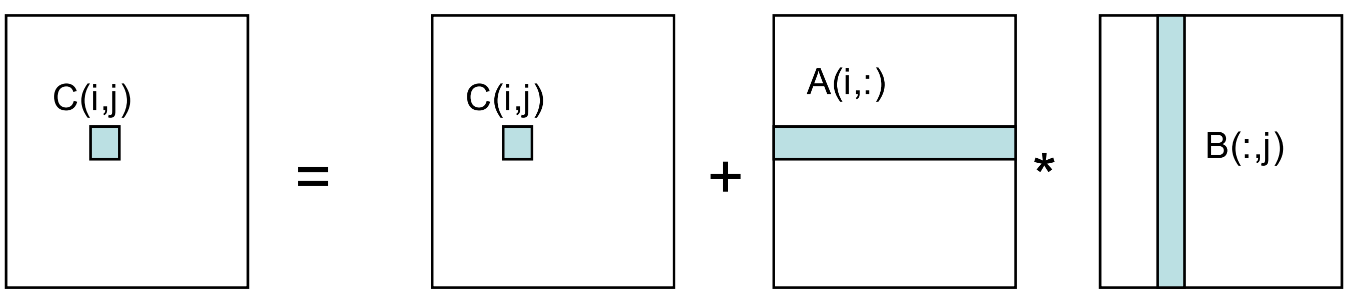
$+ 2n^2$ to read and write each element of C once

$= n^3 + 3n^2$

So $q = f / m = 2n^3 / (n^3 + 3n^2)$ = computational intensity

≈ 2 for large n, no improvement over matrix-vector multiply

Reason: Inner two loops are just matrix-vector multiply, of row i of A times matrix B



Better Implementation with Blocked Matrix Multiplication

- **Example of submartix partitioning:** Divide A into 4 submatrices

$$A = \left(\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right) \Longrightarrow \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

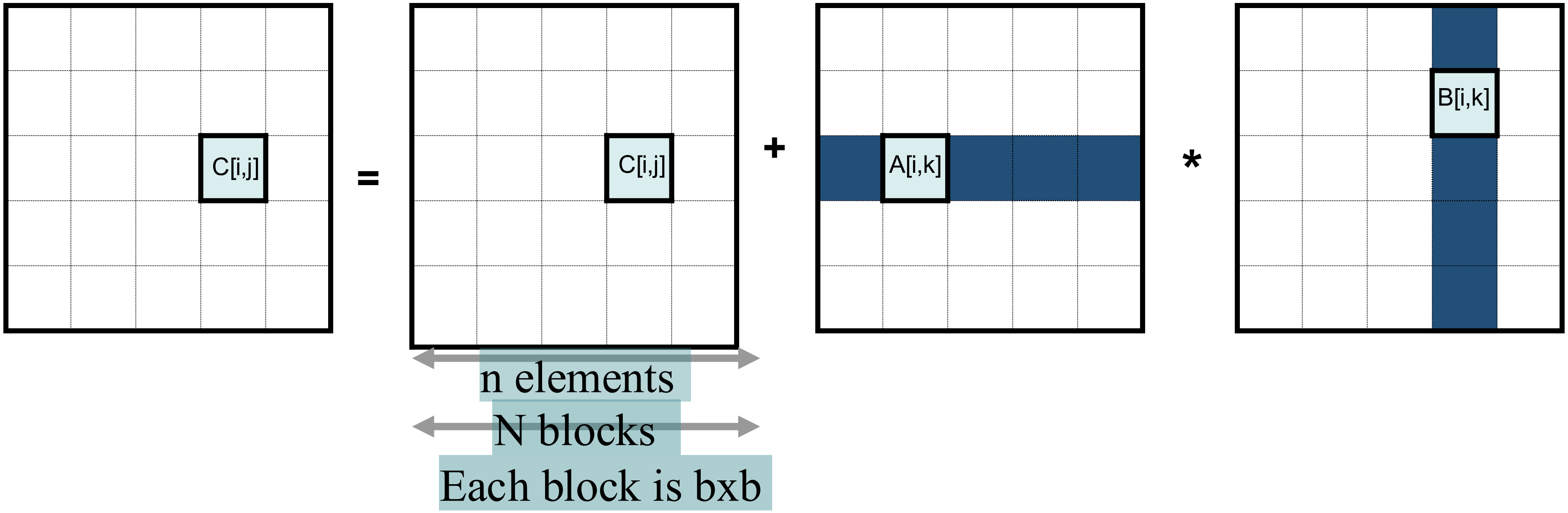
$$A_{11} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, A_{12} = \begin{pmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{pmatrix}$$

$$A_{21} = \begin{pmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix}, A_{22} = \begin{pmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{pmatrix}$$

- **Blocked matrix multiply:** Element-wise multiply is submatrix multiply

Blocked [Tiled] Matrix Multiply

Consider A,B,C to be N-by-N matrices of b-by-b blocks
where b=n / N is called the block size



Blocked (Tiled) Matrix Multiply with Six-Nested Loops

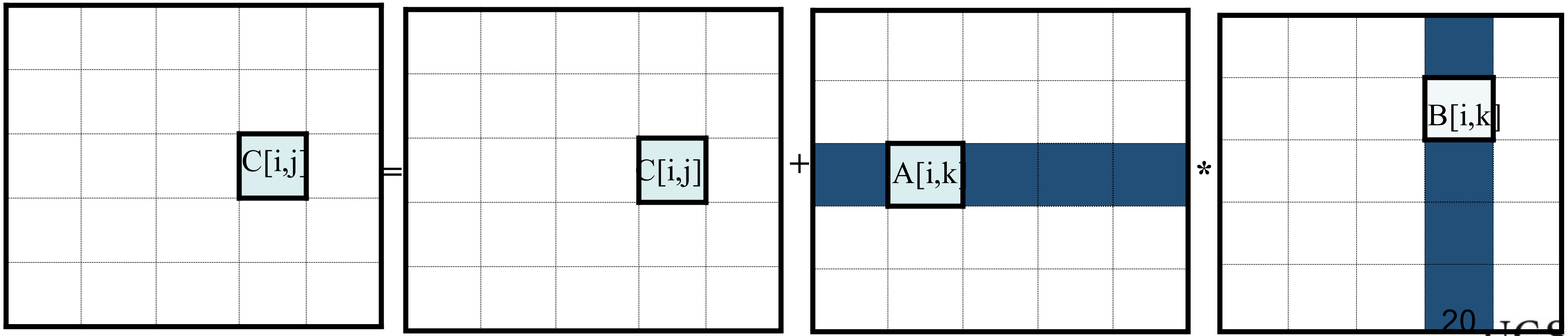
Consider A,B,C to be N-by-N matrices of b-by-b blocks

Each element is a block

b=n / N is called the block size

```
for i = 1 to N
  for j = 1 to N
    for k = 1 to N
        C(i,j) = C(i,j) + A(i,k) * B(k,j) //  block submatrix multiply
```

3 nested loops inside



Blocked (Tiled) Matrix Multiply with Memory-Cache Data Movement

Consider A,B,C to be N-by-N matrices of b-by-b blocks where

b=n / N is called the block size

for i = 1 to N

for j = 1 to N

{Read block C(i,j) into cache}

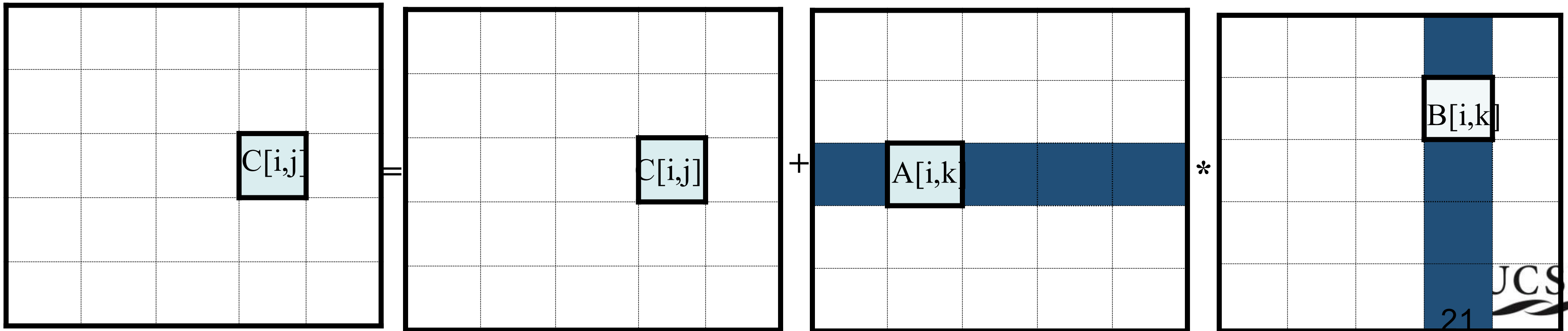
for k = 1 to N

{Read block A(i,k) into cache}

{Read block B(k,j) into cache}

C(i,j) = C(i,j) + A(i,k) * B(k,j)// Block submatrix multiply

{Write block C(i,j) back to slow memory}



UCSB

Blocked (Tiled) Matrix Multiply with Memory-Cache Data Movement

A,B,C to be N-by-N matrices of b-by-b blocks

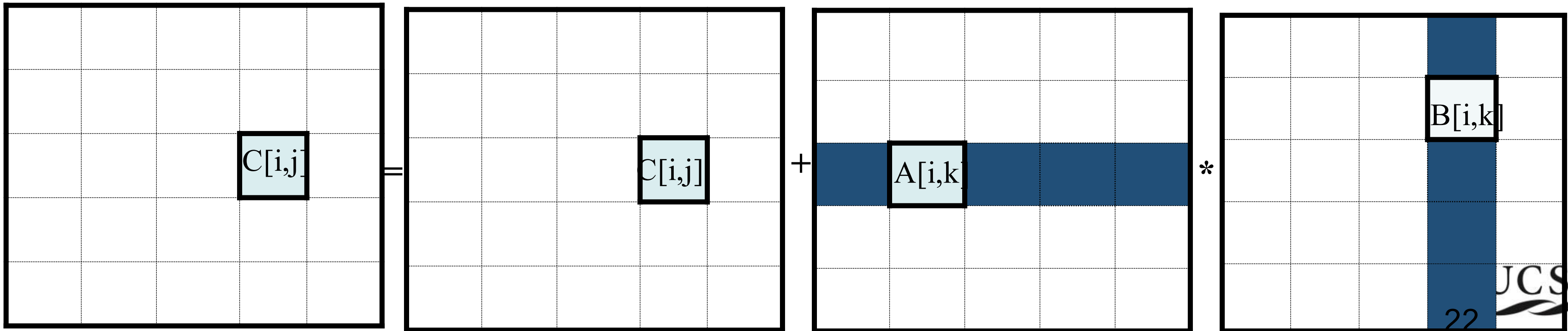
b=n / N is called the block size

```
for i = 1 to N
  for j = 1 to N
    {Read block C(i,j) into cache}
    for k = 1 to N
        {Read block A(i,k) into cache}
        {Read block B(k,j) into cache}
        C(i,j) = C(i,j) + A(i,k) * B(k,j)
    {Write block C(i,j) back to memory}
```

$2n^2$ to read/write each block of C once

$N*n^2$ to read each block of A N^3 times ($N^3 *b^2 = N^3 *(n/N)^2$)

$N*n^2$ to read each block of B N^3 times



Blocked (Tiled) Matrix Multiply

Recall:

m is amount memory traffic between memory and cache

matrix has nxn elements, and NxN blocks each of size bxb

f is number of floating point operations, $f=2n^3$

q = f / m is our measure of memory access efficiency

So: #slow memory access

$m = N*n^2$ read each block of B N^3 times ($N^3 * b^2 = N^3 * (n/N)^2 = N*n^2$)

$+ N*n^2$ read each block of A N^3 times

$+ 2n^2$ read and write each block of C once

$= (2N + 2) * n^2$

So computational intensity $q = f / m = 2n^3 / ((2N + 2) * n^2)$

$\approx n / N = b$ for large n

So we can improve performance by increasing the block size b

Blocked version can be much faster than naïve version which has q=2

UCSB

Block Size Limited by Cache Size & Takeaways

Blocked matrix multiply has computational intensity $q \approx b$

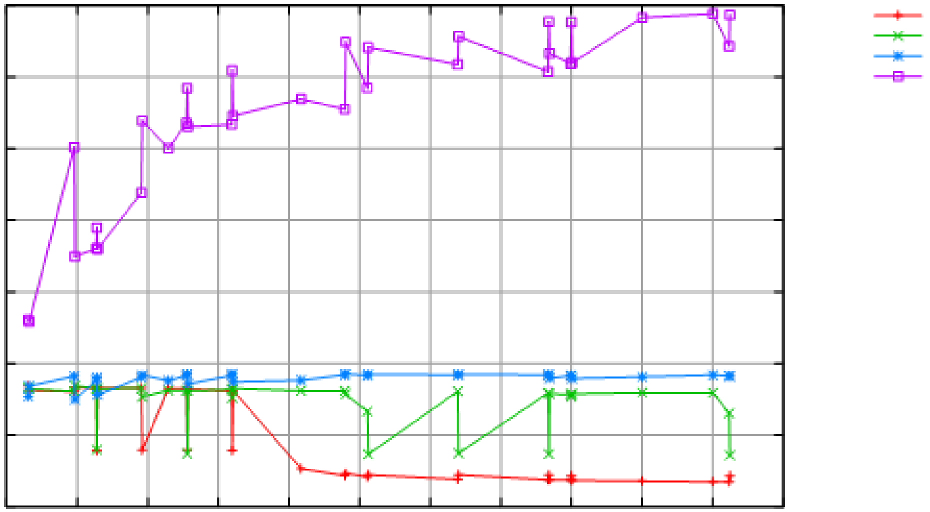
- Larger the block size → more efficient
- Limit: All three blocks from A,B,C must fit in cache
- Assume L1 cache has size M_{size}

$$3b^2 \le M_{size}, \quad \text{so} \quad q \approx b \le (M_{size}/3)^{1/2}$$

- Assume L1 cache has size 32KB, $b \le 104$

Takeaways from this figure:

- Blocked matrix multiply: 2-4.8x faster than naïve version
- BLAS library from vendors with more optimization: 10-19x faster



Basic Linear Algebra Subroutines (BLAS)

- Industry standard interface: **www.netlib.org/blas**
- **Vendors supply optimized BLAS** implementations
  - **BLAS1**: Vector operations: dot product, saxpy (y=a*x+y), etc
    - m=2*n, f=2*n, low computational density ~1 or less
  - **BLAS2**
    - E.g. Matrix-vector multiplication. m=n^2, f=2*n^2
    - Moderate computational density~2
    - Computation expressed with BLAS2 can be faster than BLAS1
  - **BLAS3**
    - E.g. Matrix-matrix multiplication with m <= O(n^2), f=O(n^3)
    - Higher computational density > 2
- **Applications may be expressed a mixed set of BLAS1, BLAS2, or BLAS3 operations**

UCSB

GEMM and GEMV in Intel/NVIDIA BLAS Libraries

- Intel Math Kernel Library (**MKL**) for Intel CPUs and GPUs, and it works on AMD CPUs (e.g. CPU servers on Expanse)
  - cblas_sgemm, cblas_dgemm, sgemm, dgemv
- **cuBLAS** : NVIDIA-optimized implementation for use with **CUDA** on its GPUs.
  - cublasSgemm, cublasDgemm, cublasSgemv, cublasDgemv
- API of MKL and cuBLAS is almost identical

SGEMM (single-precision general matrix-matrix multiplication) and **DGEMM** for double-precision: $\mathbf{C=\alpha \cdot op(A) \cdot op(B)+\beta \cdot C}$

- A, B, and C are M*K, K*N, M*N matrices.
- op(X) can be X (no transpose), X^T (transpose)
- α and β are scalar coefficients.

SGEMV and **DGEMV** for matrix vector multiplication:

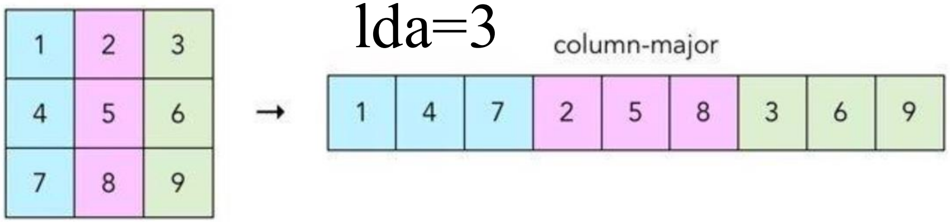
$\mathbf{y = \alpha \cdot op(A) \cdot x + \beta \cdot y}$

- **x** and **y** are column vectors of size K.

DGEMV function in MKL: y =α·op(A)·x+β·y

- A is M*K matrix. **x** and **y** are column vectors of size K.
- op(A) can be A (no transpose), A^T (transpose)
- α and β are scalar coefficients

lda=3



void **cblas_dgemv**(

CblasColMajor or CblasRowMajor //Choose CblasColMajor

CblasNoTrans or CblasTrans, // no transpose or transpose of A

MKL_int M, MKL_int K,

double alpha, double *A, MKL_int *lda,

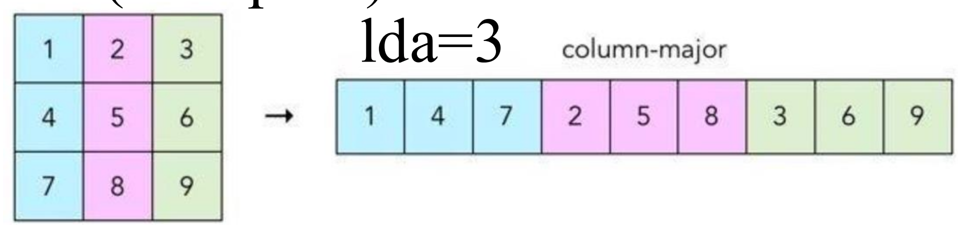
double *x, MKL_int incx,

double beta, double *y, MKL_int incy);

incx, incy: Stride(increment) of next element in vectors x and y. Normally choose 1.

DGEMM function in MKL: C=α·op(A)·op(B)+β·C

- A, B, and C are single-precision M*K, K*N, M*N matrices.
- op(X) can be X (no transpose), X^T (transpose)
- α and β are scalar coefficients




```
void  cblas_gemm(
  CblasColMajor or CblasRowMajor //Choose CblasColMajor
  CblasNoTrans or CblasTrans,  //   no transpose or transpose of A
  CblasNoTrans or CblasTrans,  //   no transpose or transpose of B
  MKL_int M,  MKL_int N, MKL_int K,
  double alpha, double *A,   MKL_int lda,
  double  *B,  MKL_int ldb,
  double  beta, double *C,  MKL_int ldc);
```

lda, ldb, ldc: Leading dimensions of A, B, and C as # of elements between the start of successive columns (Column-Major)

Use of GEMV for GEMM Implementation

- Matrix-matrix multiplication with size N*N can be can be expressed as N matrix-vector multiplications. For example, N=2

$$A=\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}, B = \begin{bmatrix}1 & 3\\ 2 & 4\end{bmatrix}$$

$$C = AB \rightarrow C = \begin{bmatrix}3 & 7\\ 3 & 3\end{bmatrix}$$

Decomposed

$$\begin{bmatrix}3\\ 3\end{bmatrix} = A*\begin{bmatrix}1\\ 2\end{bmatrix}$$

$$\begin{bmatrix}7\\ 3\end{bmatrix} = A*\begin{bmatrix}3\\ 4\end{bmatrix}$$

- In general, a computing problem may be expressed by
  - a set of BLAS-1 operations
  - or BLAS-2 operations
  - or BLAS-3 operations
  - or mixed of all levels

Concluding Remarks

To optimize serial code efficiency

- **Cache-aware programming** to exploit spatial and temporal locality
- It is recommended to **use fully optimized vendor's or open-source BLAS library functions** for time-consuming core scientific computation
  - Compare FLOPS difference when code can use different levels of BLAS
  - For larger problem sizes, BLAS3 is faster with cache optimization and SIMD vectorization
  - BLAS has calling overhead while unoptimized code may fit in cache well for small problem sizes

Other serial code optimization strategies discussed earlier

- Use compiler optimization level as high as possible
- SIMD vectorization on Intel/AMD CPUs if compiler cannot vectorize serial code well