Linear Algebra in C++

Lecture 8

Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4700/

Vector Spaces

Mathematical Vector Space

- Definition (Halmos): A vector space is a set V of elements called *vectors* satisfying the following axioms:
  1. To every pair x and y of vectors in V there corresponds a vector $x + y$ called the *sum* of x and y in such a way that:
     a) Addition is commutative: $x + y = y + x$
     b) Addition is associative: $x + (y + z) = (x + y) + z$
     c) There exists in V a unique vector 0 (the origin) such that $x + 0 = x$ for every x
     d) To every vector x in V there corresponds a unique vector $-x$ such that $x + (-x) = 0$
  2. To every pair of a and x, where a is a scalar and x is a vector in V, there corresponds a vector ax in V called the product of a and x in such a way that:
     a) Multiplication by scalars is associative: $a(bx) = (ab)x$
     b) For every vector x the following holds: $1x = x$
  3. Multiplication by scalars is distributive with respect to vector addition:
     $$a(x + y) = ax + ay$$
  4. Multiplication by vectors is distributive with respect to scalar addition:
     $$(a + b)x = ax + bx$$

Mathematical Vector Space Examples

- Math calls this a
  - Additive Group
  - With the idempotent element of 0
  - With an invertive element of -1

- Examples
  - Set of all integer numbers
  - Set of all complex numbers
  - Set of all polynomials
  - Set of all n-tuples of real numbers

$$R^N$$

The vector space used in scientific computing

Computer Representation of Vector Space

- In the bad old days, vectors were represented as arrays

```
REAL X(N)
REAL Y(N)
```

- Add them `CALL SAXPY(N, ALPHA, X, Y)` $Y \leftarrow \alpha X + Y$

- Double precision

```
DOUBLE X(N)
DOUBLE Y(N)
```

- Add them `CALL DAXPY(N, ALPHA, X, Y)` $Y \leftarrow \alpha X + Y$

```
for (int i = 0; int < N; ++i) y[i]  = alpha * x[i];
```

Vectors Spaces in C++

- Despite the clumsiness of Fortran interface (or maybe because of it) the performance of vector operations was quite good
- In C/C++, there are numerous options for vectors (and matrices)

```
double x[N];
    // Not dynamically sizable*

double* x = (double*) malloc(N * sizeof(double));
    // memory management hell

std::vector<double> x(N);
    // Limited to interface of vector<double> (not a 'vector')

Vector x(N);
    // Just right, has proper interface and implementation
```

Vectors Spaces in C++

- Despite the clumsiness of Fortran interface (or maybe because of it) the performance of vector operations was quite good
- In C/C++, there are numerous options for vectors (and matrices)

```
double a[N][M];
    // Not dynamically sizable

double** x = ??;
    // Really easy to get bad performance

std::vector<std::vector<double>> x(N);
    // Not a matrix (or a 2D array for that matter) at all

Matrix<double> x(N, M);
    // Just right, has proper interface and implementation
```

Classes (Types)

- First principles: Abstraction, simplicity, consistent specification

- Domain: Scientific computing
- Domain abstractions: Matrices and vectors
- Programming abstractions: Matrix type and Vector type

- C++ types enable encapsulation of related data and functions
  - Provide visible interfaces
  - Hide implementation details

std::vector<double>

- Before rushing off to implement fancy interfaces
- Understand what we are working with
- And how hardware and software interact
- std::vector<double> will be our storage
- But its interface won't be our interface
  - We will use types defined in Blaze
  - But we will look at implementation for those

Linear Algebra Operations

Vector Dot Product

- Multiplication of 2 vectors followed by Summation

A[i]
X_1
X_2
X_3
X_4
X_5
… …
X_n

$\cdot$

B[i]
Y_1
Y_2
Y_3
Y_4
Y_5
… …
Y_n

$$= \sum_{i=1}^{n}$$

A[i] * B[i]
$X_1 * Y_1$
$X_2 * Y_2$
$X_3 * Y_3$
$X_4 * Y_4$
$X_5 * Y_5$
… …
$X_n * Y_n$

Vector Dot Product

```cpp
int main()
{
    std::vector<double> a = {1.0, 2.0, 3.0, 4.0, 5.0};
    std::vector<double> b = {5.0, 4.0, 3.0, 2.0, 1.0};

    double dot_result = 0.0;
    for (size_t i = 0; i != a.size(); ++i)
        dot_result += a[i] * b[i];

    std::println("dot-product: {}", dot_result);
}
```

Matrix-Vector Multiplication

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_{11}b_1 + a_{12}b_2 + \dots & a_{1n}b_n \\ a_{21}b_1 + a_{22}b_2 + \dots & a_{2n}b_n \\ \vdots \\ a_{m1}b_1 + a_{m2}b_2 + \dots & a_{mn}b_n \end{bmatrix}$$

- Result is a vector that holds the dot-products of each line of the matrix with the given right-hand-side vector

Matrix in Math

- Matrix is an array of numbers, e.g., 2 rows by 2 columns $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$
- Add two matrices: C = A + B

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 5 \\ 5 & 5 \end{bmatrix}$$

- Subtract two matrices: C = A – B

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

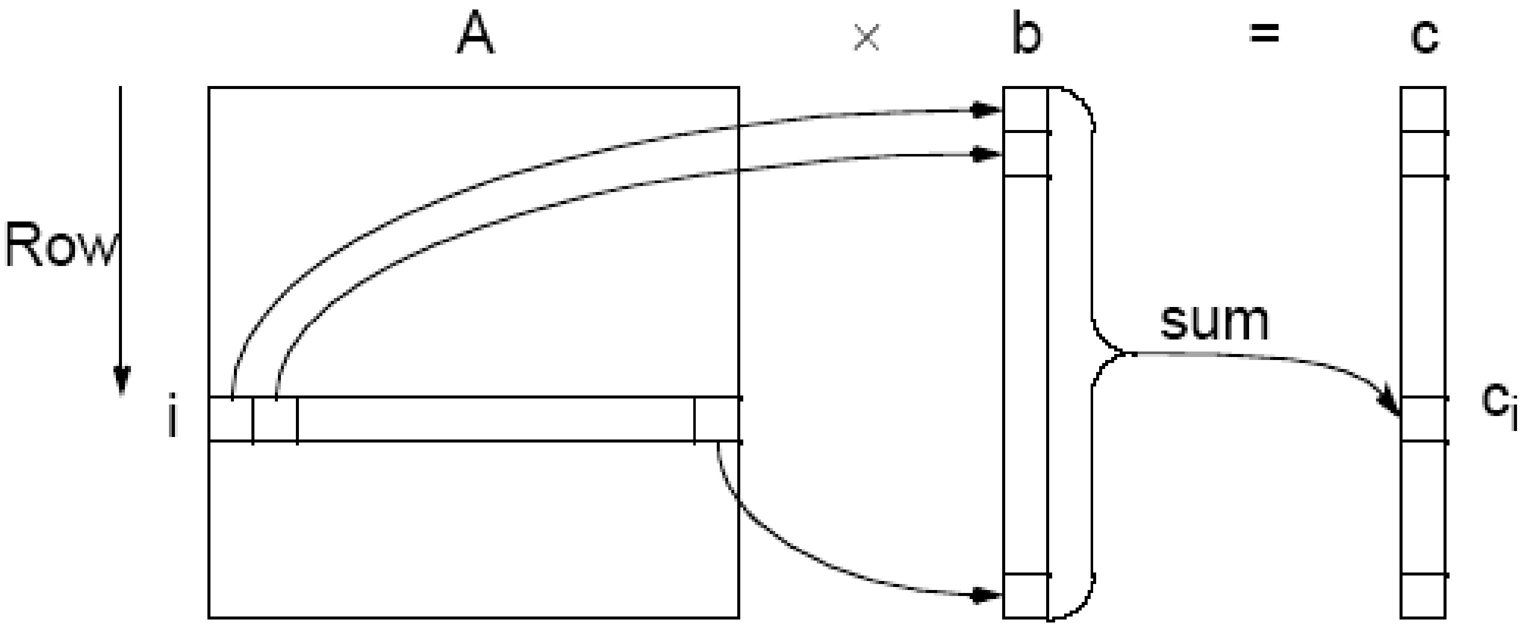
- Multiply one matrix by another matrix: C = A × B
- Transpose a matrix: $A \rightarrow A^T$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

Matrix-Matrix Multiplication

$$C = A \times B$$

$$C_{ij} = \sum_k A_{ik} B_{kj}$$



Matrix Transpose

- The transpose of the (m x n) matrix *A* is the (n x m) matrix formed by interchanging the rows and columns such that row i becomes column i of the transposed matrix

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \qquad \boldsymbol{A}^T = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & & a_{m2} \\ \vdots & & \ddots & \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{pmatrix}$$

$$\mathbf{A} = \begin{bmatrix} 1 & 3 \\ 2 & 5 \end{bmatrix} \quad \boldsymbol{A}^T = \begin{bmatrix} 1 & 2 \\ 3 & 5 \end{bmatrix} \qquad \boldsymbol{A} = \begin{bmatrix} 1 & 3 & 4 \\ 0 & 1 & 0 \end{bmatrix} \quad \boldsymbol{A}^T = \begin{bmatrix} 1 & 0 \\ 3 & 1 \\ 4 & 0 \end{bmatrix}$$

Matrix Representation

Matrix Representation in Computers

- Two issues
  - Interface (what is the abstraction we want to present?)
  - Implementation (how is the abstraction realized?)
- Sometimes there are tradeoffs
  - Evaluate relative to end user
  - In HPC – performance is most important
  - Elsewhere – safety, ease of use, standards compliance, etc.

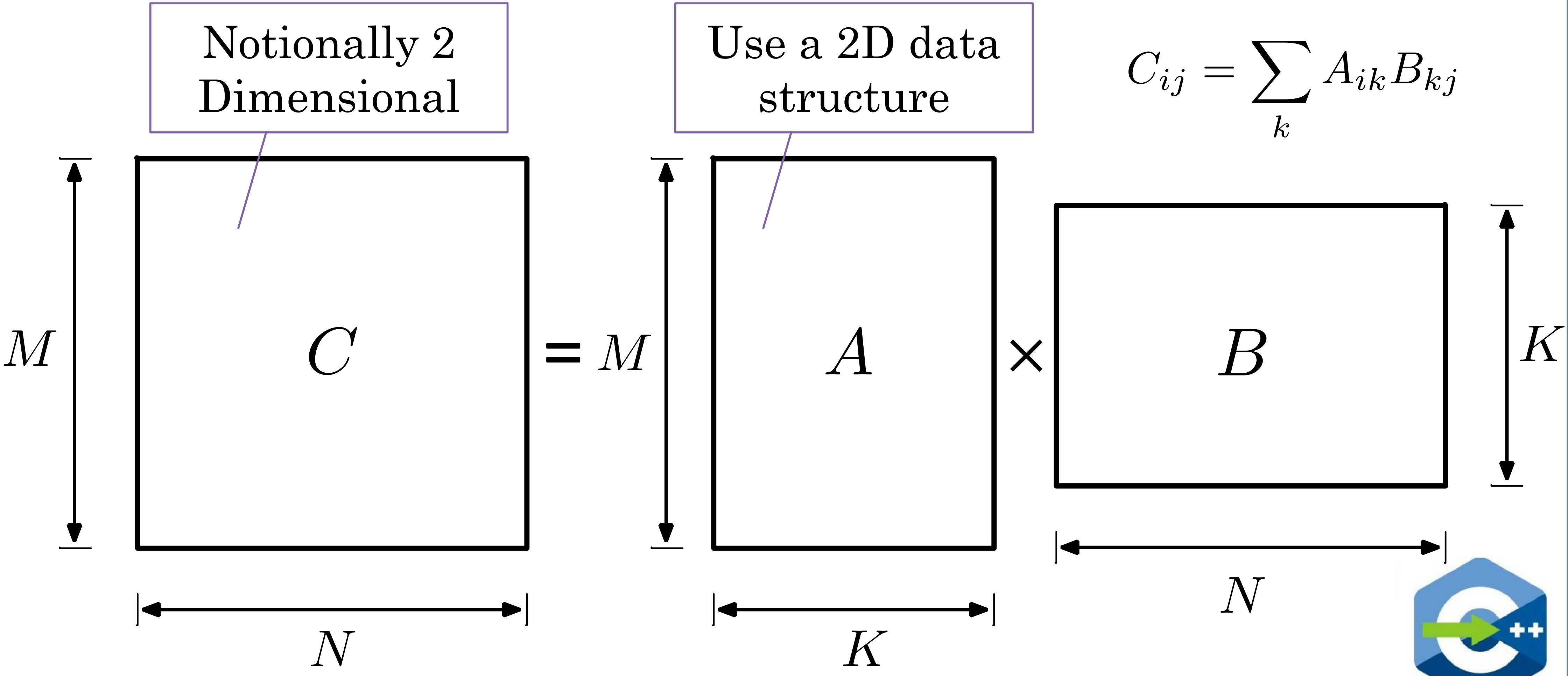
$$C_{ij} = \sum_k A_{ik} B_{kj}$$

```
Matrix A(M, K), B(K, M), C(M, N);

for (size_t i = 0; i != A.num_rows(); ++i)
    for (size_t j = 0; j != B.num_cols(); ++j)
        for (size_t k = 0; k != A.num_cols(); ++k)
            C(i, j) += A(i, k) * B(k, j);
```

Matrix Matrix Product

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

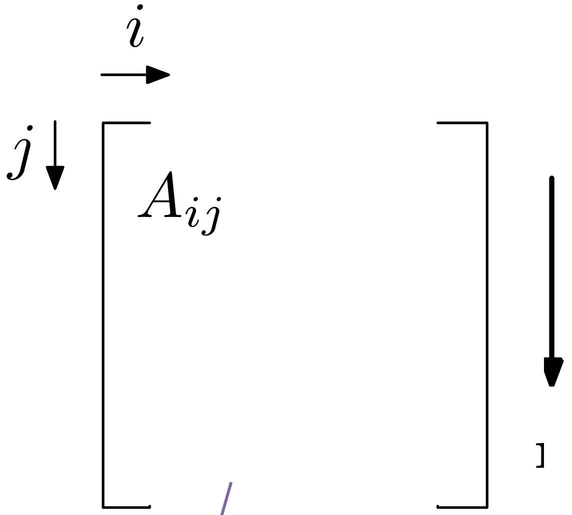


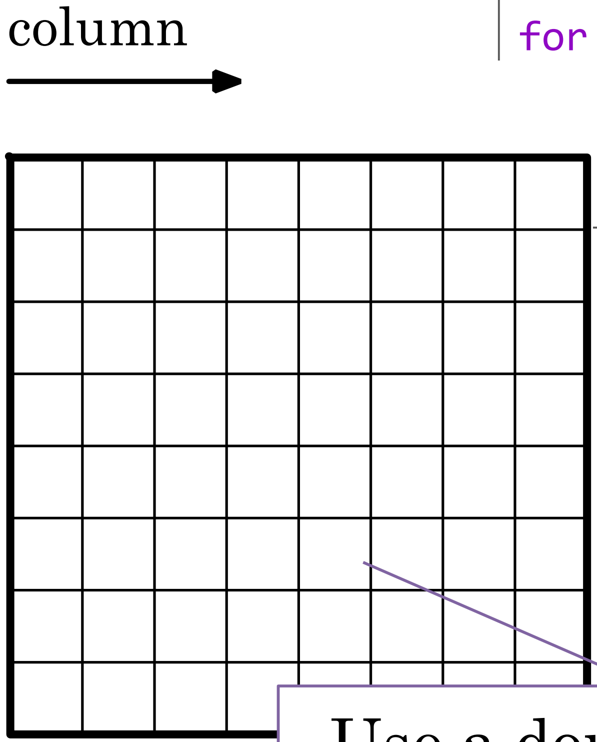
Matrix Representation

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

```
Matrix A(M, K), B(K, M), C(M, N);

for (size_t i = 0; i != A.num_rows(); ++i)
    for (size_t j = 0; j != B.num_cols(); ++j)
        for (size_t k = 0; k != A.num_cols(); ++k)
            C(i, j) += A(i, k) * B(k, j);
```





A matrix is a doubly indexed set

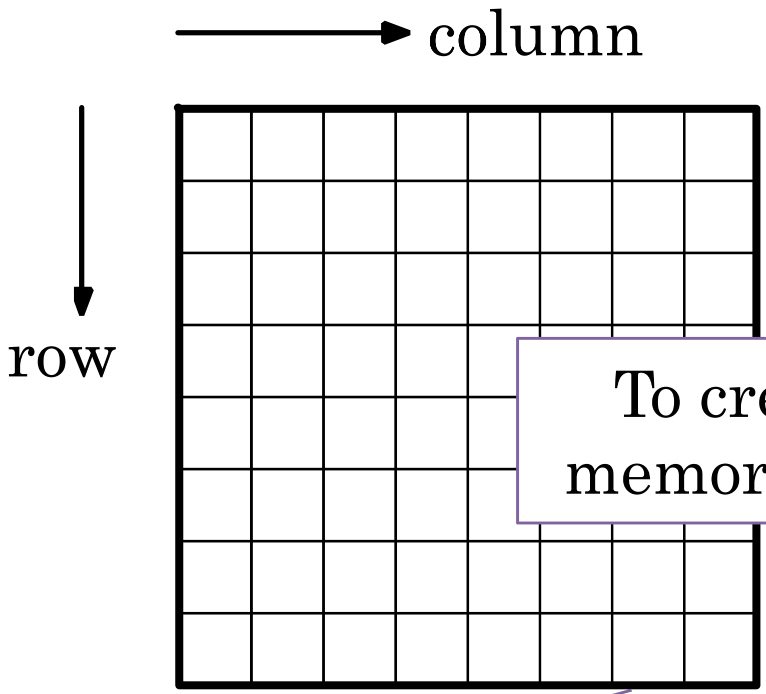
Use a doubly indexed data structure

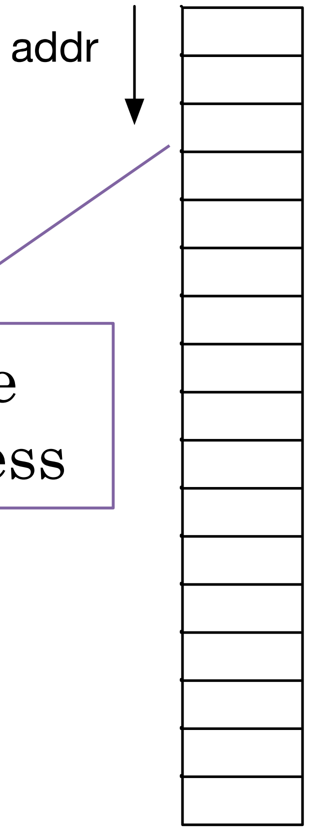
Use a doubly indexed accessor

Matrix Representation

```
Matrix A(M, K), B(K, M), C(M, N);

for (size_t i = 0; i != A.num_rows(); ++i)
    for (size_t j = 0; j != B.num_cols(); ++j)
        for (size_t k = 0; k != A.num_cols(); ++k)
            C(i, j) += A(i, k) * B(k, j);
```





To create one memory address

Memory is a contiguous address space

Use a doubly indexed accessor

Matrix Representation

- Several options to represent a 2D structure in memory
  - Translate double index to single address

- Arrays of arrays:

```cpp
double** storage = ...;                          // storage[i][j];
```

  - Use outer pointer to get inner pointer: `[i]`
  - Lookup element from inner pointer: `[j]`

- Vector of vectors:

```cpp
std::vector<std::vector<double>> storage;   // storage[i][j];
```

  - Lookup inner vector from outer vector: `[i]`
  - Use inner vector to get data element: `[j]`

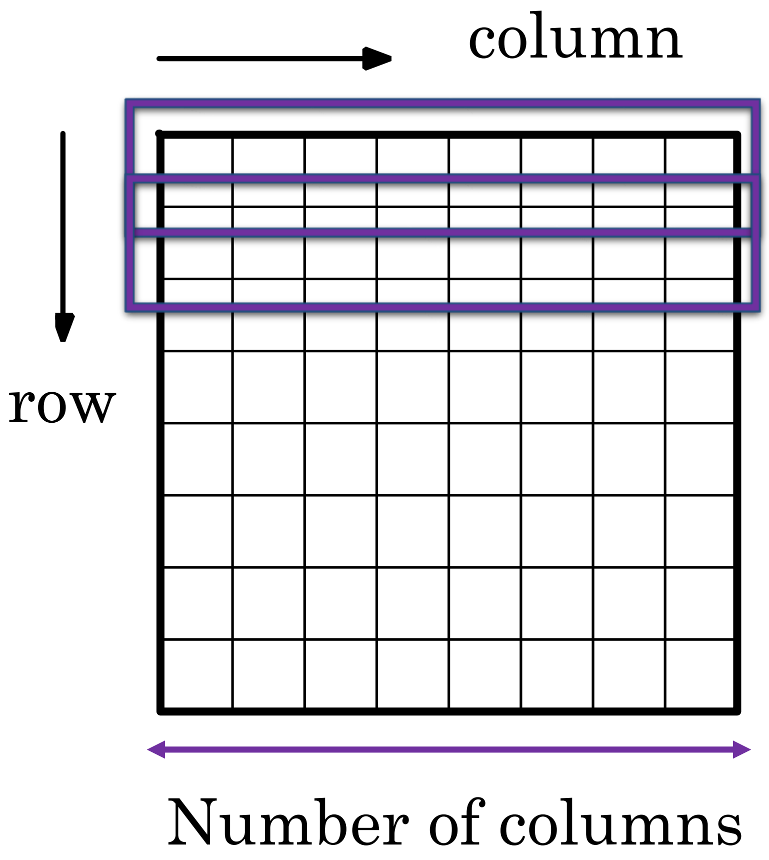
Matrix Representation

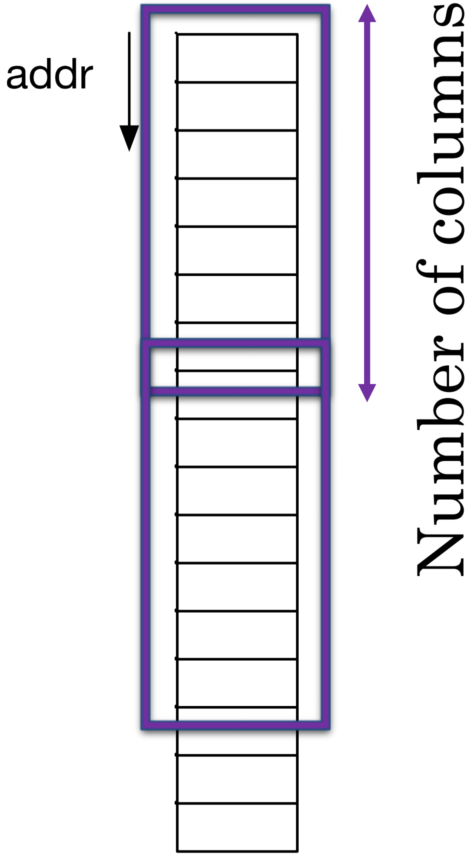
- Several options to represent a 2D structure in memory
  - Translate double index to single address
- Use single vector to get data element:

```
std::vector<double> storage;                     // storage[k];
```

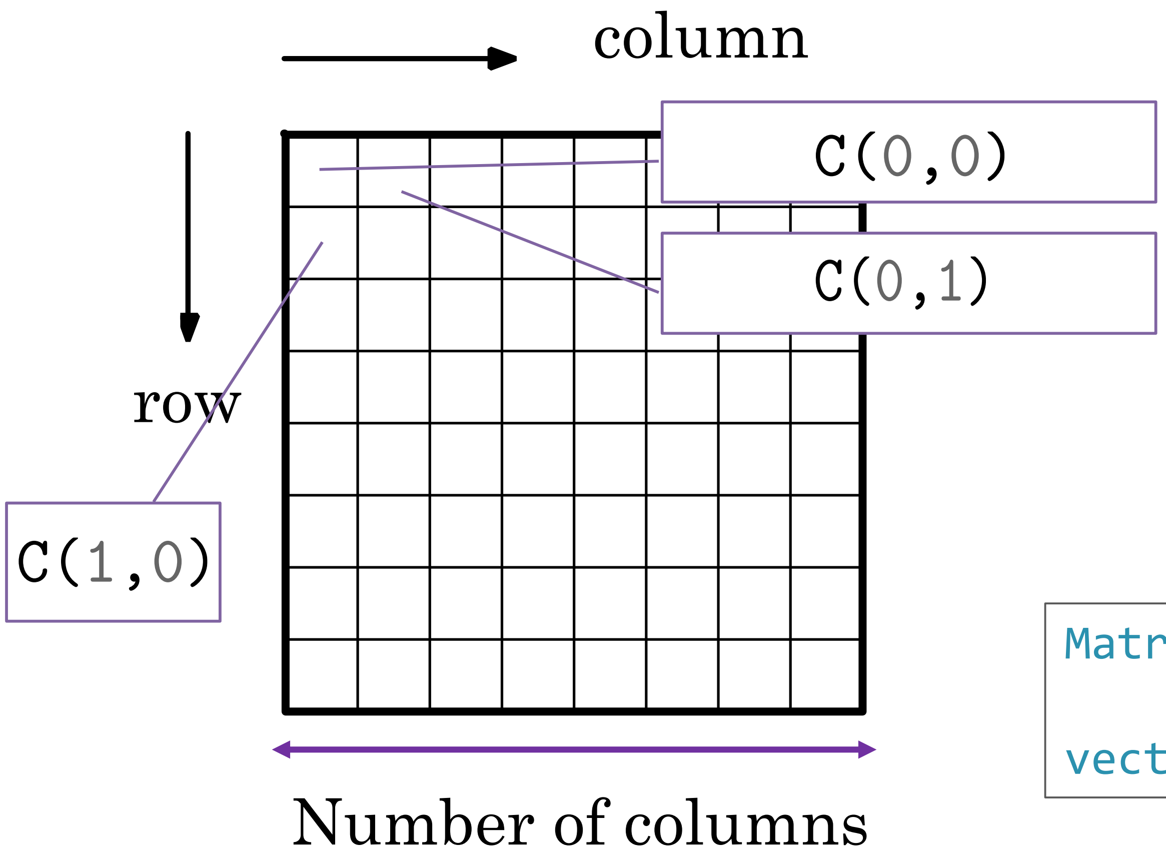
  - Need to compute index: `[k]`

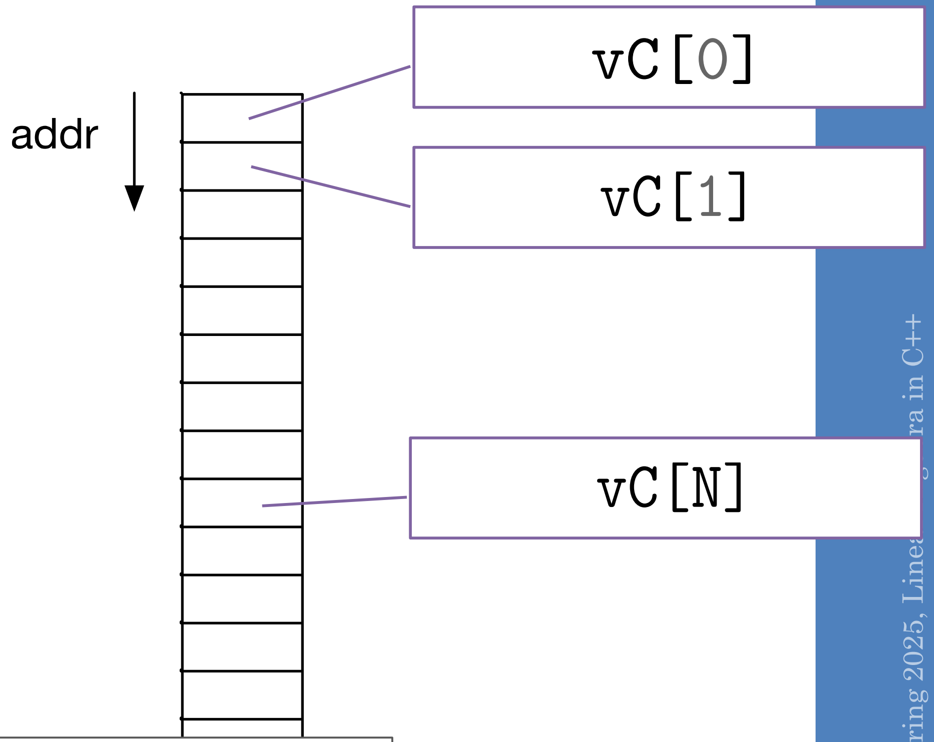
Matrix Representation





Matrix Representation

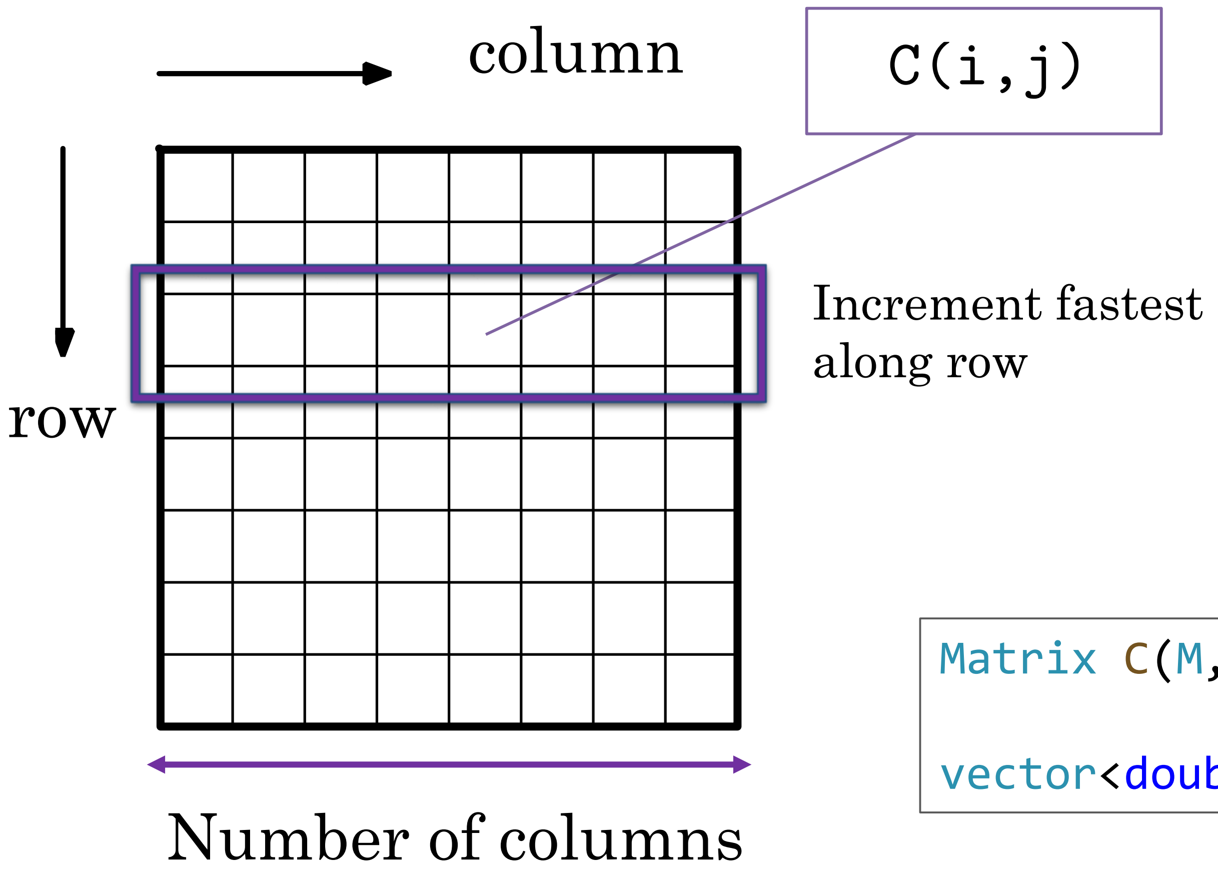


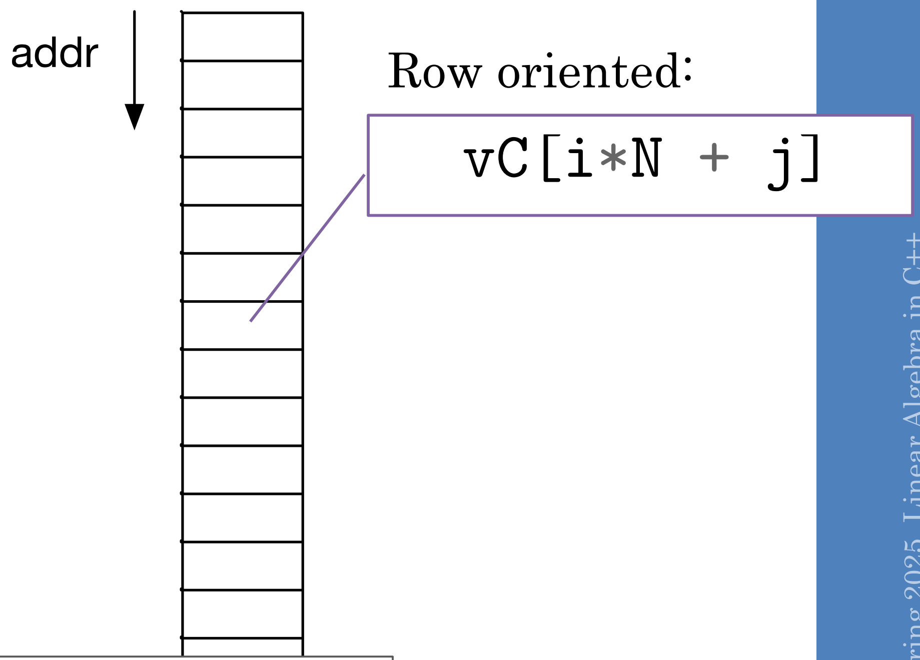



```
Matrix C(M, N);

vector<double> vC(M * N);
```

Matrix Representation

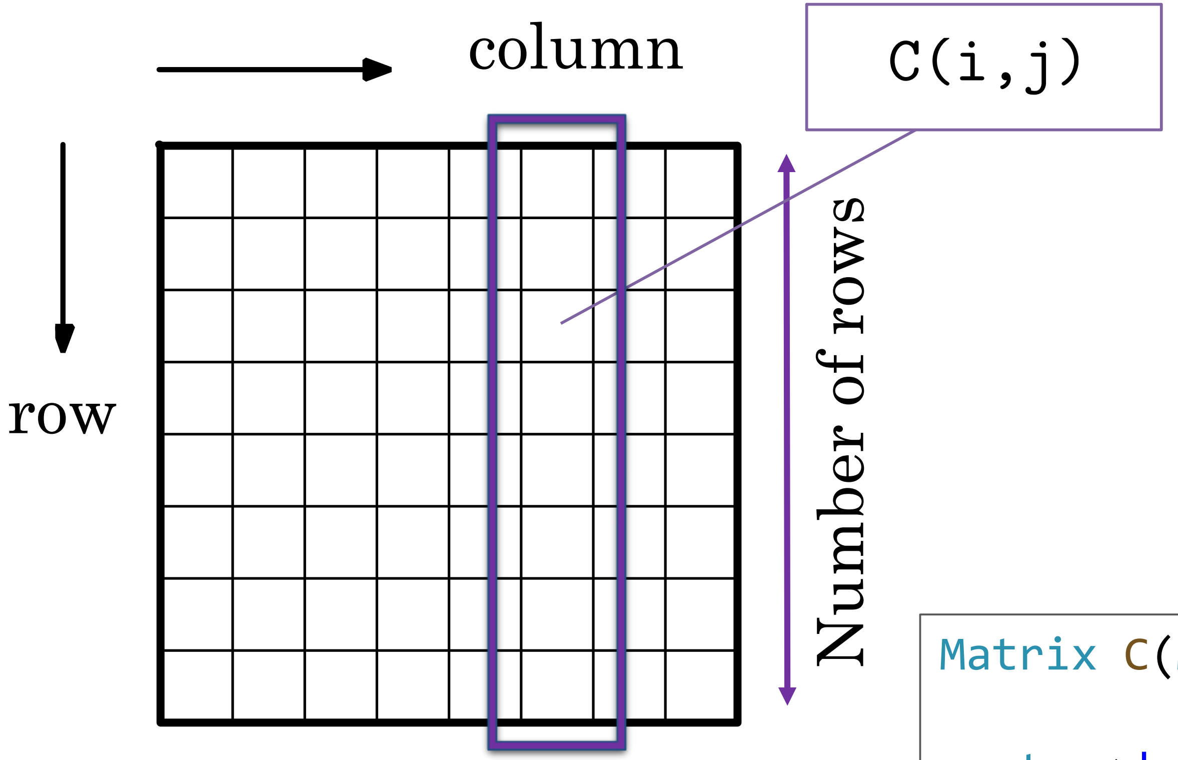




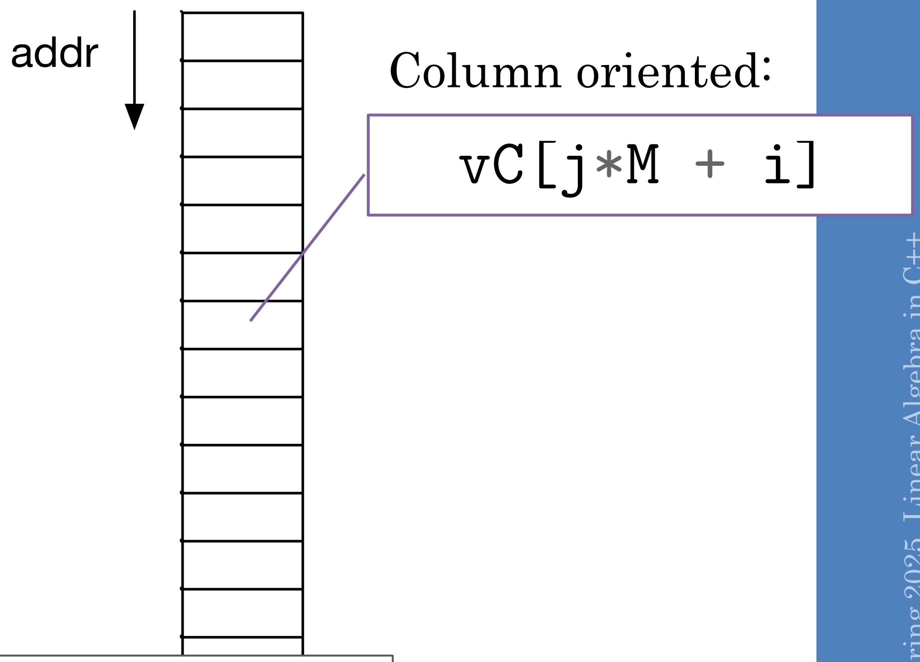

```
Matrix C(M, N);

vector<double> vC(M * N);
```

Matrix Representation



Increment fastest along column




```
Matrix C(M, N);

vector<double> vC(M * N);
```

Matrix Implementation

- Row oriented memory layout (row major):
  - We write: `C(i,j)`
  - But will do this (under the hood): `vC[i*N + j]`

- Column oriented memory layout (column major):
  - We write: `C(i,j)`
  - But will do this (under the hood): `vC[j*M + i]`

Matrix Implementation in C++

- Row oriented memory layout (row major):
  - We write: `C(i,j)`
  - But will do this (under the hood): `vC[i*N + j]`

- Simultaneously (and safely) need a matrix type with
  - (i, j) two dimensional accessor
  - Transparent translation to one dimensional accessor

- Preprocessor? `#define C(i,j) vC[i*N+j]`

Matrix in C++

```cpp
class Matrix {
public:
    Matrix(size_t M, size_t N) : num_rows(M), num_columns(N), storage(N * M) {}
    Matrix(size_t M, size_t N, double init)
      : num_rows(M), num_columns(N), storage(N * M, init) {}

    double& operator()(size_t i, size_t j) {
        return storage[i * num_columns + j]; }
    double const& operator()(size_t i, size_t j) const {
        return storage[i * num_columns + j]; }

    size_t rows() const { return num_rows; }
    size_t columns() const { return num_columns; }

private:
    size_t num_rows, num_columns;
    std::vector<double> storage;
};
```

Matrix in C++: Memory Layouts

- Different memory layouts:
  - Row-major

```
double& operator()(size_t i, size_t j) {
    return storage[i * num_columns + j]; }
double const& operator()(size_t i, size_t j) const {
    return storage[i * num_columns + j]; }
```

  - Column-major:

```
double& operator()(size_t i, size_t j) {
    return storage[j * num_rows + i]; }
double const& operator()(size_t i, size_t j) const {
    return storage[j * num_rows + i]; }
```

Matrix Multiplication

Matrix-Matrix Multiplication, Sequential Code

- Assume throughout that the matrices are square (n x n matrices).
- The sequential code to compute *A x B* could simply be:

```
for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j) {
        c[i][j] = 0;
        for (k = 0; k < n; ++k)
            c[i][j] += a[i][k] * b[k][j];
    }
```

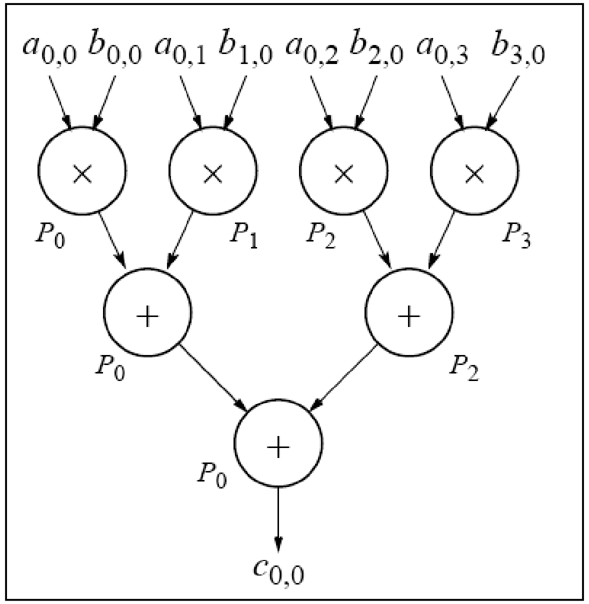
- This algorithm requires n^3 multiplications and n^3 additions, leading to a sequential time complexity of $O(n^3)$
- Very easy to parallelize
- Very difficult to parallelize efficiently

Matrix-Matrix Multiplication

- For $n \; x \; n$ matrices, we can obtain:
  - Time complexity of $O(n^2)$ with n processors
    - Each instance of inner loop is independent and can be done by a separate processor
  - Time complexity of $O(n)$ with n^2 processors
    - One element of C assigned to each processor
    - Cost optimal since $O(n^3) = n \; x \; O(n^2) = n^2 \; x \; O(n)$
  - Time complexity of $O(\log n)$ with n^3 processors
    - By parallelizing the inner loop.
    - Not cost-optimal since $O(n^3) < n^3 \; x \; O(\log n)$

Matrix-Matrix Multiplication

- Using tree construction n numbers can be added in $O(\log n)$ steps (using n^3 processors)



- $O(\log n)$ lower bound for parallel matrix multiplication

Timing and Benchmarking

Timing and Benchmarking

- Humans have pathological need to see who is better at everything
- But ordering requires a single number corresponding to "goodness"
  - Which is impossible of course
- So we take one task and turn that into the definition of goodness (e.g., IQ)
  - (What is IQ? It's the thing that the IQ test measures.)
- In HPC, we take performance on a particular computational task to rank the worlds computers with the 500 best scores on this task
  - Linear system solution – matrix-matrix product at the core
  - Performance = FLOP/s = (Total computations) / (Time to compute)
  - Linpack → $2N^3$ / (Time to compute)

Timing a Program

- The time program in Linux (Unix) will measure time resources a process uses

```
$ time ls -lR /usr > /dev/null

real   0m0.464s
user   0m0.080s
sys   0m0.380s
```

- "real": Elapsed Wall Clock – this is what we're interested in

C++ Timer

```cpp
class timer {
private:
    using time_t = std::chrono::time_point<std::chrono::system_clock>;
    time_t start_time, stop_time;

public:
    timer() = default;

    time_t start()    { return (start_time = std::chrono::system_clock::now()); }
    time_t stop()     { return (stop_time  = std::chrono::system_clock::now()); }
    double elapsed() {
        auto diff = stop_time - start_time;
        return std::chrono::duration_cast<std::chrono::milliseconds>(diff).count();
    }
};
```

And this will be provided to you

All you need to worry about

Measuring Matrix-Matrix Product

$$C_{ij} = \sum_{k=0}^{K-1} A_{ik} B_{kj}$$

- Three nested loops
  - Multiplication
  - Addition
  - How much data?
  - How many FLOP/s?

```
Matrix A(M, K), B(K, M), C(M, N);

for (size_t i = 0; i != A.num_rows(); ++i)
    for (size_t j = 0; j != B.num_cols(); ++j)
        for (size_t k = 0; k != A.num_cols(); ++k)
            C(i, j) += A(i, k) * B(k, j);
```

Measuring Matrix Matrix Product

```
int main()
{
    for (int size = 8; size != 1024; size *= 2)
    {
        Matrix A(size, size), B(size, size);

        timer t;    // define timer t
        t.start()      // start timer t
        Matrix C = A * B;
        t.stop();      // stop timer t

        std::println("size: {}, elapsed: {}",
            size, t.elapsed());
    }
    return 0;
}
```

$ ./a.out

N	Elapsed
8	0
16	0
32	0
64	0
128	2
256	28
512	315

Insufficient resolution!

What Are We Timing?

```
Matrix operator*(Matrix const& A, Matrix const& B)
{
    Matrix C(A.num_rows(), A.num_cols(), 0);    // allocate and zeroize

    // the actual matrix product
    for (size_t i = 0; i != A.num_rows(); ++i) {
        for (size_t j = 0; j != B.num_cols(); ++j) {
            for (size_t k = 0; k != A.num_cols(); ++k) {
                C(i, j) += A(i, k) * B(k, j);
            }
        }
    }
    return C;
}
```

- Never allocate new memory in performance critical sections of code!

Just For Benchmarking

```
void multiply(Matrix const& A, Matrix const& B, Matrix& C)
{
    for (size_t i = 0; i != A.num_rows(); ++i) {
        for (size_t j = 0; j != B.num_cols(); ++j) {
            for (size_t k = 0; k != A.num_cols(); ++k) {
                C(i, j) += A(i, k) * B(k, j);
            }
        }
    }
}

Matrix operator*(Matrix const& A, Matrix const& B)
{
    Matrix C(A.num_rows(), A.num_cols(), 0);
    multiply(A, B, C);
    return C;
}
```

Benchmarking

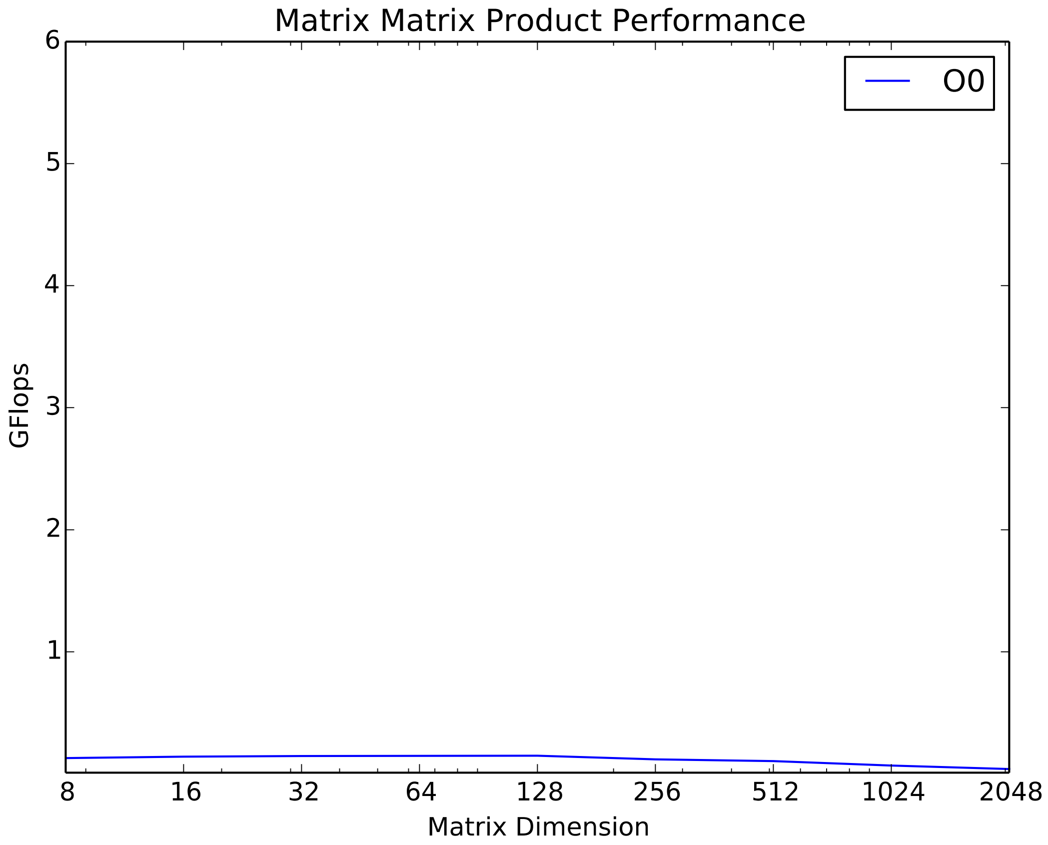
```
int main() {
    for (int size = 8; size != 1024; size *= 2) {
        matrix A(size, size), B(size, size), C(size, size);

        timer t;     // define timer t
        t.start()        // start timer t
        for (int i = 0; i != num_samples; ++i)
            multiply(A, B, C);
        t.stop();        // stop timer t

        std::println("size: {}, elapsed: {}", size, t.elapsed());
    }
    return 0;
}
```

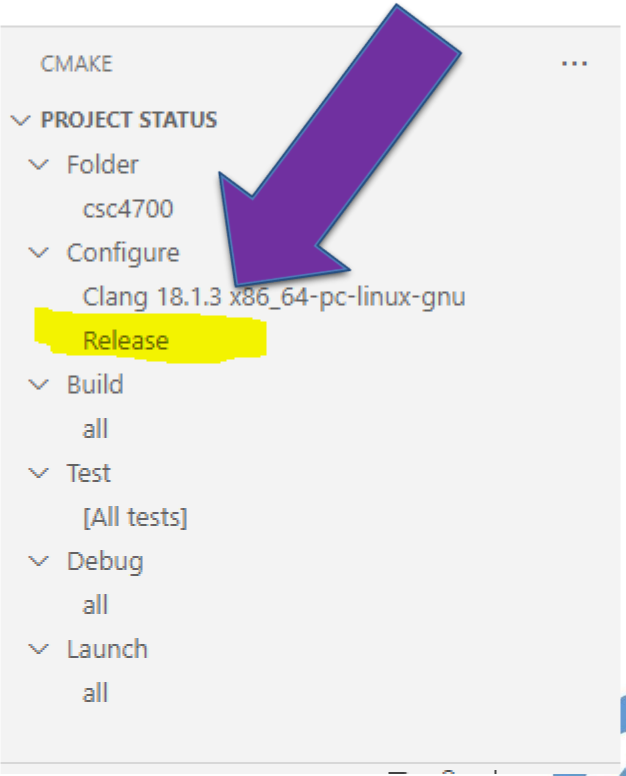
- Run the core loop many times to get sufficient resolution for small(er) sizes

Base Performance Results

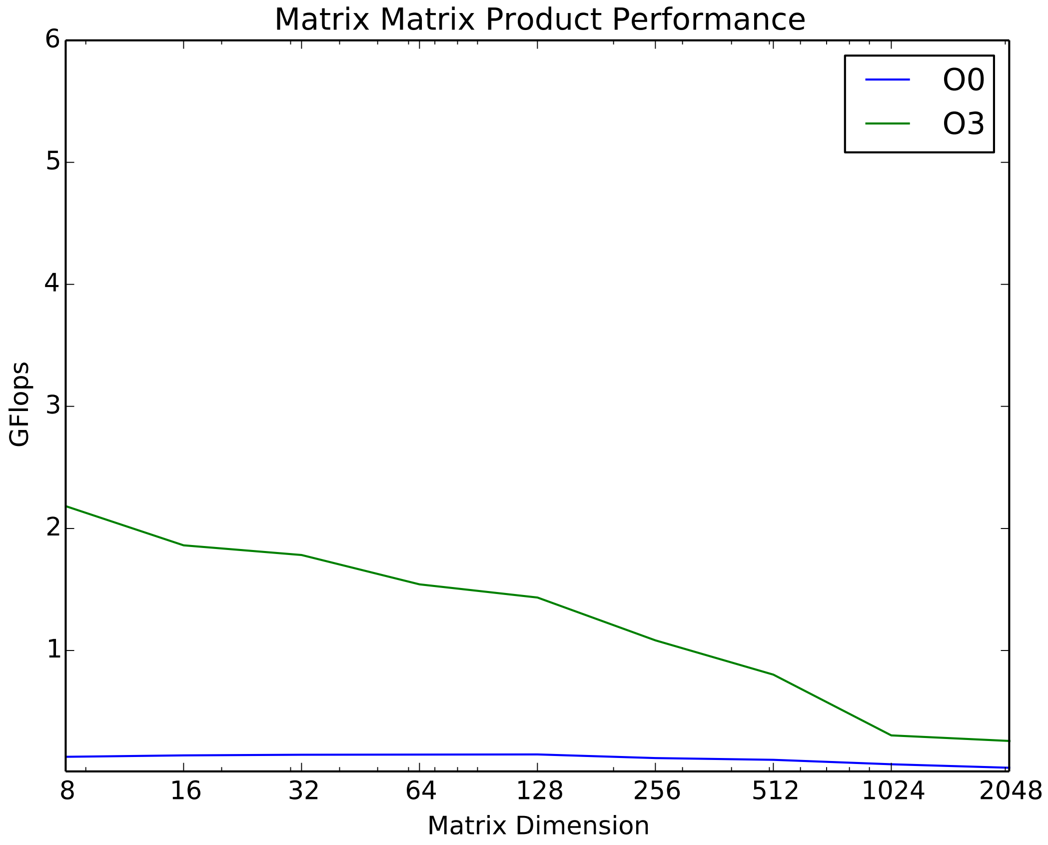


Let’s Make One Small Change

- Change build system to **Release**
- Always do that for performance measurements!



Base Performance Results



Locality → Performance

- Caches are much smaller than main memory. How do we decide what data to keep in cache to effect higher performance (more accesses)?
- Temporal Locality: if a program accesses a memory location, there is a much higher than random probability that the same location will be accessed again
  - Cache replacement policies attempt to keep cached elements in the cache for as long as possible
- Spatial Locality: if a program accesses a memory location, there is a much higher than random probability that nearby locations will also be accessed (soon)
  - Cache policies read contiguous chunks of data – a referenced element and its neighbors – not just single elements

Improving (temporal) Locality

- Consider each step of inner loop:
  - Load `C(i, j)` into register
  - Load `A(i, k)` into register
  - Load `B(k, j)` into register
  - Multiply
  - Add
  - Store `C(i, j)`

```
for (size_t i = 0; i != M; ++i)
    for (size_t j = 0; j != N; ++j)
        for (size_t k = 0; k != N; ++k)
            C(i, j) += A(i, k) * B(k, j);
```

- Four memory operations and two floating point operations per iteration
  - 2/6 == 1/3 flop per cycle (if each operation is one cycle)

Hoisting

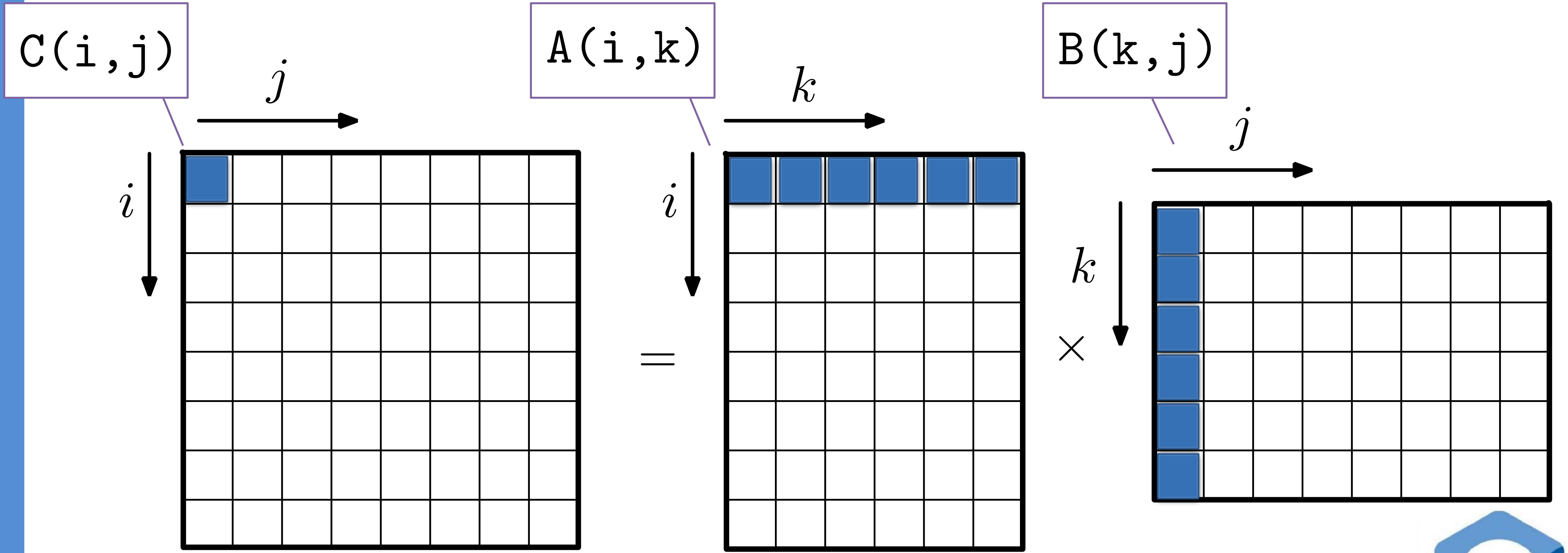
- What can be reused?
  - `C(i, j)`!
  - Hoist it!

    - Load `A(i, k)`
    - Load `B(k, j)`
    - Multiply
    - Add

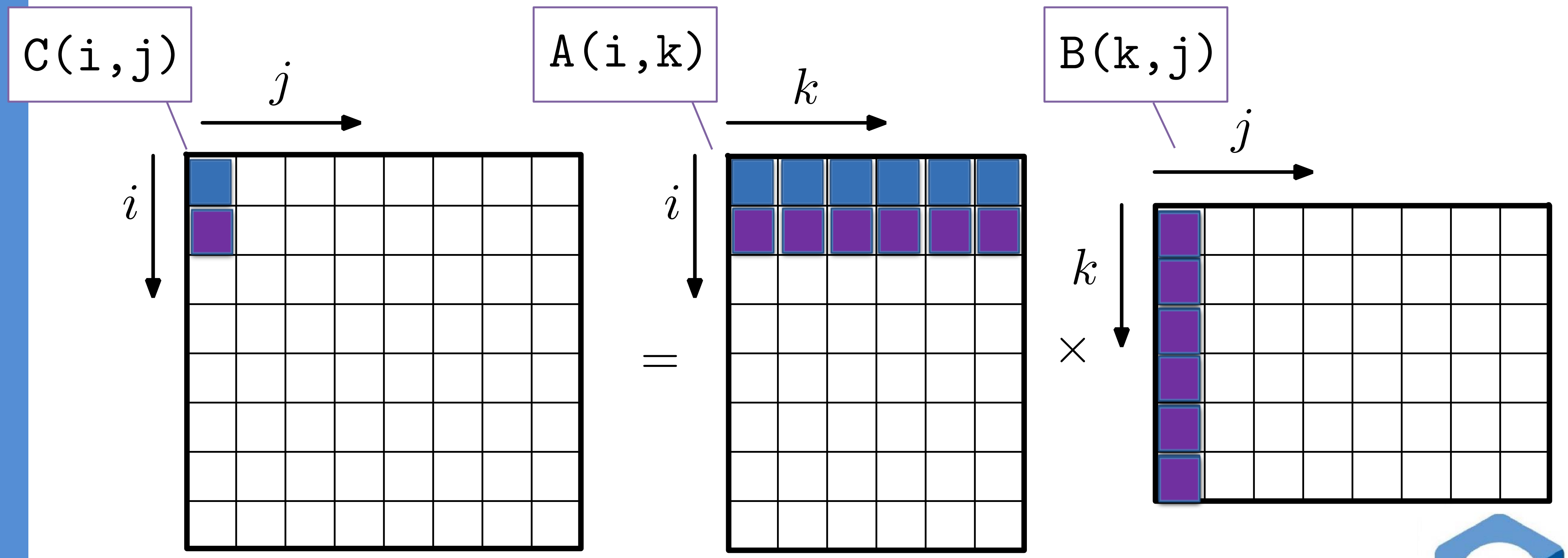
```
for (size_t i = 0; i != M; ++i)
    for (size_t j = 0; j != N; ++j) {
        double t = C(i, j);
        for (size_t k = 0; k != N; ++k)
            t += A(i, k) * B(k, j);
        C(i, j) = t;
    }
```

- Two memory operations and two floating point operations per iteration
  - 2/4 == 1/2 flop per cycle (if each operation is one cycle)

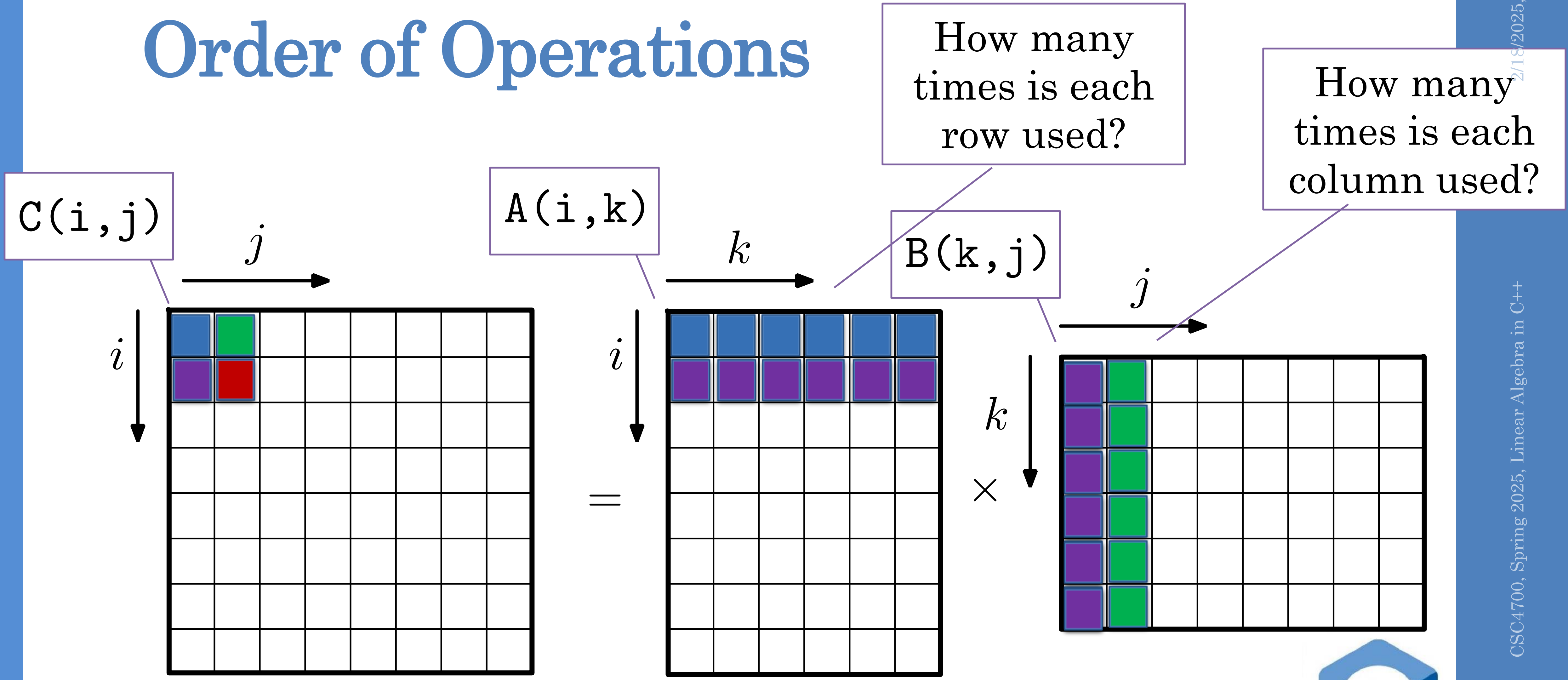
Order of Operations



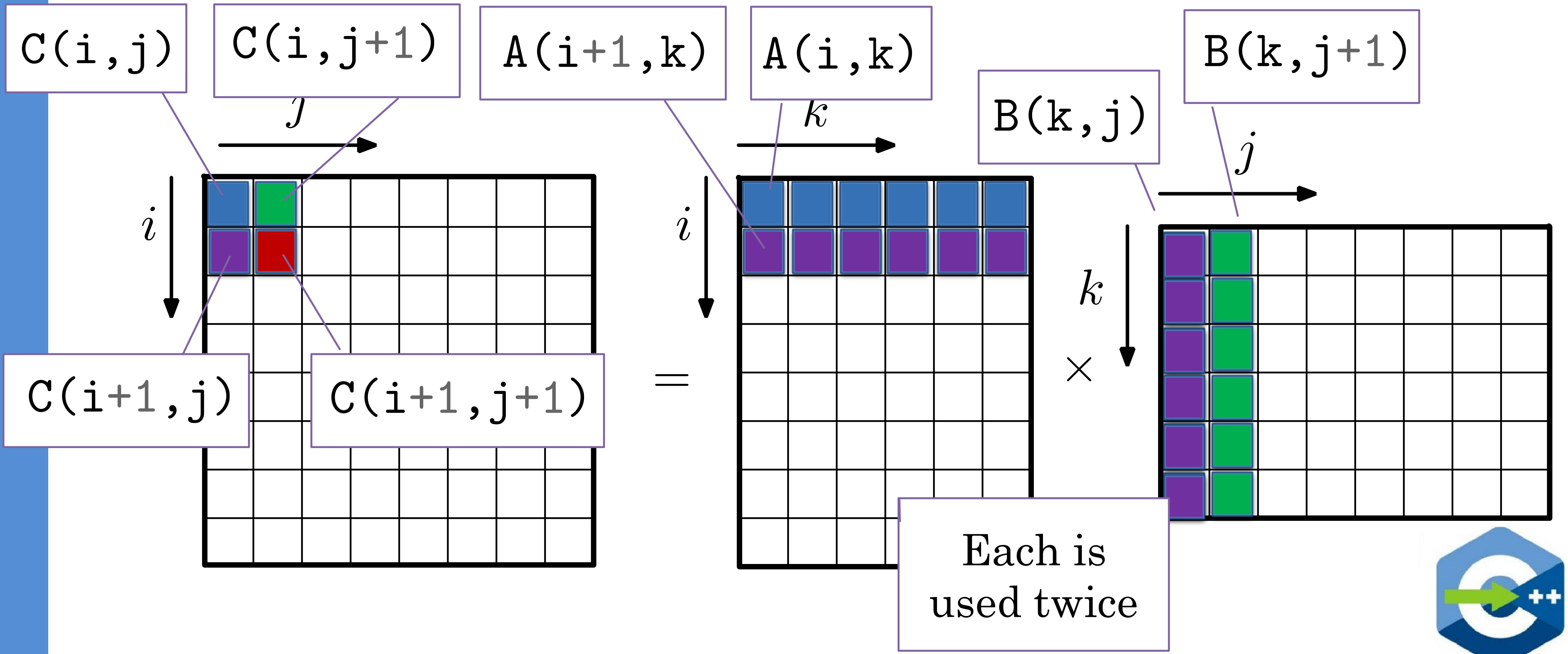
Order of Operations



Order of Operations



Reuse: How Many Times Are Data Reused?

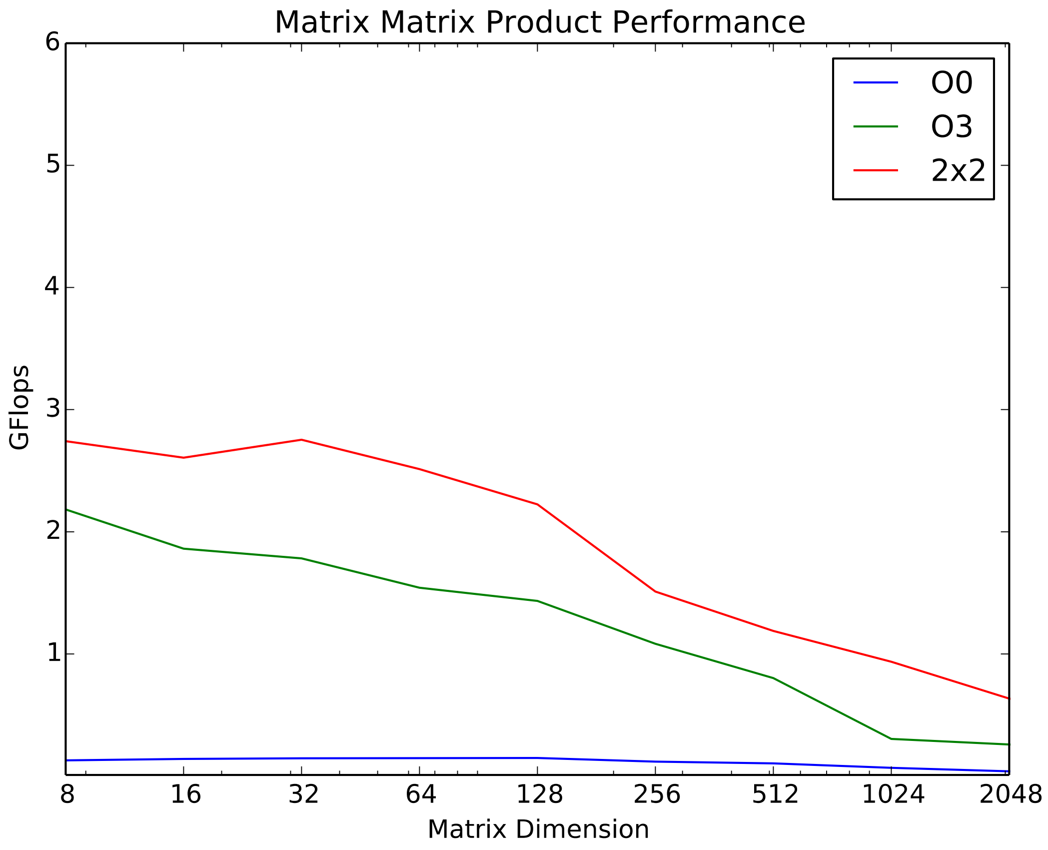


Improving (temporal) Locality: Unroll

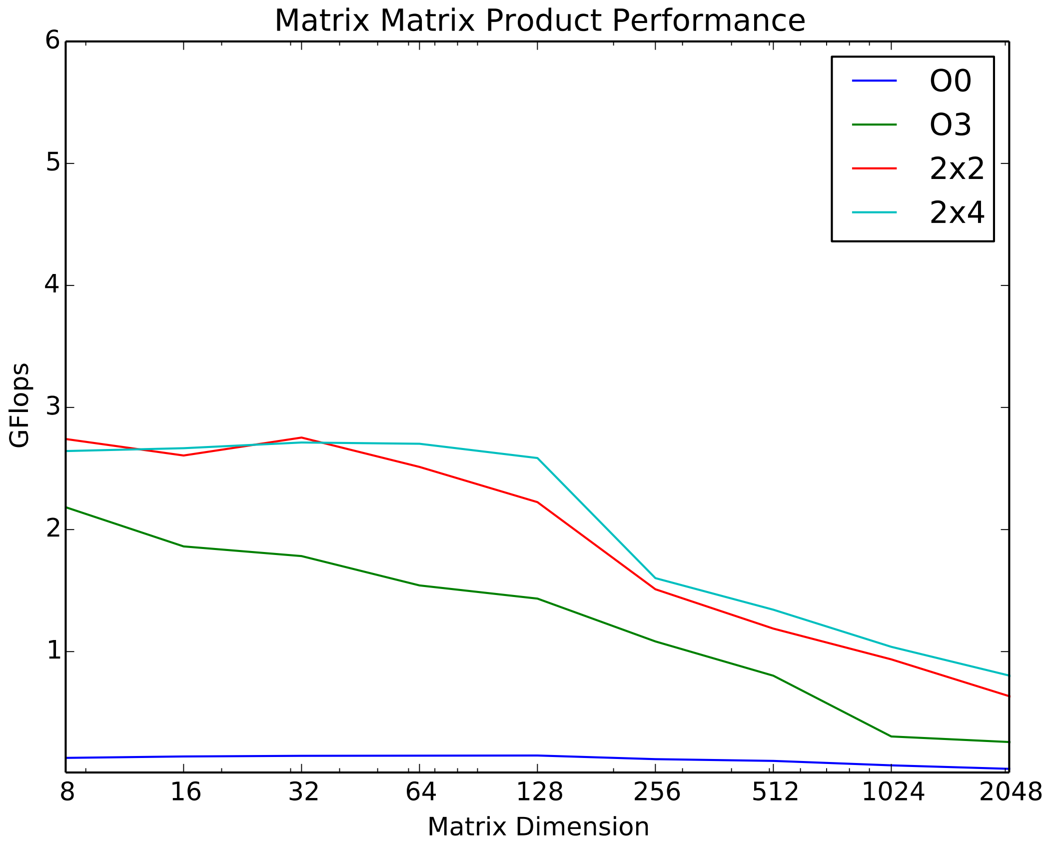
```
void tiled_multiply_2x2(matrix const& A, matrix const& B, matrix& C) {
    for (size_t i = 0; i != A.num_rows(); i += 2)
        for (size_t j = 0; j != B.num_cols(); j += 2)
            for (size_t k = 0; k != A.num_cols(); ++k) {
                C(i, j)         += A(i, k)     * B(k, j);
                C(i, j + 1)     += A(i, k)     * B(k, j + 1);
                C(i + 1, j)     += A(i + 1, k) * B(k, j);
                C(i + 1, j + 1) += A(i + 1, k) * B(k, j + 1);
            }
}
```

- `A(i, k), A(i+1, k), B(k, j), B(k, j+1)`: used twice
- `C(i, j), C(i, j + 1), C(i + 1, j), C(i + 1, j + 1)`: can be hoisted
- Four memory operations and eight floating point operations per iteration
  - 8/12 = 2/3 flop per cycle (if each operation is one cycle) – 2X the base case

Example: Register Locality (2 by 2)

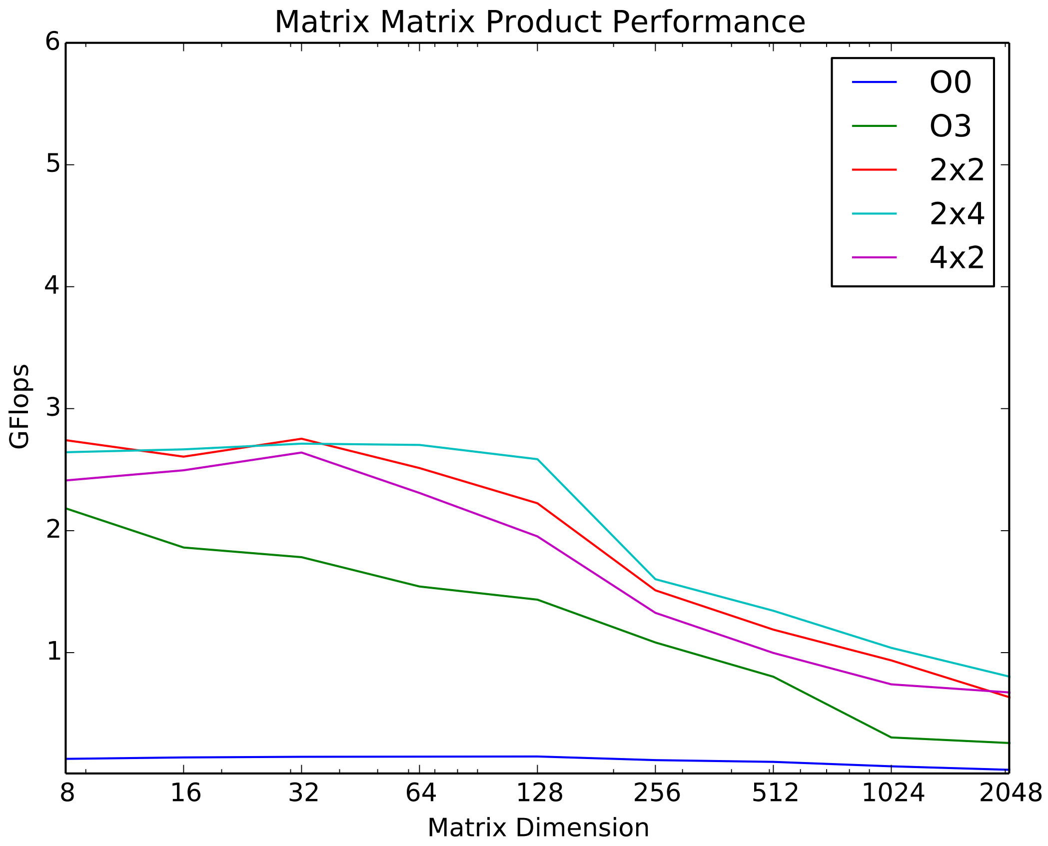


2 by 4



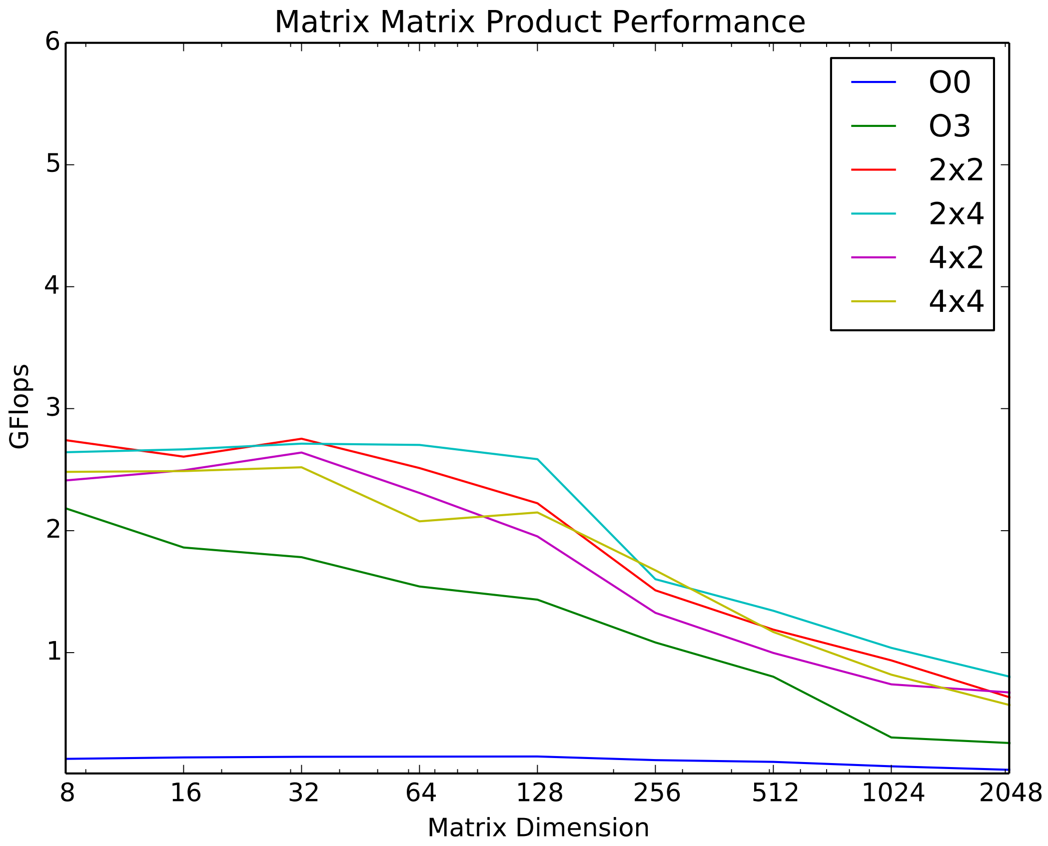
Matrix Matrix Product Performance
O0
O3
2x2
2x4
GFlops
Matrix Dimension
8
16
32
64
128
256
512
1024
2048
1
2
3
4
5
6

4 by 2



Matrix Matrix Product Performance
GFlops
Matrix Dimension
6
5
4
3
2
1
8
16
32
64
128
256
512
1024
2048
O0
O3
2x2
2x4
4x2

4 by 4

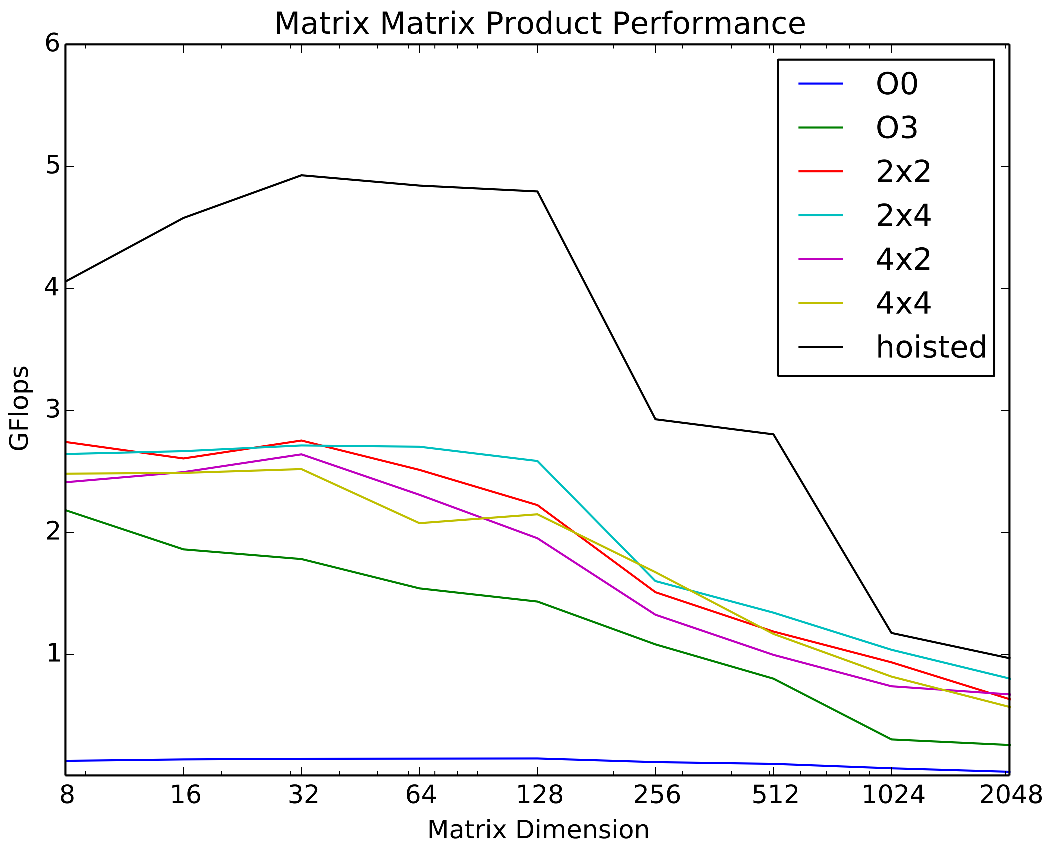


Matrix Matrix Product Performance
GFlops
6
5
4
3
2
1
8
16
32
64
128
256
512
1024
2048
Matrix Dimension
O0
O3
2x2
2x4
4x2
4x4

Tiling and Hoisting

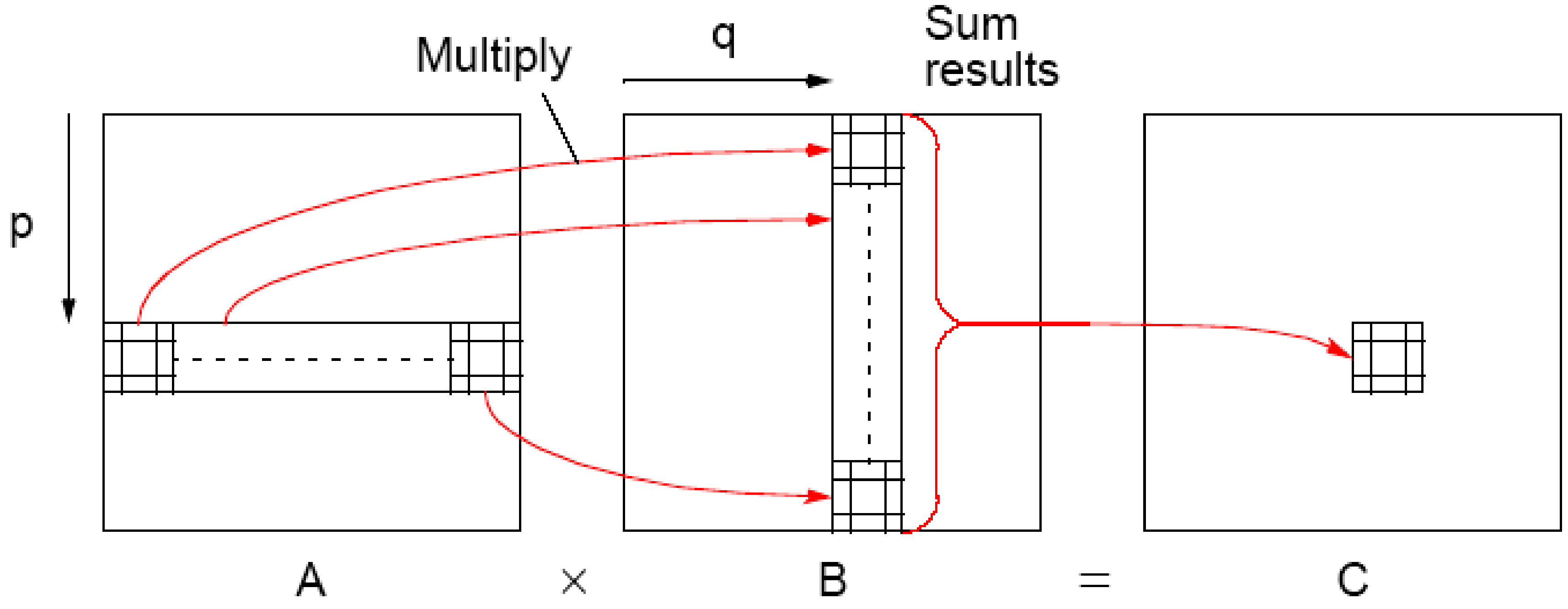
```
void hoisted_tiled_multiply_2x2(matrix const& A, matrix const& B, matrix& C)
{
    for (size_t i = 0; i != M; ++i) {
        for (size_t j = 0; j != N; ++j) {
            double t00 = C(i, j),     t01 = C(i, j + 1);
            double t01 = C(i + 1, j), t11 = C(i + 1, j + 1);
            for (size_t k = 0; k != N; ++k) {
                t00 += A(i, k)     * B(k, j);
                t01 += A(i, k)     * B(k, j + 1);
                t10 += A(i + 1, k) * B(k, j);
                t11 += A(i + 1, k) * B(k, j + 1);
            }
            C(i, j)     = t00; C(i, j + 1)     = t01;
            C(i + 1, j) = t10; C(i + 1, j + 1) = t11;
        }
    }
}
```

Tiling 2 by 2 and Hoisting



Block Matrix Multiplication

- Partitioning into sub-matricies



Block Matrix Multiplication

$$\left[\begin{array}{cc|cc} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ \hline a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{array}\right] \times \left[\begin{array}{cc|cc} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{array}\right]$$

$$\overset{A_{0,0}}{\begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix}} \times \overset{B_{0,0}}{\begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix}} + \overset{A_{0,1}}{\begin{bmatrix} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \end{bmatrix}} \times \overset{B_{1,0}}{\begin{bmatrix} b_{2,0} & b_{2,1} \\ b_{3,0} & b_{3,1} \end{bmatrix}}$$

$$= \begin{bmatrix} a_{0,0}b_{0,0}+a_{0,1}b_{1,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1} \end{bmatrix} + \begin{bmatrix} a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{bmatrix}$$

$$= \begin{bmatrix} a_{0,0}b_{0,0}+a_{0,1}b_{1,0}+a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1}+a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0}+a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1}+a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{bmatrix}$$

$$= C_{0,0}$$

Improving Locality: Cache

- Large matrix problems won't fit completely into cache
- Use blocked algorithm – work with blocks that will fit into cache

$$\begin{bmatrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} & B_{02} & B_{03} \\ B_{10} & B_{11} & B_{12} & B_{13} \\ B_{20} & B_{21} & B_{22} & B_{23} \\ B_{30} & B_{31} & B_{32} & B_{33} \end{bmatrix}$$

$$C_{IJ} = \sum_K A_{IK} B_{KJ}$$

$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31}$$

- Each product term fits completely into cache and runs at high-performance
- Cache misses amortized $O(N^3)$ work with $O(N^2)$ data

Blocking and Tiling

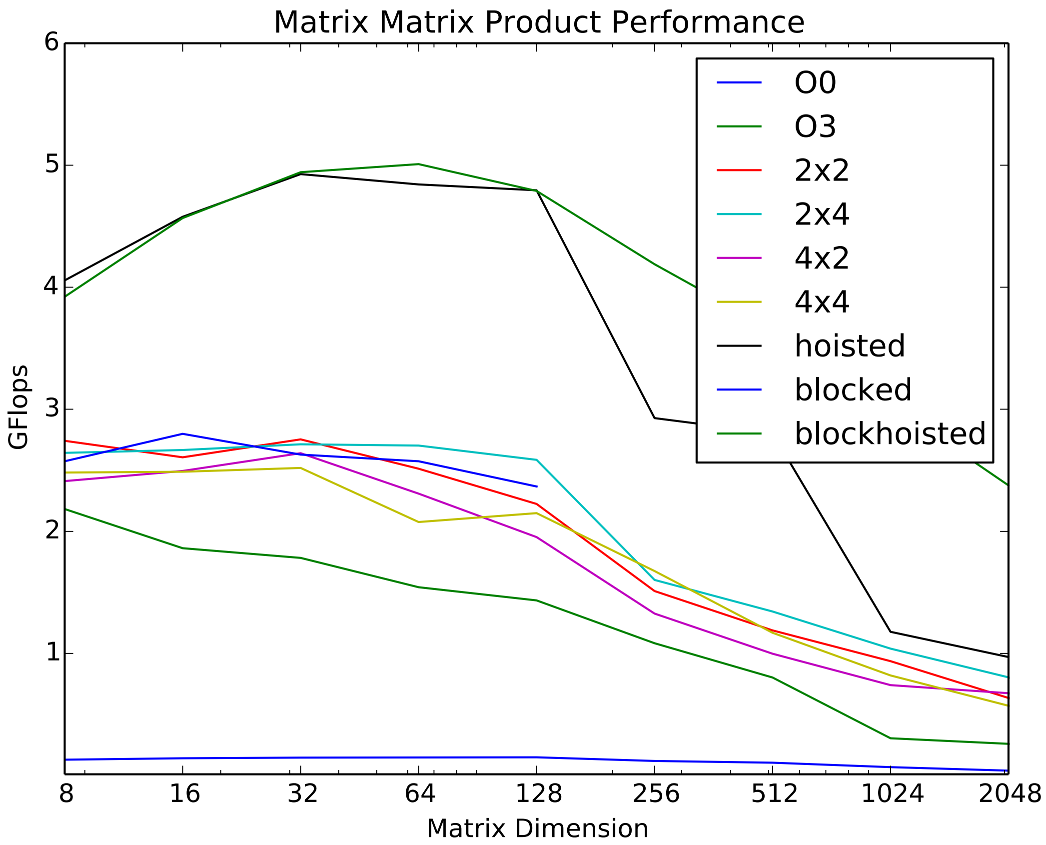
```
void block_tiled_multiply_2x2(matrix const& A, matrix const& B, matrix& C)
{
    for (size_t ii = 0; ii != A.num_rows(); ii += blocksize) {
        for (size_t jj = 0; jj != B.num_cols(); jj += blocksize) {
            for (size_t kk = 0; kk != A.num_cols(); kk += blocksize) {

                for (size_t i = ii; i != ii + blocksize; i += 2) {
                    for (size_t j = jj; j != jj + blocksize; j += 2) {
                        for (size_t k = kk; k != kk + blocksize; ++k) {
                            C(i    , j    ) += A(i    , k) * B(k, j);
                            C(i    , j + 1) += A(i    , k) * B(k, j + 1);
                            C(i + 1, j    ) += A(i + 1, k) * B(k, j);
                            C(i + 1, j + 1) += A(i + 1, k) * B(k, j + 1);
                        }
                    }
                }
            }
        }
    }
}
```

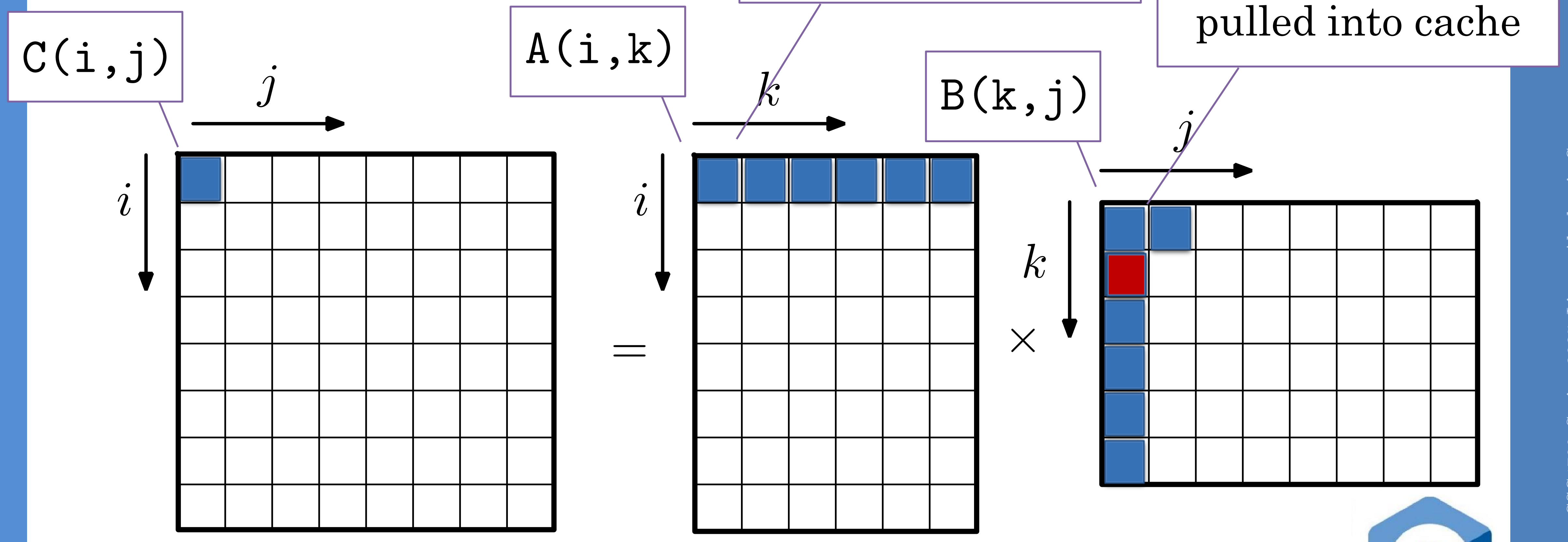
Outer loops work across blocks (for each block)

Inner loops work on blocks

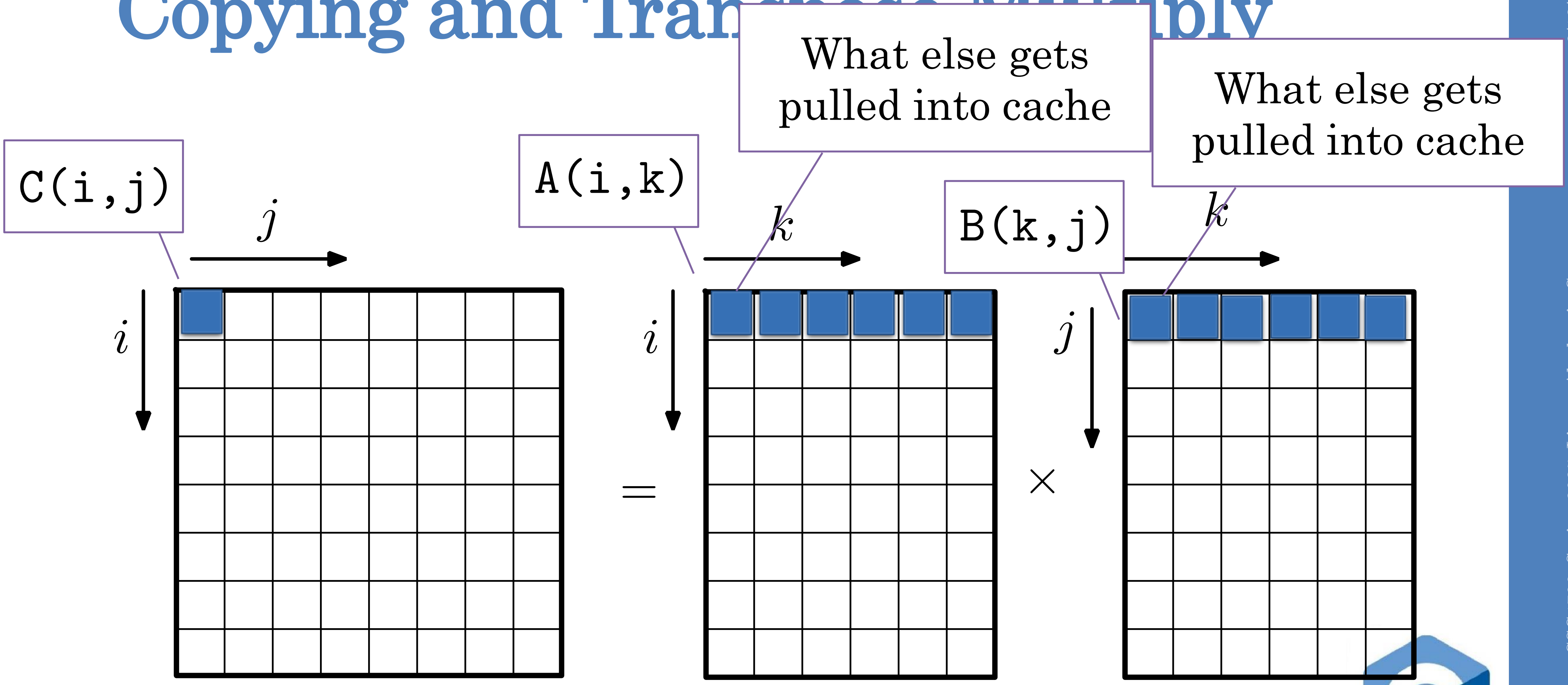
Blocking and Tiling and Hoisting



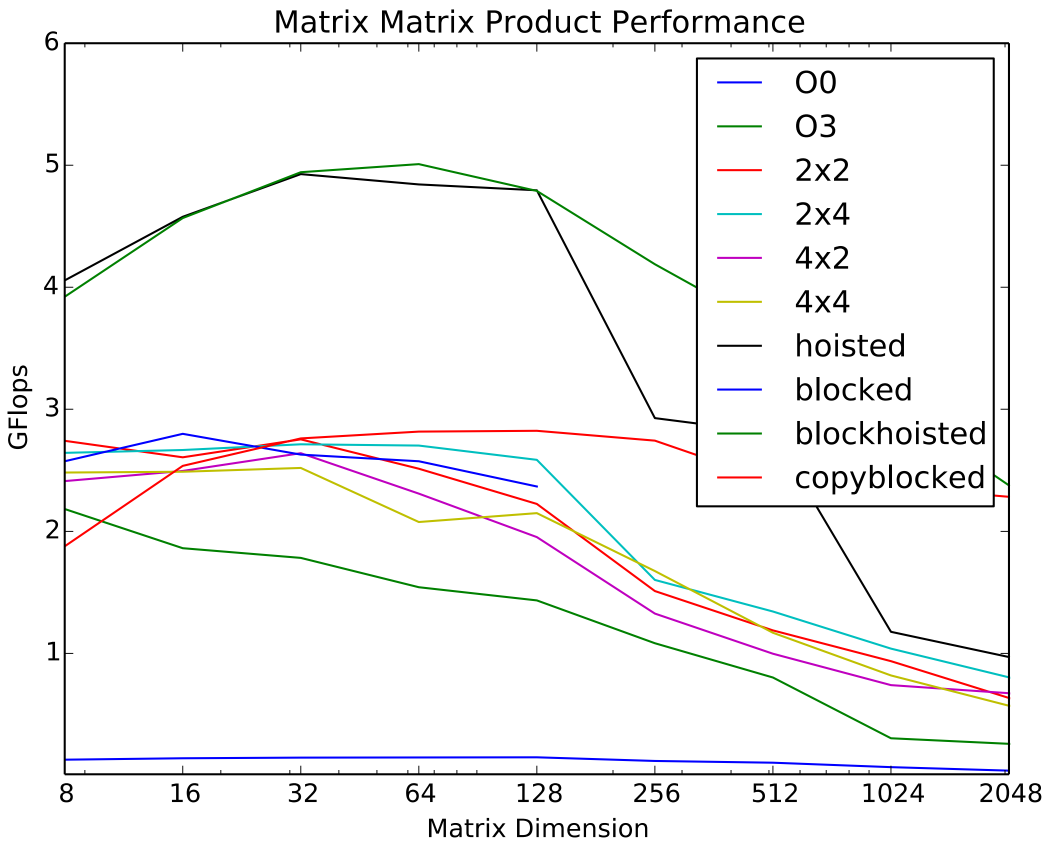
Copying and Tran... Multiply



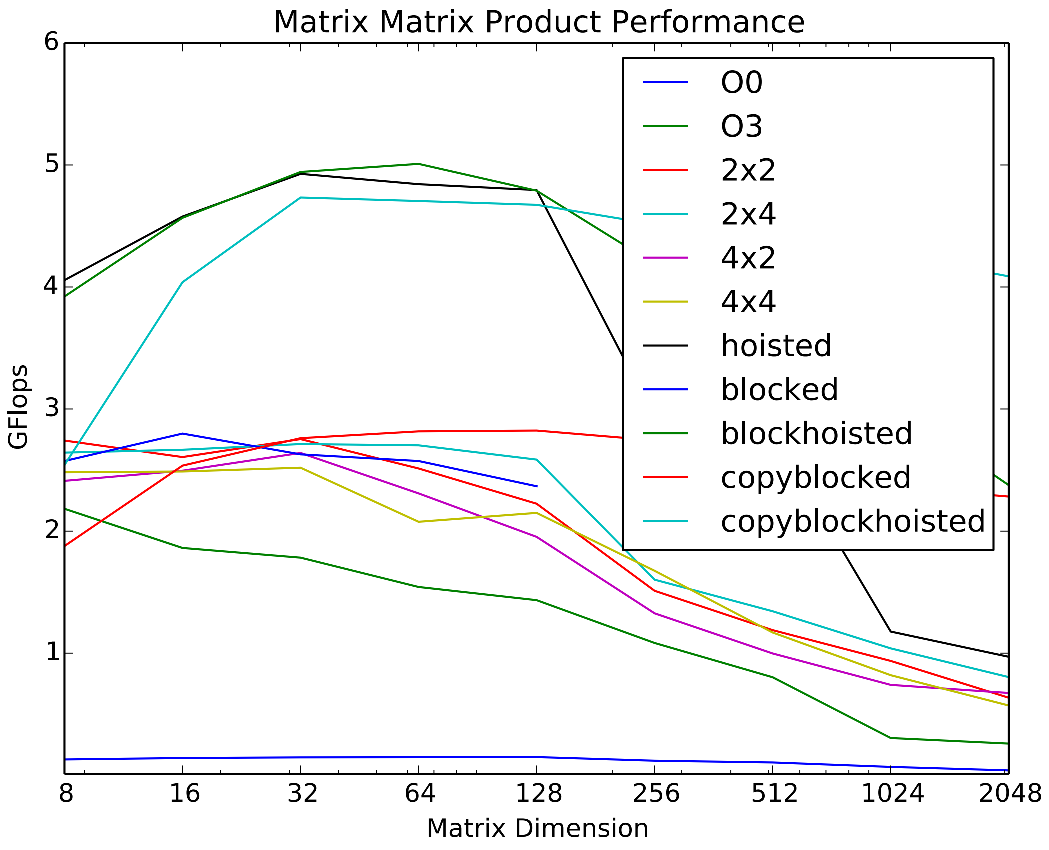
Copying and Transpose Multiply



Copying and Blocking and Tiling



Blocking and Tiling and Hoisting and Copying



Recap

- Locality: Write software so hardware can leverage it
- Register (temporal) locality (tiling / unrolling)
  - Hoisting
  - Blocking
- Spatial locality
  - Copying / transpose multiply
- Always use –O3 for release (not for debug)

Tuning

- Starting with base code
- Various compiler optimizations help
- Tiling (which size)
- Blocking (what size)
- What size works best for Tiling and Blocking together?
- What loop ordering? Matrix-matrix product has six different orderings? What block ordering?
- What about when we add vectorization, and threads, etc?

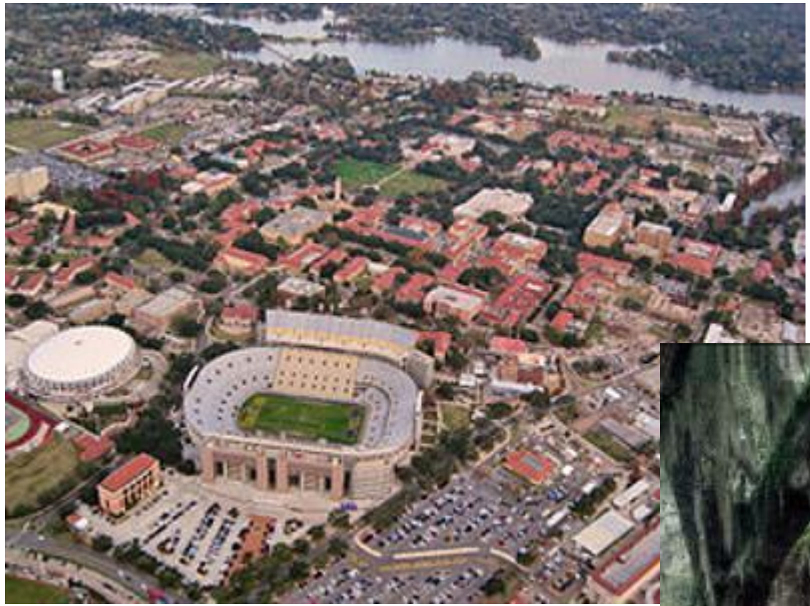
Magic: the power of apparently influencing the course of events by using mysterious or supernatural forces

- How do we find the optimal combination?
- The answer will be different for different CPUs

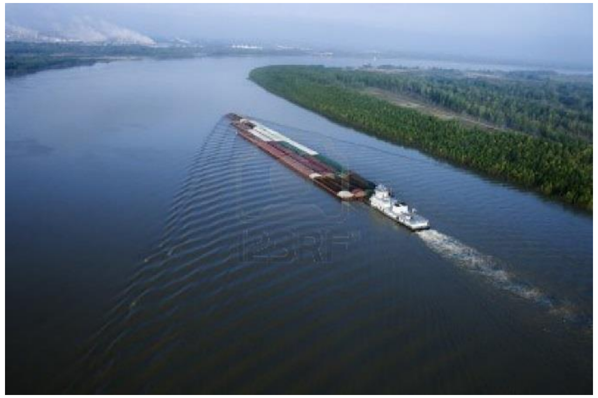
Finding the Sweet Spot

- Exhaustive parameter space search
  - Tiling, Blocking, Compiler flags, AVX instr, loop ordering
  - This started as a final course project
  - The competition was to write fastest matrix-matrix product
  - Students were the good kind of lazy
  - And wrote a program to generate different multiply functions

- Original project at UC Berkeley phiPAC (Bilmes et. Al.)
- Further developed by Whaley and Dongarra → Automatically Tuned Linear Algebra Subprograms (ATLAS)
  - Recently honored with "test of time" award



LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP