Dense Matrix Computations

Optimizing GEMM Operations in OpenMP

GEMM : *General-purpose Matrix-Matrix multiplication*

Cornerstone of many numerical algorithms
(including GPU-accelerated Deep Learning workloads)

Fast implementations available in MKL and other libraries

Great example for design of parallel optimizations
(including both multi-threading and SIMD)
as it's easy to prototype but trickier to optimize

Most clear example we've seen so far of
*a **compute-bound** kernel*

Theory of GEMM operation

*For simplicity : **A** and **B** are square NxN matrices*

*Each element of the matrix product **C = A*B** given as:*

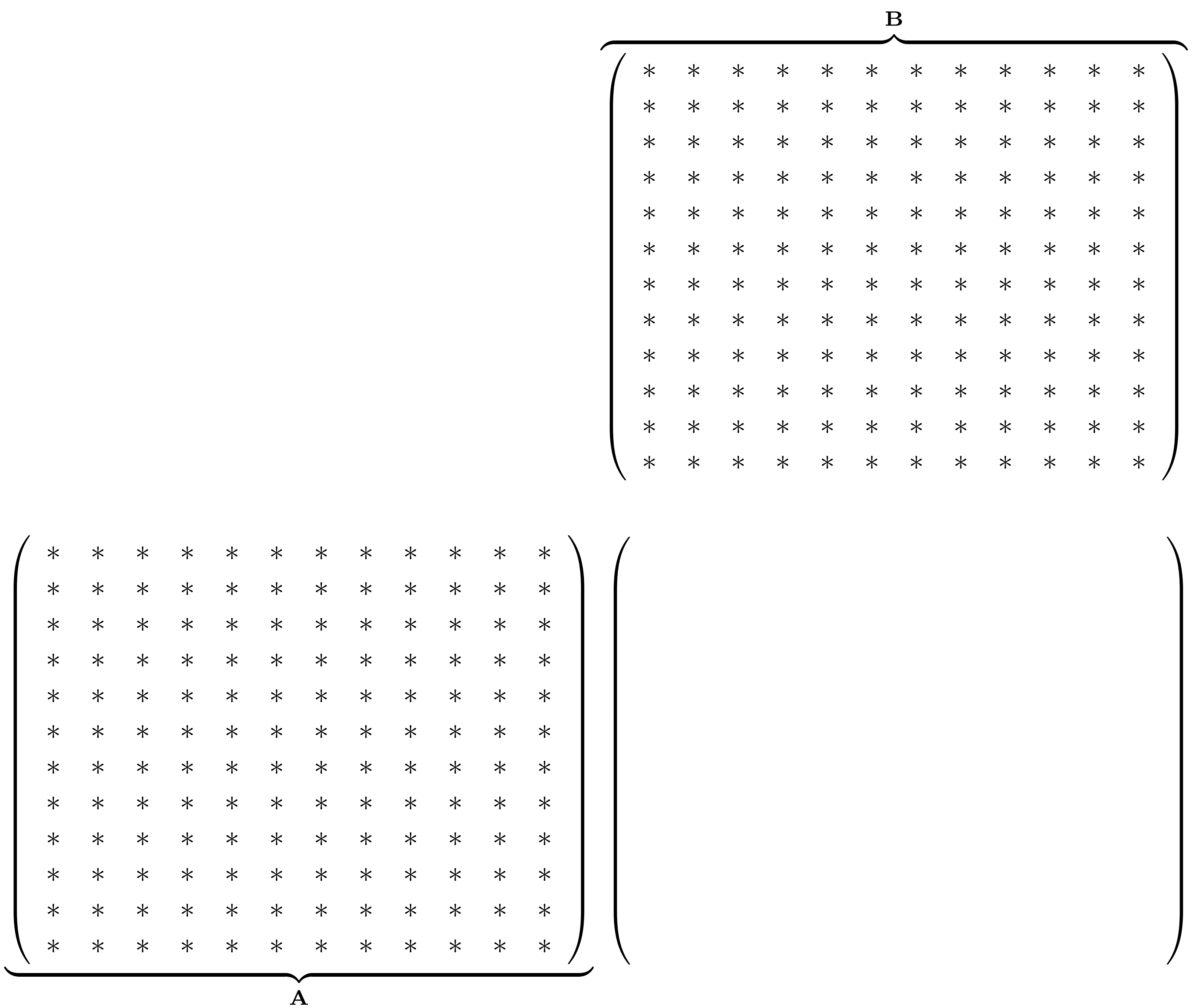
$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$$

In pseudocode:

$$
\begin{array}{l}
\texttt{for}\ \ i = 1 \ldots N \\
\quad \texttt{for}\ \ j = 1 \ldots N \\
\quad\quad C_{ij} \leftarrow 0 \\
\quad\quad \texttt{for}\ \ k = 1 \ldots N \\
\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} B_{kj}
\end{array}
$$

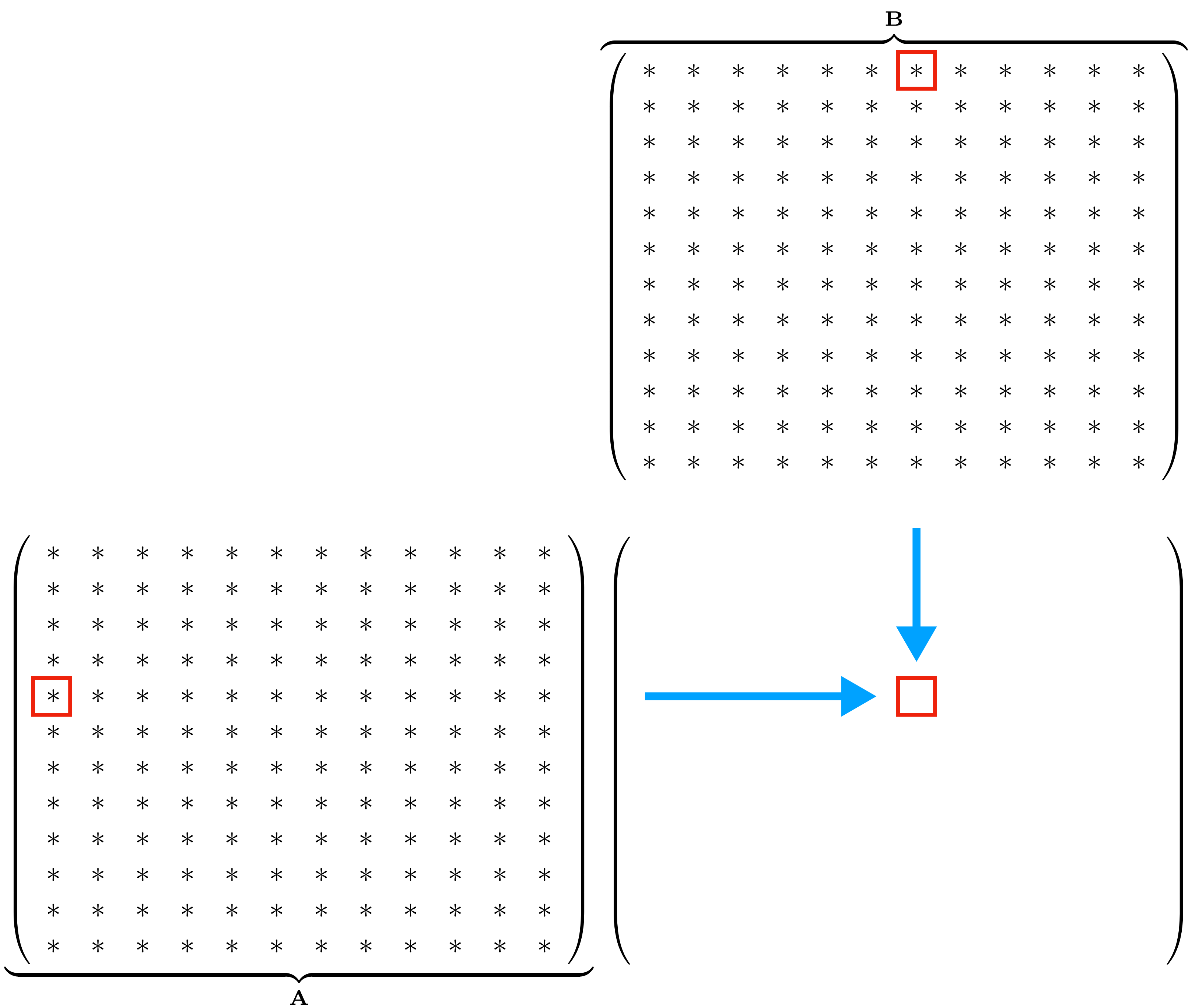
Theory of GEMM operation

A visual illustration …



Theory of GEMM operation

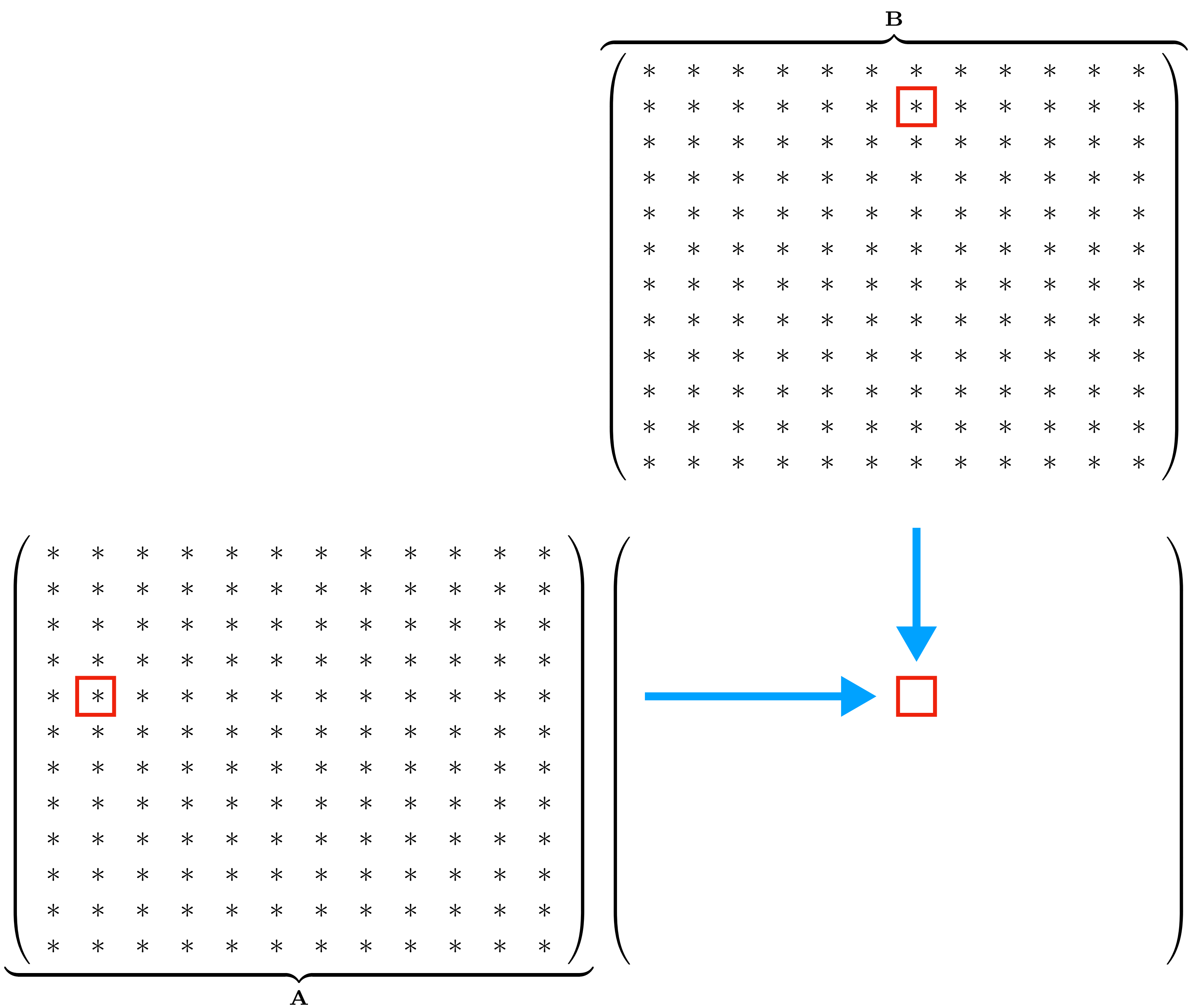
A visual illustration …



Multiply respective entries of ***A*** *&* ***B****,*
accumulate on highlighted entry of ***C=A*B***

Theory of GEMM operation

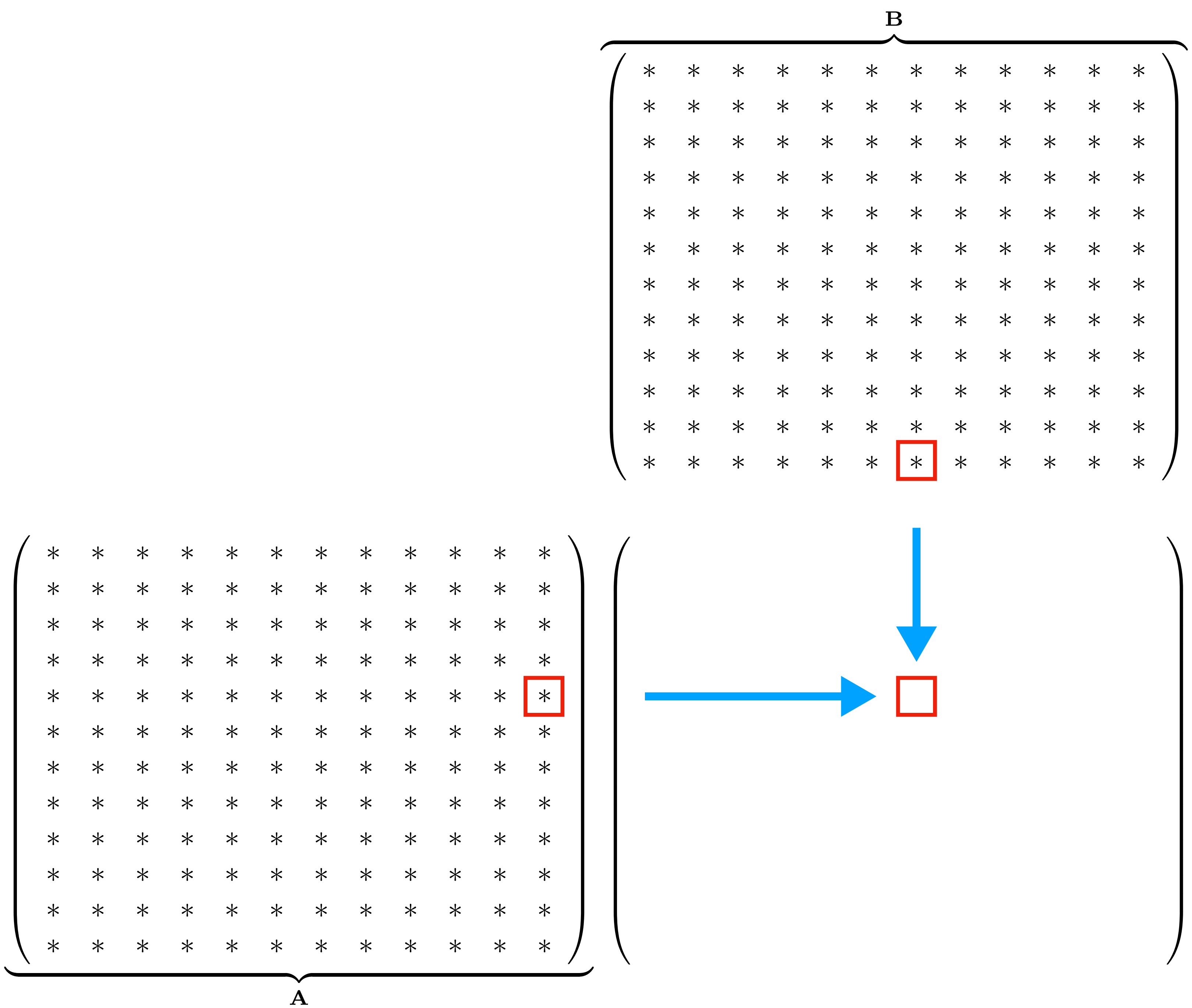
A visual illustration …



Multiply respective entries of ***A*** *&* ***B****,*
accumulate on highlighted entry of ***C=A*B***

Theory of GEMM operation

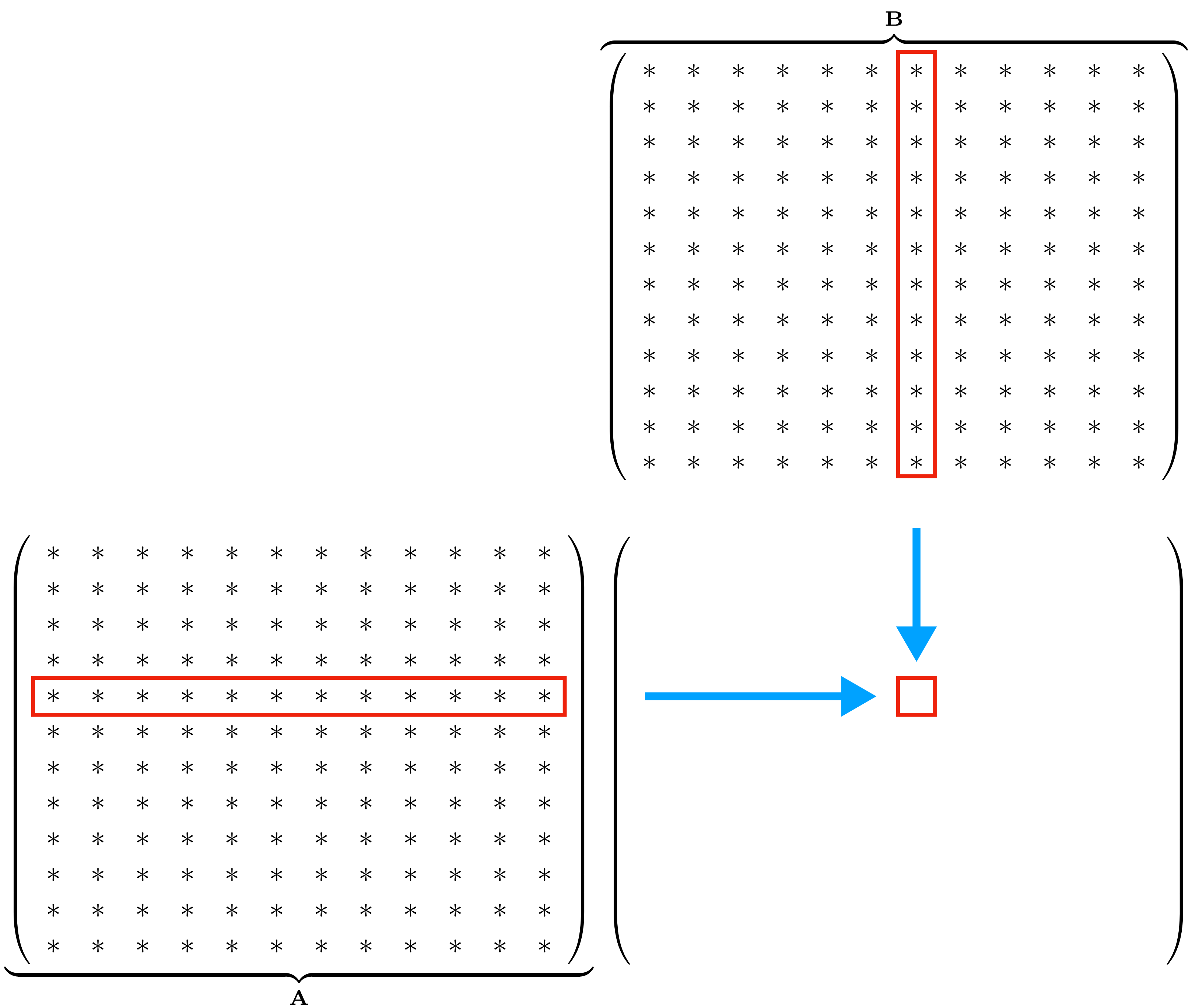
A visual illustration …



*Multiply respective entries of **A** & **B**,*
*accumulate on highlighted entry of **C=A*B***

Theory of GEMM operation

A visual illustration …



for $i = 1 \ldots N$
  for $j = 1 \ldots N$
    $C_{ij} \leftarrow 0$
    for $k = 1 \ldots N$
      $C_{ij} \leftarrow C_{ij} + A_{ik} B_{kj}$

GEMM : *General-purpose Matrix-Matrix multiplication*

Our objective today:

Transition from the "straightforward prototype" (suboptimal by ~100x)
to a somewhat competitive implementation (within ~4x of MKL)

Will require:
- Re-thinking the theory and the data layout
- Careful use of multithreading
- Some use of OpenMP-assisted vectorization

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_0

New test directory : Look at test GEMM_Test_0_[0-8]

```
#include "MatMatMultiply.h"
#include "Timer.h"
#include "Utilities.h"
#include <iostream>
#include <iomanip>

int main(int argc, char *argv[])
{
    float *Araw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Braw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Craw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));

    using matrix_t = float (&) [MATRIX_SIZE][MATRIX_SIZE];
    matrix_t A = reinterpret_cast<matrix_t>(*Araw);
    matrix_t B = reinterpret_cast<matrix_t>(*Braw);
    matrix_t C = reinterpret_cast<matrix_t>(*Craw);

    InitializeMatrices(A, B);
    Timer timer;

    for(int test = 1; test <= 10; test++){
        std::cout << "Running test iteration " << std::setw(2) << test << " ";
        timer.Start();
        MatMatMultiply(A, B, C);
        timer.Stop("Elapsed time : ");
    }

    return 0;
}
```

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_0

Allocates enough memory to fit a square matrix ***ensuring*** *that the allocated memory starts at a cache line (i.e. at a byte address multiple of 64)*

```
#include "MatMatMultiply.h"
#include "Timer.h"
#include "Utilities.h"
#include <iostream>
#include <iomanip>

int main(int argc, char *argv[])
{
    float *Araw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Braw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Craw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));

    using matrix_t = float (&) [MATRIX_SIZE][MATRIX_SIZE];
    matrix_t A = reinterpret_cast<matrix_t>(*Araw);
    matrix_t B = reinterpret_cast<matrix_t>(*Braw);
    matrix_t C = reinterpret_cast<matrix_t>(*Craw);

    InitializeMatrices(A, B);
    Timer timer;

    for(int test = 1; test <= 10; test++){
        std::cout << "Running test iteration " << std::setw(2) << test << " ";
        timer.Start();
        MatMatMultiply(A, B, C);
        timer.Stop("Elapsed time : ");
    }

    return 0;
}
```

Allocate/Initialize (Utilities.cpp)

DenseAlgebra/GEMM_Test_0_0

```
#include "Utilities.h"

#include <memory>
#include <new>
#include <random>

void* AlignedAllocate(const std::size_t size, const std::size_t alignment)
{
    std::size_t capacity = size + alignment - 1;
    void *ptr = new char[capacity];
    auto result = std::align(alignment, size, ptr, capacity);
    if (result == nullptr) throw std::bad_alloc();
    if (capacity < size) throw std::bad_alloc();
    return ptr;
}

void InitializeMatrices(float (&A)[MATRIX_SIZE][MATRIX_SIZE],float (&B)[MATRIX_SIZE][MATRIX_SIZE])
{
    std::random_device rd; std::mt19937 gen(rd());
    std::uniform_real_distribution<float> uniform_dist(-1., 1.);
    for (int i = 0; i < MATRIX_SIZE; i++)
        for (int j = 0; j < MATRIX_SIZE; j++) {
            A[i][j] = uniform_dist(gen);
            B[i][j] = uniform_dist(gen);
        }
}
```

RECAP

Allocate/Initialize (Utilities.cpp)

DenseAlgebra/GEMM_Test_0_0

```
#include "Utilities.h"

#include <memory>
#include <new>
#include <random>

void* AlignedAllocate(const std::size_t size, const std::size_t alignment)
{
    std::size_t capacity = size + alignment - 1;
    void *ptr = new char[capacity];
    auto result = std::align(alignment, size, ptr, capacity);
    if (result == nullptr) throw std::bad_alloc();
    if (capacity < size) throw std::bad_alloc();
    return ptr;
}

void InitializeMatrices(float (&A)[MATRI                    X_SIZE])
{
    std::random_device rd; std::mt19937
    std::uniform_real_distribution<float
    for (int i = 0; i < MATRIX_SIZE; i++
        for (int j = 0; j < MATRIX_SIZE;
            A[i][j] = uniform_dist(gen);
            B[i][j] = uniform_dist(gen);
        }
}
```

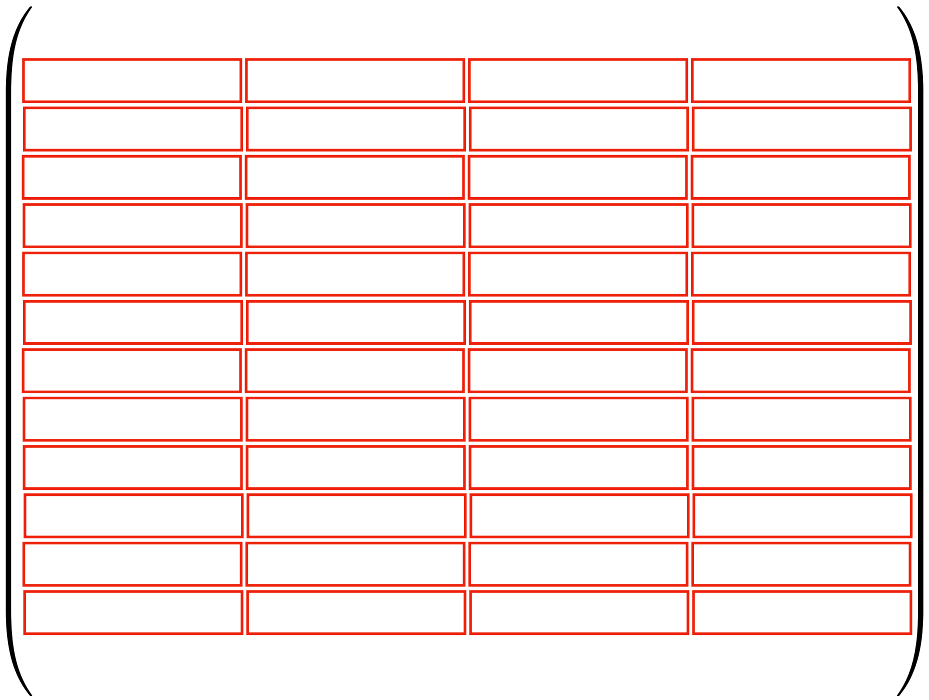
We allocate enough space so there's enough to "trim" the beginning to make it aligned (if you needed to explicitly delete the memory, you will need to also keep a pointer to the originally allocated memory)

RECAP

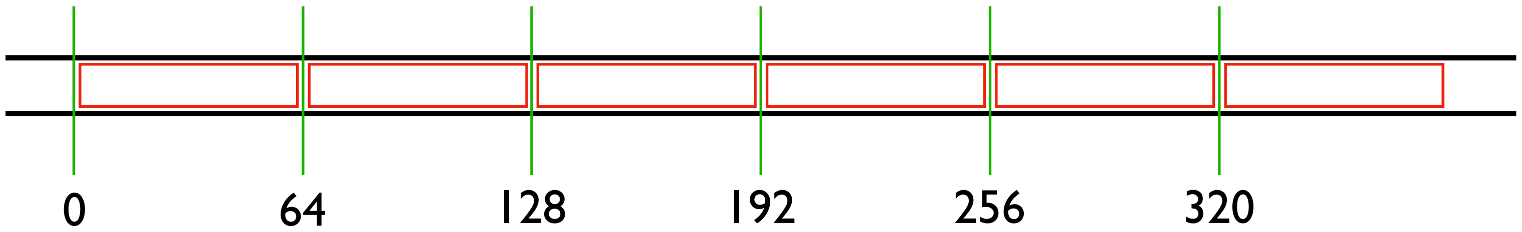
Impact of alignment

Assume:
- matrix is stored as row-major
- matrix dimension is multiple of 16
(16 floats = 64 bytes)

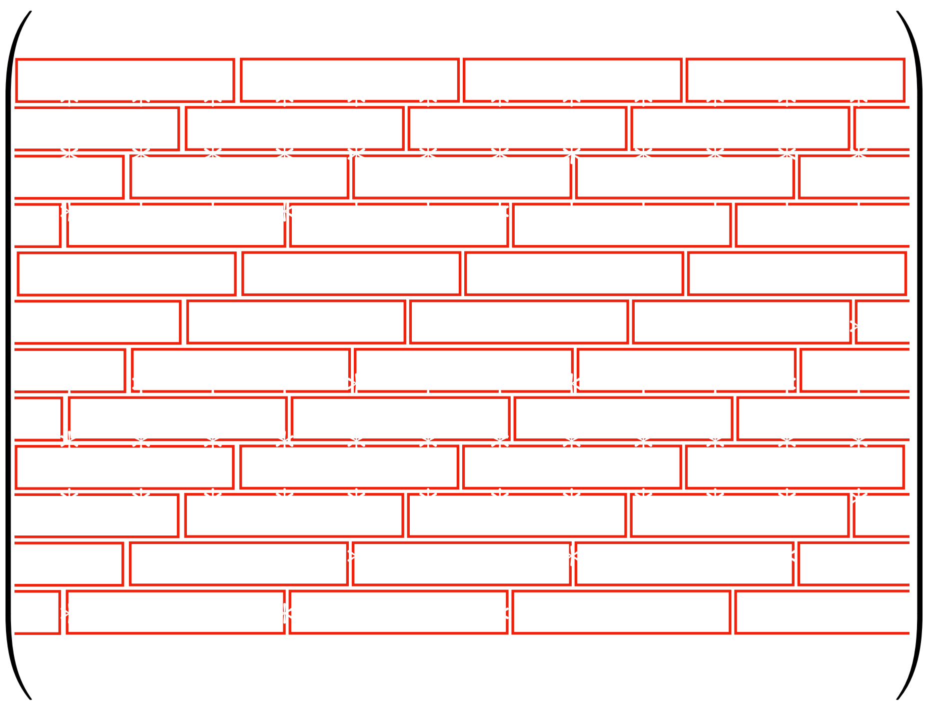
Impact of alignment



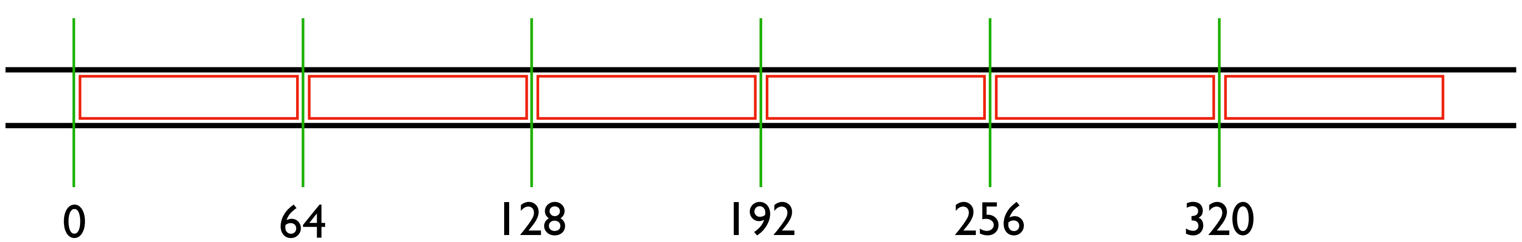
Assume:
- matrix is stored as row-major
- matrix dimension is multiple of 16 (16 floats = 64 bytes)



Impact of alignment



Assume:
- matrix is stored as row-major
- matrix dimension is multiple of 16
(16 floats = 64 bytes)



RECAP

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_0

```
#include "MatMatMultiply.h"
#include "Timer.h"
#include "Utilities.h"
#include <iostream>
#include <iomanip>

int main(int argc, char *argv[])
{
    float *Araw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Braw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Craw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));

    using matrix_t = float (&) [MATRIX_SIZE][MATRIX_SIZE];
    matrix_t A = reinterpret_cast<matrix_t>(*Araw);
    matrix_t B = reinterpret_cast<matrix_t>(*Braw);
    matrix_t C = reinterpret_cast<matrix_t>(*Craw);

    InitializeMatrices(A, B);
    Timer timer;

    for(int test = 1; test <= 10; test++){
        std::cout << "Running test iteration
        timer.Start();
        MatMatMultiply(A, B, C);
        timer.Stop("Elapsed time : ");
    }

    return 0;
}
```

Recast the allocated memory to a "matrix" that can be indexed just like an array (e.g. A[i][j] contains element (i,j) of the array)
Note: *This is effectively a Row Major matrix*

RECAP

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_0

```
#include "MatMatMultiply.h"
#include "Timer.h"
#include "Utilities.h"
#include <iostream>
#include <iomanip>

int main(int argc, char *argv[])
{
    float *Araw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Braw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Craw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));

    using matrix_t = float (&) [MATRIX_SIZE][MATRIX_SIZE];
    matrix_t A = reinterpret_cast<matrix_t>(*Araw);
    matrix_t B = reinterpret_cast<matrix_t>(*Braw);
    matrix_t C = reinterpret_cast<matrix_t>(*Craw);

    InitializeMatrices(A, B);
    Timer timer;

    for(int test = 1; test <= 10; test++){
        std::cout << "Running test iteration " << std::setw(2) << test << " ";
        timer.Start();
        MatMatMultiply(A, B, C);
        timer.Stop("Elapsed time : ");
    }

    return 0;
}
```

*Fill matrices **A & B** with random entries*

Allocate/Initialize (Utilities.cpp)

DenseAlgebra/GEMM_Test_0_0

```
#include "Utilities.h"

#include <memory>
#include <new>
#include <random>

void* AlignedAllocate(const std::size_t size, const std::size_t alignment)
{
    std::size_t capacity = size + alignment - 1;
    void *ptr = new char[capacity];
    auto result = std::align(alignment, size, ptr, capacity);
    if (result == nullptr) throw std::bad_alloc();
    if (capacity < size) throw std::bad_alloc();
    return ptr;
}

void InitializeMatrices(float (&A)[MATRIX_SIZE][MATRIX_SIZE],float (&B)[MATRIX_SIZE][MATRIX_SIZE])
{
    std::random_device rd; std::mt19937 gen(rd());
    std::uniform_real_distribution<float> uniform_dist(-1., 1.);
    for (int i = 0; i < MATRIX_SIZE; i++)
        for (int j = 0; j < MATRIX_SIZE; j++) {
            A[i][j] = uniform_dist(gen);
            B[i][j] = uniform_dist(gen);
        }
}
```

Fill each matrix with random entries between [-1, +1]

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_0

```
#include "MatMatMultiply.h"
#include "Timer.h"
#include "Utilities.h"
#include <iostream>
#include <iomanip>

int main(int argc, char *argv[])
{
    float *Araw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Braw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Craw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));

    using matrix_t = float (&) [MATRIX_SIZE][MATRIX_SIZE];
    matrix_t A = reinterpret_cast<matrix_t>(*Araw);
    matrix_t B = reinterpret_cast<matrix_t>(*Braw);
    matrix_t C = reinterpret_cast<matrix_t>(*Craw);

    InitializeMatrices(A, B);
    Timer timer;

    for(int test = 1; test <= 10; test++){
        std::cout << "Running test iteration " << std::setw(2) << test << " ";
        timer.Start();
        MatMatMultiply(A, B, C);
        timer.Stop("Elapsed time : ");
    }

    return 0;
}
```

Run & time the matrix-matrix multiplication operation ***C = A*B***

Kernel parameters (Parameters.h)

DenseAlgebra/GEMM_Test_0_0

```
#pragma once

#define MATRIX_SIZE 1024
```

*For simplicity, assume the size of the matrix is known at compile time, and all matrices involved are **square** with the same dimension as **MATRIX_SIZE***

RECAP

GEMM routine (MatMatMultiply.h)

DenseAlgebra/GEMM_Test_0_0

```
#pragma once

#include "Parameters.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]);
```

*This will essentially be the interface to our hand-implemented equivalent of the BLAS **GEMM** routine (general purpose Matrix-Matrix multiply)*

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_0

```
#include "MatMatMultiply.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
#pragma omp parallel for
    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++) {
        C[i][j] = 0.;
        for (int k = 0; k < MATRIX_SIZE; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
}
```

RECAP

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_0

```
#include "MatMatMultiply.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
#pragma omp parallel for
    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++) {
        C[i][j] = 0.;
        for (int k = 0; k < MATRIX_SIZE; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
}
```

$$
\begin{array}{l}
\texttt{for}\ \ i = 1 \ldots N \\
\quad \texttt{for}\ \ j = 1 \ldots N \\
\qquad C_{ij} \leftarrow 0 \\
\qquad \texttt{for}\ \ k = 1 \ldots N \\
\qquad\quad C_{ij} \leftarrow C_{ij} + A_{ik} B_{kj}
\end{array}
$$

As we saw in theory, the triple for-loop incurs O(N^3) complexity

RECAP

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_0

At matrix size = 1024

```
#include "MatMatMultiply.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
#pragma omp parallel for
    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++) {
        C[i][j] = 0.;
        for (int k = 0; k < MATRIX_SIZE; k
            C[i][j] += A[i][k] * B[k][j];
    }
}
```

Execution:

```
Running test iteration  1 [Elapsed time : 275.052ms]
Running test iteration  2 [Elapsed time : 245.782ms]
Running test iteration  3 [Elapsed time : 244.407ms]
Running test iteration  4 [Elapsed time : 245.818ms]
Running test iteration  5 [Elapsed time : 244.987ms]
Running test iteration  6 [Elapsed time : 244.948ms]
Running test iteration  7 [Elapsed time : 245.638ms]
Running test iteration  8 [Elapsed time : 245.293ms]
Running test iteration  9 [Elapsed time : 245.689ms]
Running test iteration 10 [Elapsed time : 245.317ms]
```

RECAP

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_0

At matrix size = 512

```
#include "MatMatMultiply.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
#pragma omp parallel for
    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++) {
        C[i][j] = 0.;
        for (int k = 0; k < MATRIX_SIZE; k
            C[i][j] += A[i][k] * B[k][j];
    }
}
```

Execution:

```
Running test iteration  1 [Elapsed time : 30.0578ms]
Running test iteration  2 [Elapsed time : 13.7184ms]
Running test iteration  3 [Elapsed time : 13.3553ms]
Running test iteration  4 [Elapsed time : 13.4283ms]
Running test iteration  5 [Elapsed time : 13.3111ms]
Running test iteration  6 [Elapsed time : 13.4193ms]
Running test iteration  7 [Elapsed time : 13.1227ms]
Running test iteration  8 [Elapsed time : 13.5121ms]
Running test iteration  9 [Elapsed time : 13.248ms]
Running test iteration 10 [Elapsed time : 12.7863ms]
```

RECAP

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_0

At matrix size = 2048

```
#include "MatMatMultiply.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
#pragma omp parallel for
    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++) {
        C[i][j] = 0.;
        for (int k = 0; k < MATRIX_SIZE; k
            C[i][j] += A[i][k] * B[k][j];
    }
}
```

Execution:

```
Running test iteration  1 [Elapsed time : 1940.06ms]
Running test iteration  2 [Elapsed time : 1899.59ms]
Running test iteration  3 [Elapsed time : 1895.68ms]
Running test iteration  4 [Elapsed time : 1898.08ms]
Running test iteration  5 [Elapsed time : 1897.78ms]
Running test iteration  6 [Elapsed time : 1898.02ms]
Running test iteration  7 [Elapsed time : 1899.11ms]
Running test iteration  8 [Elapsed time : 1897.91ms]
Running test iteration  9 [Elapsed time : 1898.76ms]
Running test iteration 10 [Elapsed time : 1899.64ms]
```

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_1

```
#include "MatMatMultiply.h"
#include "mkl.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
    cblas_sgemm(
        CblasRowMajor,
        CblasNoTrans,
        CblasNoTrans,
        MATRIX_SIZE,
        MATRIX_SIZE,
        MATRIX_SIZE,
        1.,
        &A[0][0],
        MATRIX_SIZE,
        &B[0][0],
        MATRIX_SIZE,
        0.,
        &C[0][0],
        MATRIX_SIZE
    );
}
```

*We have replaced our hand-implemented code with a call to the BLAS **GEMM** routine*

cblas_?gemm

Computes a matrix-matrix product with general matrices.

Syntax

void cblas_sgemm (const CBLAS_LAYOUT *Layout*, const CBLAS_TRANSPOSE *transa*, const CBLAS_TRANSPOSE *transb*, const MKL_INT *m*, const MKL_INT *n*, const MKL_INT *k*, const float *alpha*, const float **a*, const MKL_INT *lda*, const float **b*, const MKL_INT *ldb*, const floa *beta*, float **c*, const MKL_INT *ldc*);

void cblas_dgemm (const CBLAS_LAYOUT *Layout*, const CBLAS_TRANSPOSE *transa*, const CBLAS_TRANSPOSE *transb*, const MKL_INT *m*, const MKL_INT *n*, const MKL_INT *k*, const doubl *alpha*, const double **a*, const MKL_INT *lda*, const double **b*, const MKL_INT *ldb*, const double *beta*, double **c*, const MKL_INT *ldc*);

void cblas_cgemm (const CBLAS_LAYOUT *Layout*, const CBLAS_TRANSPOSE *transa*, const CBLAS_TRANSPOSE *transb*, const MKL_INT *m*, const MKL_INT *n*, const MKL_INT *k*, const void **alpha*, const void **a*, const MKL_INT *lda*, const void **b*, const MKL_INT *ldb*, const void **beta*, void **c*, const MKL_INT *ldc*);

void cblas_zgemm (const CBLAS_LAYOUT *Layout*, const CBLAS_TRANSPOSE *transa*, const CBLAS_TRANSPOSE *transb*, const MKL_INT *m*, const MKL_INT *n*, const MKL_INT *k*, const void **alpha*, const void **a*, const MKL_INT *lda*, const void **b*, const MKL_INT *ldb*, const void **beta*, void **c*, const MKL_INT *ldc*);

RECAP

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_1

```
#include "MatMatMultiply.h"
#include "mkl.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
    cblas_sgemm(
        CblasRowMajor,
        CblasNoTrans,
        CblasNoTrans,
        MATRIX_SIZE,
        MATRIX_SIZE,
        MATRIX_SIZE,
        1.,
        &A[0][0],
        MATRIX_SIZE,
        &B[0][0],
        MATRIX_SIZE,
        0.,
        &C[0][0],
        MATRIX_SIZE
    );
}
```

*We have replaced our hand-implemented code with a call to the BLAS **GEMM** routine*

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_1

```
#include "MatMatMultiply.h"
#include "mkl.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
    cblas_sgemm(
        CblasRowMajor,
        CblasNoTrans,
        CblasNoTrans,
        MATRIX_SIZE,
        MATRIX_SIZE,
        MATRIX_SIZE,
        1.,
        &A[0][0],
        MATRIX_SIZE,
        &B[0][0],
        MATRIX_SIZE,
        0.,
        &C[0][0],
        MATRIX_SIZE
    );
}
```

At matrix size = 1024

Execution:

```
Running test iteration  1 [Elapsed time : 42.4088ms]
Running test iteration  2 [Elapsed time : 3.33403ms]
Running test iteration  3 [Elapsed time : 2.29802ms]
Running test iteration  4 [Elapsed time : 2.22505ms]
Running test iteration  5 [Elapsed time : 2.21731ms]
Running test iteration  6 [Elapsed time : 1.96854ms]
Running test iteration  7 [Elapsed time : 1.87623ms]
Running test iteration  8 [Elapsed time : 1.91837ms]
Running test iteration  9 [Elapsed time : 1.91348ms]
Running test iteration 10 [Elapsed time : 1.90199ms]
```

RECAP

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_1

```
#include "MatMatMultiply.h"
#include "mkl.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
    cblas_sgemm(
        CblasRowMajor,
        CblasNoTrans,
        CblasNoTrans,
        MATRIX_SIZE,
        MATRIX_SIZE,
        MATRIX_SIZE,
        1.,
        &A[0][0],
        MATRIX_SIZE,
        &B[0][0],
        MATRIX_SIZE,
        0.,
        &C[0][0],
        MATRIX_SIZE
    );
}
```

At matrix size = 2048

Execution:

```
Running test iteration  1 [Elapsed time : 61.1167ms]
Running test iteration  2 [Elapsed time : 14.2691ms]
Running test iteration  3 [Elapsed time : 14.1298ms]
Running test iteration  4 [Elapsed time : 14.2985ms]
Running test iteration  5 [Elapsed time : 14.2199ms]
Running test iteration  6 [Elapsed time : 14.0035ms]
Running test iteration  7 [Elapsed time : 14.2607ms]
Running test iteration  8 [Elapsed time : 14.0081ms]
Running test iteration  9 [Elapsed time : 15.484ms]
Running test iteration 10 [Elapsed time : 12.076ms]
```

GEMM routine (MatMatMultiply.cpp)

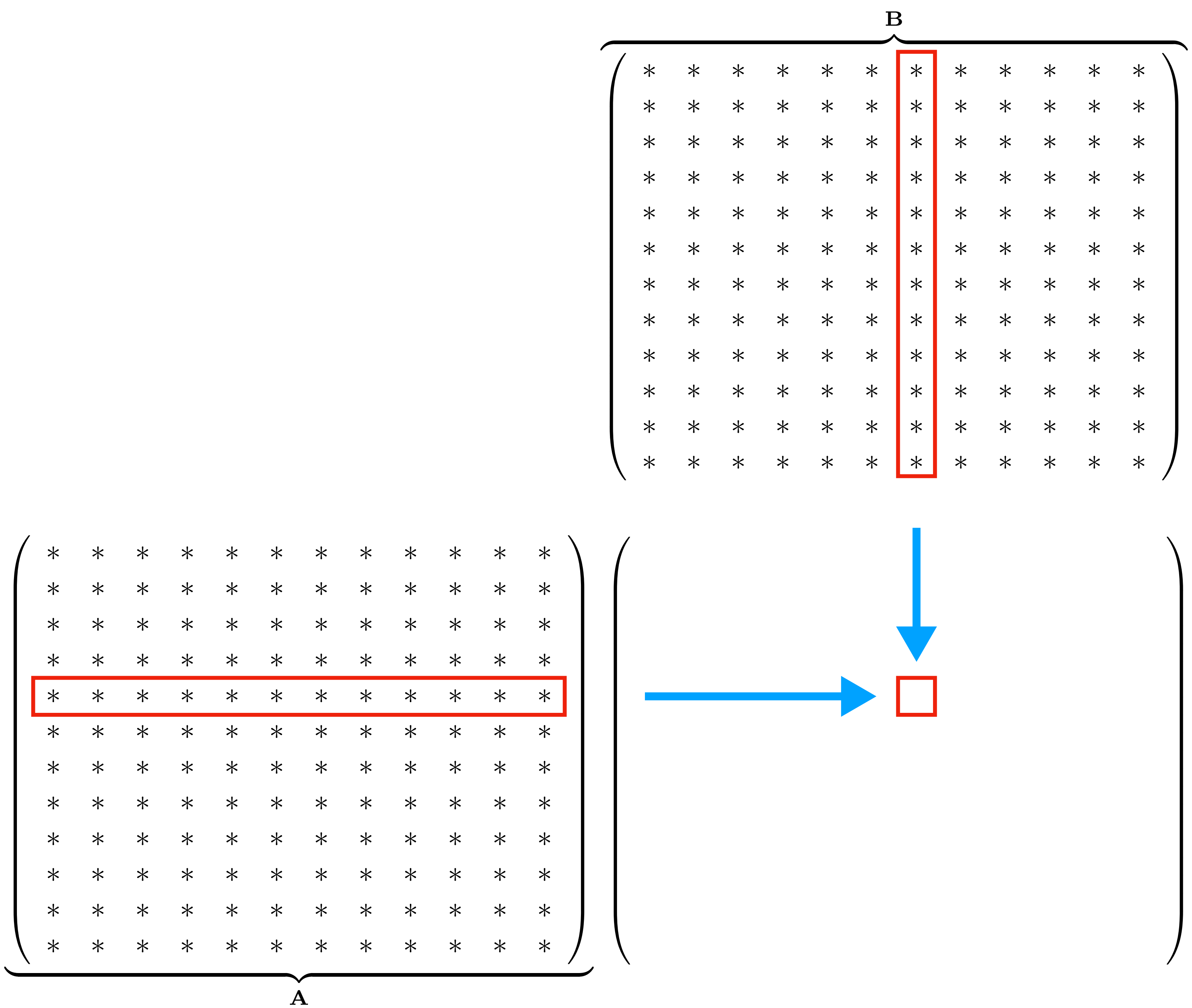
DenseAlgebra/GEMM_Test_0_2

```
#include "MatMatMultiply.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
    static constexpr int NBLOCKS = MATRIX_SIZE / BLOCK_SIZE;

    using blocked_matrix_t = float (&) [NBLOCKS][BLOCK_SIZE][NBLOCKS][BLOCK_SIZE];
    using const_blocked_matrix_t = const float (&) [NBLOCKS][BLOCK_SIZE][NBLOCKS][BLOCK_SIZE];
    auto blockA = reinterpret_cast<const_blocked_matrix_t>(A[0][0]);
    auto blockB = reinterpret_cast<const_blocked_matrix_t>(B[0][0]);
    auto blockC = reinterpret_cast<blocked_matrix_t>(C[0][0]);

#pragma omp parallel for
    for (int i = 0; i < NBLOCKS; i++)
    for (int j = 0; j < NBLOCKS; j++)
        C[i][j] = 0.;

    for (int bi = 0; bi < NBLOCKS; bi++)
    for (int bj = 0; bj < NBLOCKS; bj++)
        for (int bk = 0; bk < NBLOCKS; bk++) {

#pragma omp parallel for
            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
            for (int kk = 0; kk < BLOCK_SIZE; kk++)
                blockC[bi][ii][bj][jj] += blockA[bi][ii][bk][kk] * blockB[bk][kk][bj][jj];

        }
}
```

Adjusting our implementation to a “blocked” concept of matrix-matrix multiply

Theory of GEMM operation

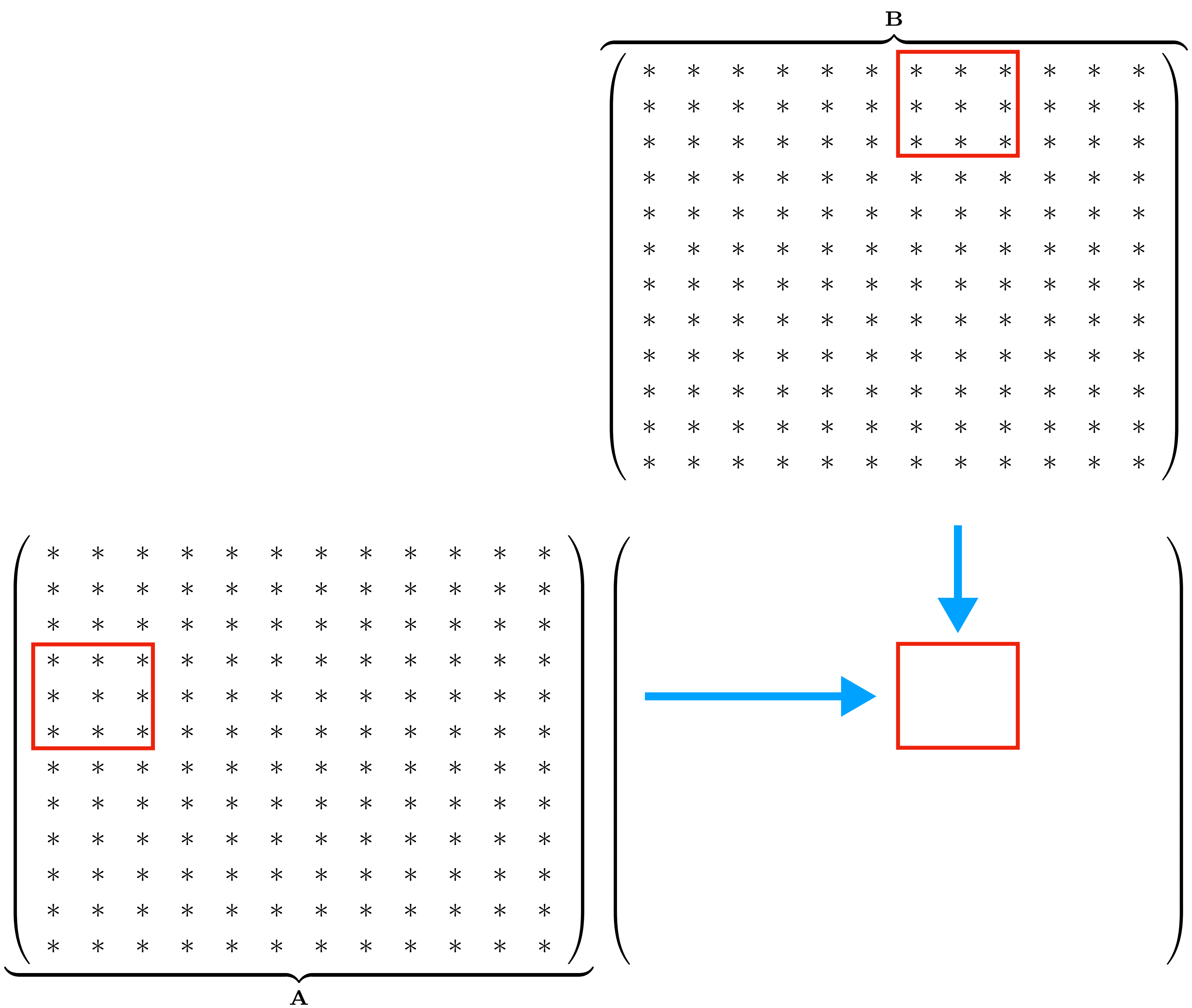
A visual illustration …



for $i = 1 \ldots N$
    for $j = 1 \ldots N$
        $C_{ij} \leftarrow 0$
        for $k = 1 \ldots N$
            $C_{ij} \leftarrow C_{ij} + A_{ik} B_{kj}$

Theory of GEMM operation

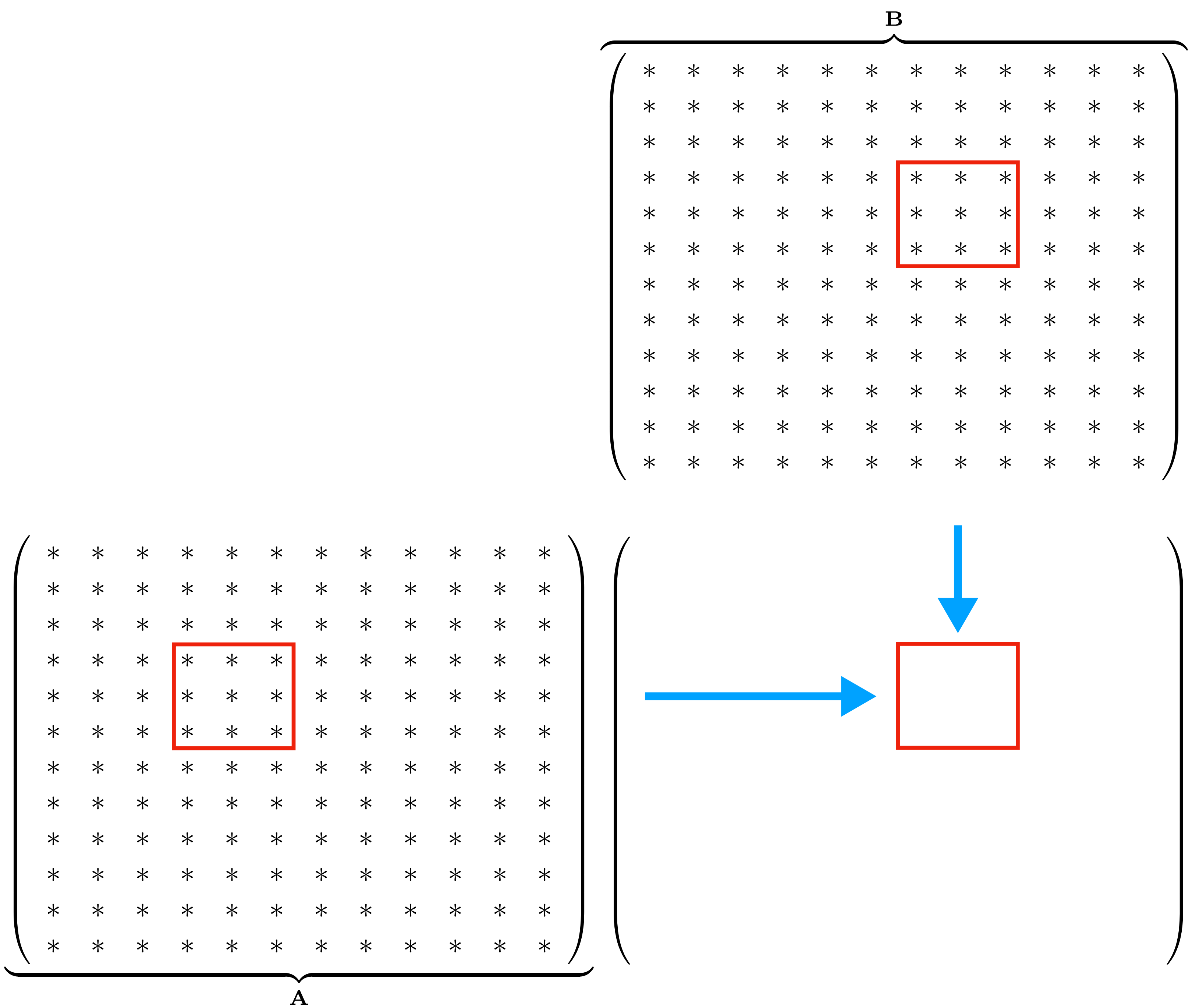
A visual illustration …



Multiply respective ***sub-matrices (blocks)*** *of* ***A*** *&* ***B****,*
accumulate on highlighted ***block*** *of* ***C=A*B***

Theory of GEMM operation

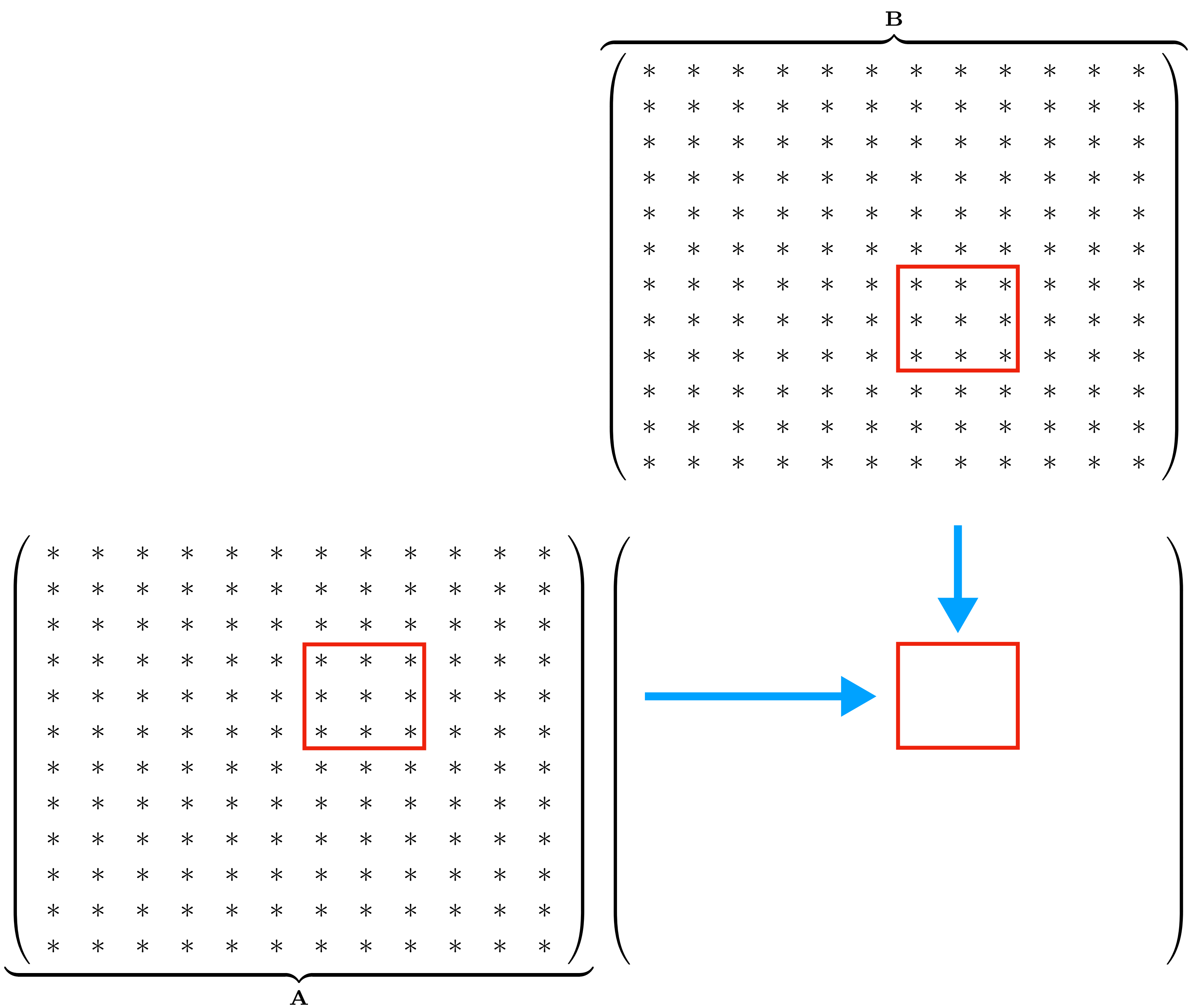
A visual illustration …



*Multiply respective **sub-matrices (blocks)** of **A** & **B**,*
*accumulate on highlighted **block** of **C=A*B***

Theory of GEMM operation

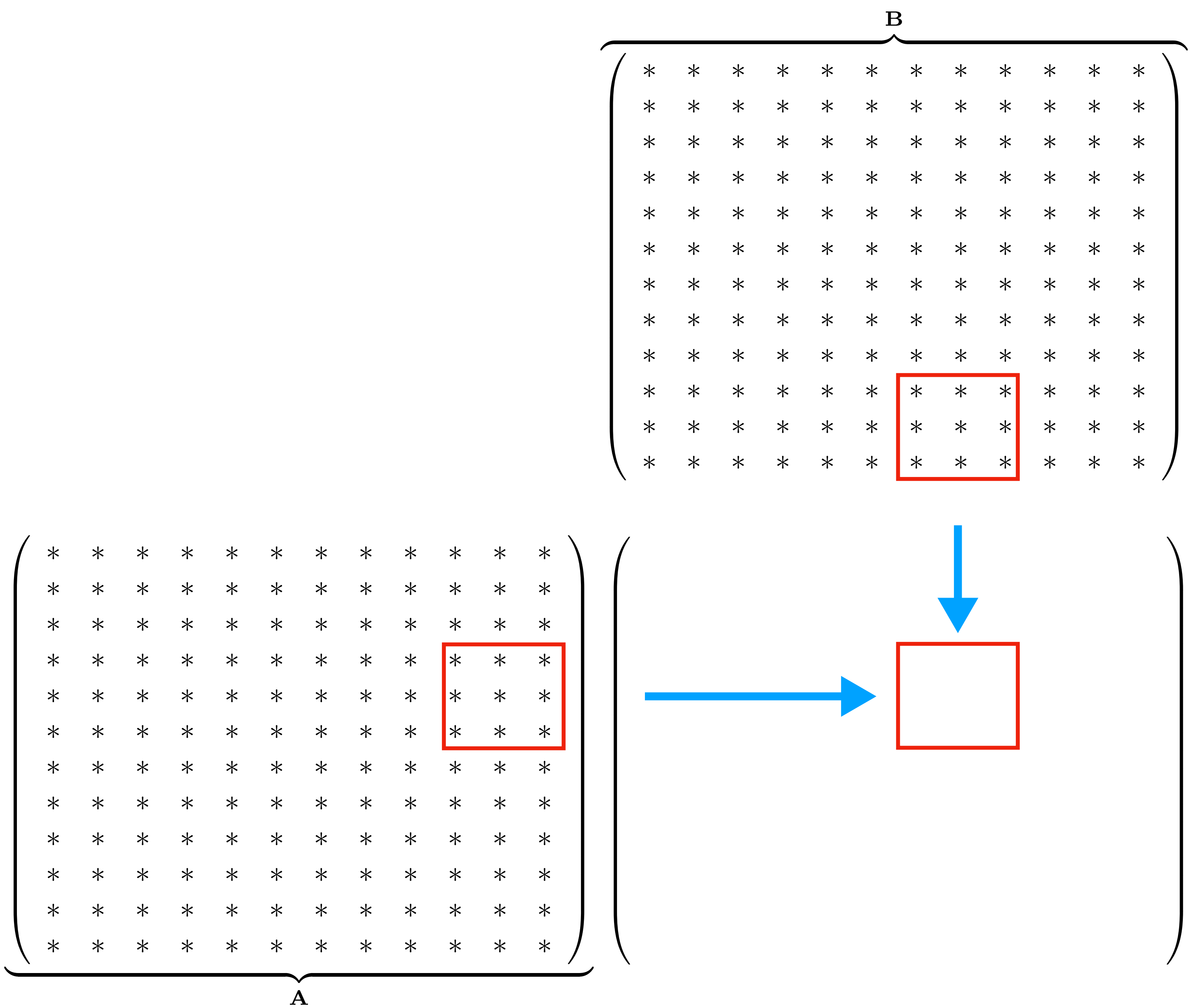
A visual illustration …



Multiply respective ***sub-matrices (blocks)*** *of* ***A*** *&* ***B****,*
accumulate on highlighted ***block*** *of* ***C=A*B***

Theory of GEMM operation

A visual illustration …

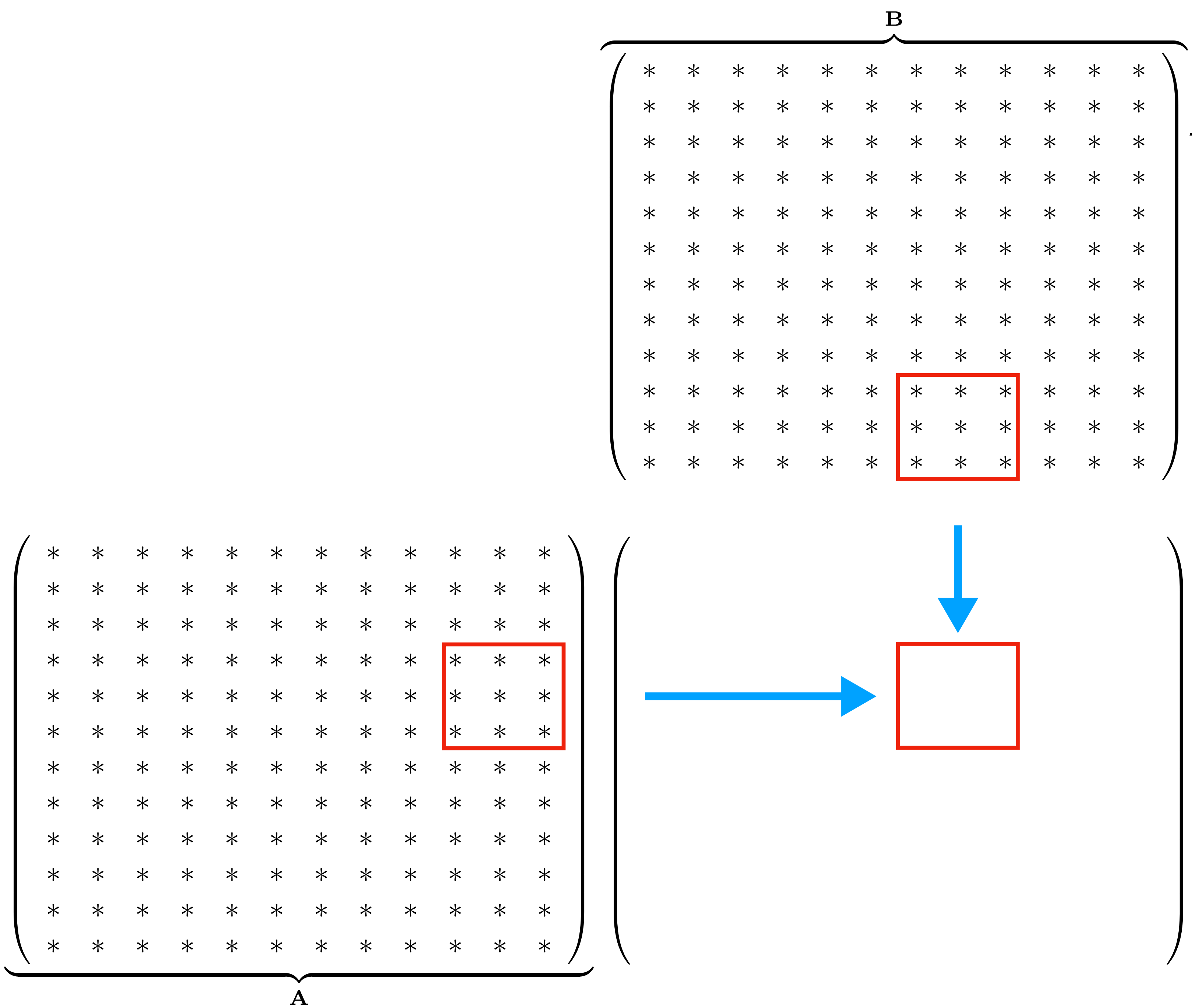



```
for i = 1...N
    for j = 1...N
        C_ij ← 0
        for k = 1...N
            C_ij ← C_ij + A_ik B_kj
```

$$
\begin{aligned}
&\texttt{for}\ \ i = 1 \ldots N \\
&\quad \texttt{for}\ \ j = 1 \ldots N \\
&\qquad C_{ij} \leftarrow 0 \\
&\qquad \texttt{for}\ \ k = 1 \ldots N \\
&\qquad\quad C_{ij} \leftarrow C_{ij} + A_{ik} B_{kj}
\end{aligned}
$$

Here Cij, Aik, and Bkj denote ***matrix blocks***

Theory of GEMM operation



Rationale for "blocking"

*- Assume we have K x K blocks, each of size B x B (N = K*B)*

- Instead of the cost being N^3 multiplications (forget additions for now) now the cost is K^3 matrix multiplications

- For every matrix multiplication, we pay the cost for B^3 multiplications (of floats) which doesn't change from before

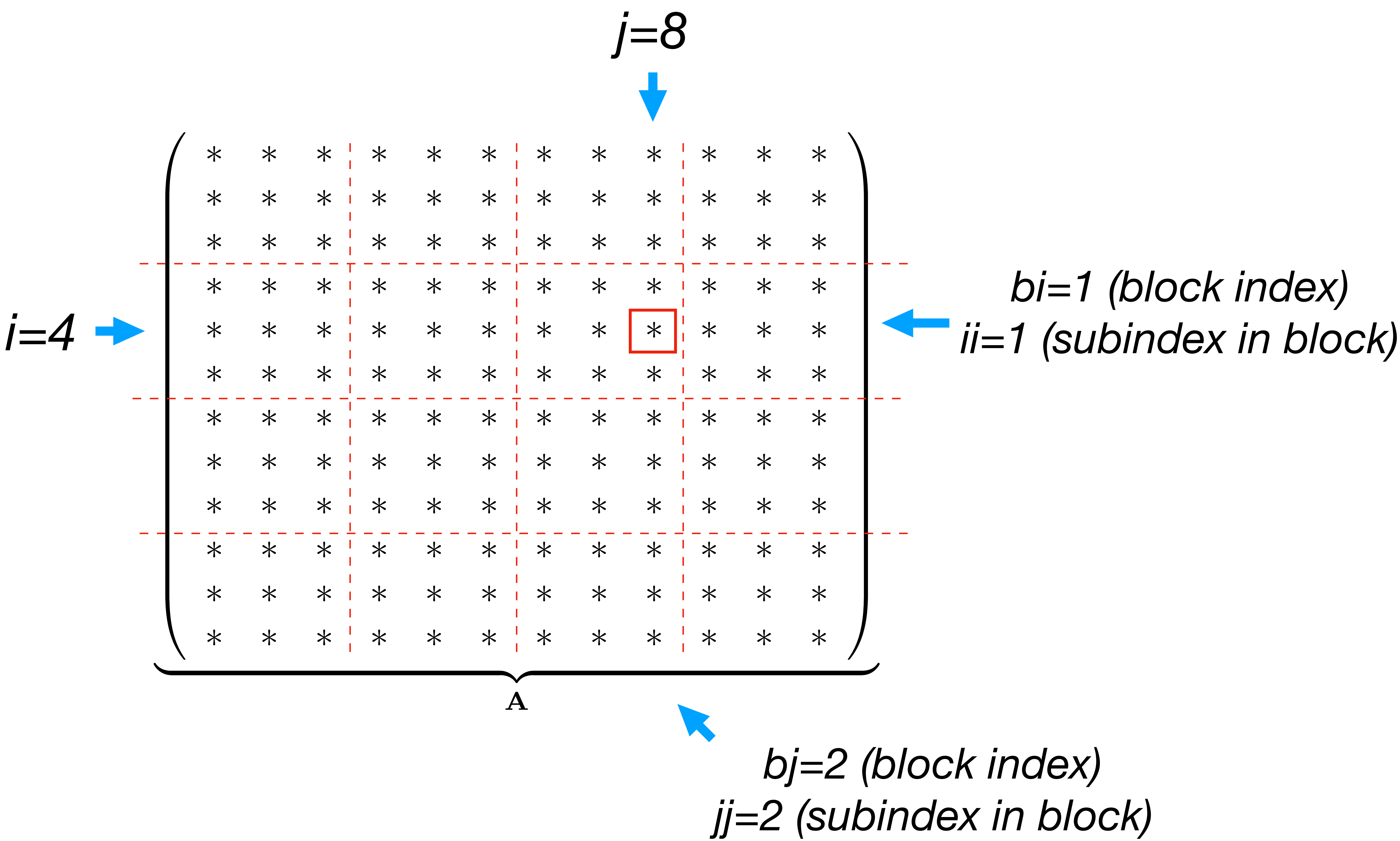
- However, previously we also paid the cost for B^3 memory accesses (the matrix couldn't be cached)

- If the block is small enough, maybe if can fit in cache, and the memory access cost drops to B^2 instead!

"Blocked" indexing of matrix entries

$$\underbrace{\left(\begin{array}{cccccc:ccc:ccc}
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & *
\end{array}\right)}_{\mathbf{A}}$$

“Blocked” indexing of matrix entries



GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_2

```
#include "MatMatMultiply.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
    static constexpr int NBLOCKS = MATRIX_SIZE / BLOCK_SIZE;

    using blocked_matrix_t = float (&) [NBLOCKS][BLOCK_SIZE][NBLOCKS][BLOCK_SIZE];
    using const_blocked_matrix_t = const float (&) [NBLOCKS][BLOCK_SIZE][NBLOCKS][BLOCK_SIZE];
    auto blockA = reinterpret_cast<const_blocked_matrix_t>(A[0][0]);
    auto blockB = reinterpret_cast<const_blocked_matrix_t>(B[0][0]);
    auto blockC = reinterpret_cast<blocked_matrix_t>(C[0][0]);

#pragma omp parallel for
    for (int i = 0; i < NBLOCKS; i++)
    for (int j = 0; j < NBLOCKS; j++)
        C[i][j] = 0.;

    for (int bi = 0; bi < NBLOCKS; bi++)
    for (int bj = 0; bj < NBLOCKS; bj++)
        for (int bk = 0; bk < NBLOCKS; bk++) {

#pragma omp parallel for
            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
            for (int kk = 0; kk < BLOCK_SIZE; kk++)
                blockC[bi][ii][bj][jj] += blockA[bi][ii][bk][kk] * blockB[bk][kk][bj][jj];

        }
}
```

Cast matrices such that they can be indexed using four numbers as follows: [row block][row subindex][col block][col subindex]

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_2

```
#include "MatMatMultiply.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
    static constexpr int NBLOCKS = MATRIX_SIZE / BLOCK_SIZE;

    using blocked_matrix_t = float (&) [NBLOCKS][BLOCK_SIZE][NBLOCKS][BLOCK_SIZE];
    using const_blocked_matrix_t = const float (&) [NBLOCKS][BLOCK_SIZE][NBLOCKS][BLOCK_SIZE];
    auto blockA = reinterpret_cast<const_blocked_matrix_t>(A[0][0]);
    auto blockB = reinterpret_cast<const_blocked_matrix_t>(B[0][0]);
    auto blockC = reinterpret_cast<blocked_matrix_t>(C[0][0]);

#pragma omp parallel for
    for (int i = 0; i < NBLOCKS; i++)
    for (int j = 0; j < NBLOCKS; j++)
        C[i][j] = 0.;

    for (int bi = 0; bi < NBLOCKS; bi++)
    for (int bj = 0; bj < NBLOCKS; bj++)
        for (int bk = 0; bk < NBLOCKS; bk++) {

#pragma omp parallel for
            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
            for (int kk = 0; kk < BLOCK_SIZE; kk++)
                blockC[bi][ii][bj][jj] += blockA[bi][ii][bk][kk] * blockB[bk][kk][bj][jj];

        }
}
```

Compare this line …

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_0

```
#include "MatMatMultiply.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
#pragma omp parallel for
    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++) {
        C[i][j] = 0.;
        for (int k = 0; k < MATRIX_SIZE; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
}
```

… with this line (they do the same thing, the latest implementation does it one-block-at-a-time)

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_2

```
#include "MatMatMultiply.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
    static constexpr int NBLOCKS = MATRIX_SIZE / BLOCK_SIZE;

    using blocked_matrix_t = float (&) [NBLOCKS][BLOCK_SIZE][NBLOCKS][BLOCK_SIZE];
    using const_blocked_matrix_t = const float (&) [NBLOCKS][BLOCK_SIZE][NBLOCKS][BLOCK_SIZE];
    auto blockA = reinterpret_cast<const_blocked_matrix_t>(A[0][0]);
    auto blockB = reinterpret_cast<const_blocked_matrix_t>(B[0][0]);
    auto blockC = reinterpret_cast<blocked_matrix_t>(C[0][0]);

#pragma omp parallel for
    for (int i = 0; i < NBLOCKS; i++)
    for (int j = 0; j < NBLOCKS; j++)
        C[i][j] = 0.;

    for (int bi = 0; bi < NBLOCKS; bi++)
    for (int bj = 0; bj < NBLOCKS; bj++)
        for (int bk = 0; bk < NBLOCKS; bk+

#pragma omp parallel for
            for (int ii = 0; ii < BLOCK_SI
            for (int jj = 0; jj < BLOCK_SI
            for (int kk = 0; kk < BLOCK_SI
                blockC[bi][ii][bj][jj] +=

        }
}
```

At matrix size = 1024

Execution:

```
Running test iteration  1 [Elapsed time : 171.81ms]
Running test iteration  2 [Elapsed time : 134.102ms]
Running test iteration  3 [Elapsed time : 133.837ms]
Running test iteration  4 [Elapsed time : 134.035ms]
Running test iteration  5 [Elapsed time : 134.137ms]
Running test iteration  6 [Elapsed time : 139.447ms]
Running test iteration  7 [Elapsed time : 133.784ms]
Running test iteration  8 [Elapsed time : 134.448ms]
Running test iteration  9 [Elapsed time : 134.428ms]
Running test iteration 10 [Elapsed time : 164.302ms]
```

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_3

```
#include "MatMatMultiply.h"
#include "Timer.h"
#include "Utilities.h"
#include <iostream>
#include <iomanip>

int main(int argc, char *argv[])
{
    float *Araw =
        static_cast<float*>( AlignedAllocate( MATRIX_SIZE * MATRIX_SIZE * sizeof(float), 64 ) );
    float *Braw =
        static_cast<float*>( AlignedAllocate( MATRIX_SIZE * MATRIX_SIZE * sizeof(float), 64 ) );
    float *Craw =
        static_cast<float*>( AlignedAllocate( MATRIX_SIZE * MATRIX_SIZE * sizeof(float), 64 ) );
    float *referenceCraw =
        static_cast<float*>( AlignedAllocate( MATRIX_SIZE * MATRIX_SIZE * sizeof(float), 64 ) );

    using matrix_t = float (&) [MATRIX_SIZE][MATRIX_SIZE];

    matrix_t A = reinterpret_cast<matrix_t>(*Araw);
    matrix_t B = reinterpret_cast<matrix_t>(*Braw);
    matrix_t C = reinterpret_cast<matrix_t>(*Craw);
    matrix_t referenceC = reinterpret_cast<matrix_t>(*referenceCraw);

    InitializeMatrices(A, B);
    Timer timer;

    [...]
}
```

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_3

__Very Important:__
Build infrastructure for testing correctness before implementing code transformations

```
#include "MatMatMultiply.h"
#include "Timer.h"
#include "Utilities.h"
#include <iostream>
#include <iomanip>

int main(int argc, char *argv[])
{
    float *Araw =
        static_cast<float*>( AlignedAllocate( MATRIX_SIZE * MATRIX_SIZE * sizeof(float), 64 ) );
    float *Braw =
        static_cast<float*>( AlignedAllocate( MATRIX_SIZE * MATRIX_SIZE * sizeof(float), 64 ) );
    float *Craw =
        static_cast<float*>( AlignedAllocate( MATRIX_SIZE * MATRIX_SIZE * sizeof(float), 64 ) );
    float *referenceCraw =
        static_cast<float*>( AlignedAllocate( MATRIX_SIZE * MATRIX_SIZE * sizeof(float), 64 ) );

    using matrix_t = float (&) [MATRIX_SIZE][MATRIX_SIZE];

    matrix_t A = reinterpret_cast<matrix_t>(*Araw);
    matrix_t B = reinterpret_cast<matrix_t>(*Braw);
    matrix_t C = reinterpret_cast<matrix_t>(*Craw);
    matrix_t referenceC = reinterpret_cast<matrix_t>(*referenceCraw);

    InitializeMatrices(A, B);
    Timer timer;

    [...]
}
```

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_3

Very Important:
Build infrastructure for testing correctness before implementing code transformations

```
[...]

int main(int argc, char *argv[])
{
    [...]

    // Correctness test
    std::cout << "Running candidate kernel for correctness test ... " << std::flush;
    timer.Start();
    MatMatMultiply(A, B, C);
    timer.Stop("Elapsed time : ");

    std::cout << "Running reference kernel for correctness test ... " << std::flush;
    timer.Start();
    MatMatMultiplyReference(A, B, referenceC);
    timer.Stop("Elapsed time : ");

    float discrepancy = MatrixMaxDifference(C, referenceC);
    std::cout << "Discrepancy between two methods : " << discrepancy << std::endl;

    for(int test = 1; test <= 20; test++)
    {
        std::cout << "Running kernel for performance run #" << std::setw(2) << test << " ... ";
        timer.Start();
        MatMatMultiply(A, B, C);
        timer.Stop("Elapsed time : ");
    }

    return 0;
}
```

GEMM routine (MatMatMultiply.h)

DenseAlgebra/GEMM_Test_0_3

```
#pragma once

#include "Parameters.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]);

void MatMatMultiplyReference(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]);
```

The “reference” implementation is using MKL BLAS (the “non-reference” is our hand-built version, with any transformations we enact)

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_3

```
#include "MatMatMultiply.h"
#include "mkl.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
#pragma omp parallel for
    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++) {
        C[i][j] = 0.;
        for (int k = 0; k < MATRIX_SIZE; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
}

void MatMatMultiplyReference(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
    cblas_sgemm(
        CblasRowMajor, CblasNoTrans, CblasNoTrans,
        MATRIX_SIZE, MATRIX_SIZE, MATRIX_SIZE,
        1.,
        &A[0][0], MATRIX_SIZE,
        &B[0][0], MATRIX_SIZE,
        0.,
        &C[0][0], MATRIX_SIZE
    );
}
```

Comparison code (Utilities.cpp)

DenseAlgebra/GEMM_Test_0_3

```
[...]

float MatrixMaxDifference(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE])
{
    float result = 0.;
    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++)
        result = std::max( result, std::abs( A[i][j] - B[i][j] ) );
    return result;
}
```

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_3

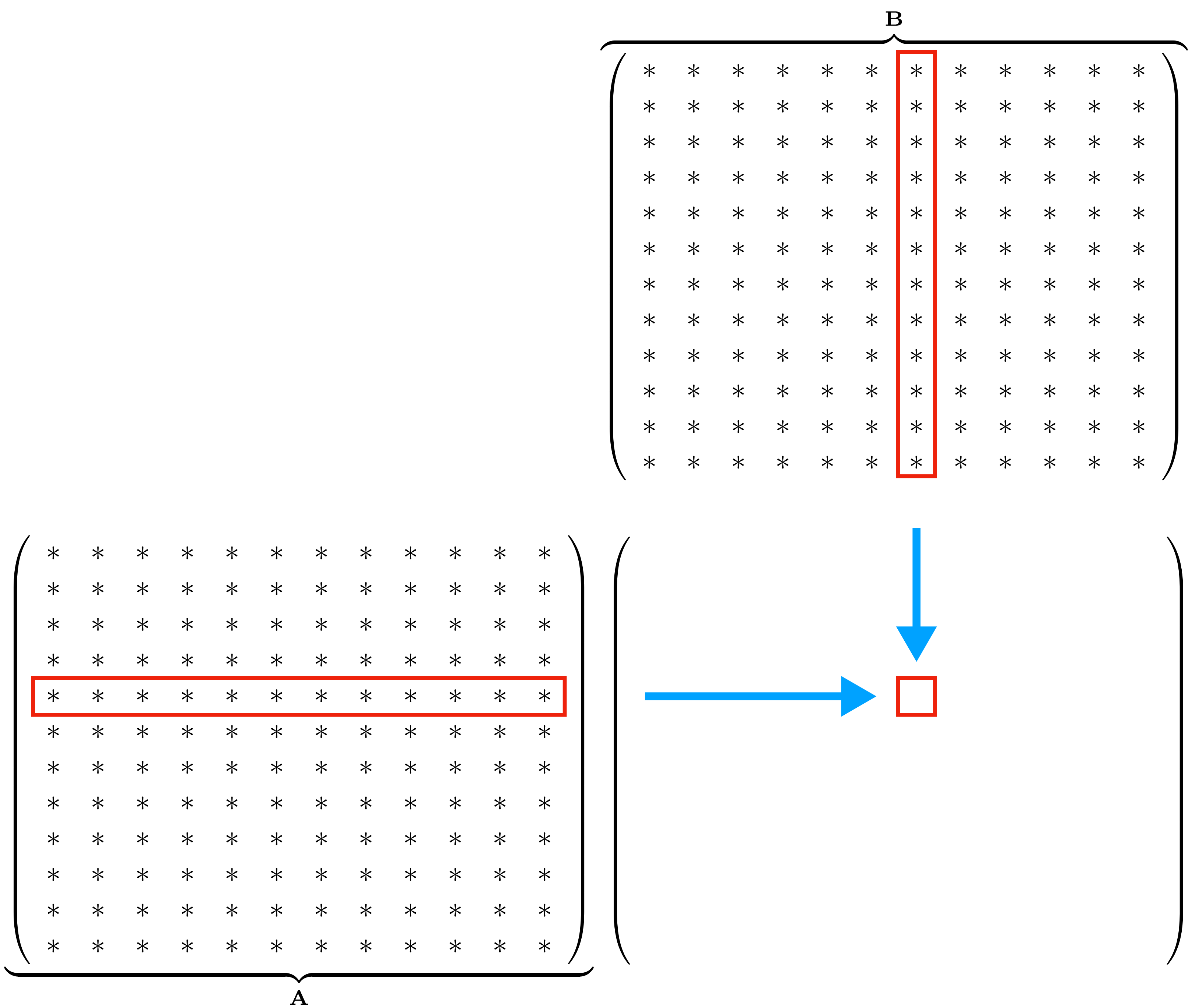
```
[...]

int main(int argc, char *argv[])
{
    [...]

    // Correctness test
    std::cout << "Running candidate kernel for correctness test ... " << std::flush;
    timer.Start();
    MatMatMultiply(A, B, C);
    timer.Stop("Elapsed time : ");

    std::cout << "Running reference kernel for correctness test ... " << std::flush;
    timer.Start();
    MatMatMultiplyReference(A, B, referenceC);
    timer.Stop("Elapsed time : ");

    float discrepancy = MatrixMaxDifference(C, referenceC);
    std::cout << "Discrepancy between two methods : " << discrepancy << std::endl;

    for(int test =
    {
        std::cout <
        timer.Start
        MatMulti
        timer.Stop(
    }

    return 0;
}
```

Execution:

```
Running candidate kernel for correctness test ... [Elapsed time : 273.398ms]
Running reference kernel for correctness test ... [Elapsed time : 29.5605ms]
Discrepancy between two methods : 8.01086e-05
Running kernel for performance run # 1 ... [Elapsed time : 221.153ms]
Running kernel for performance run # 2 ... [Elapsed time : 222.238ms]
Running kernel for performance run # 3 ... [Elapsed time : 221.794ms]
Running kernel for performance run # 4 ... [Elapsed time : 224.306ms]
[...]
```

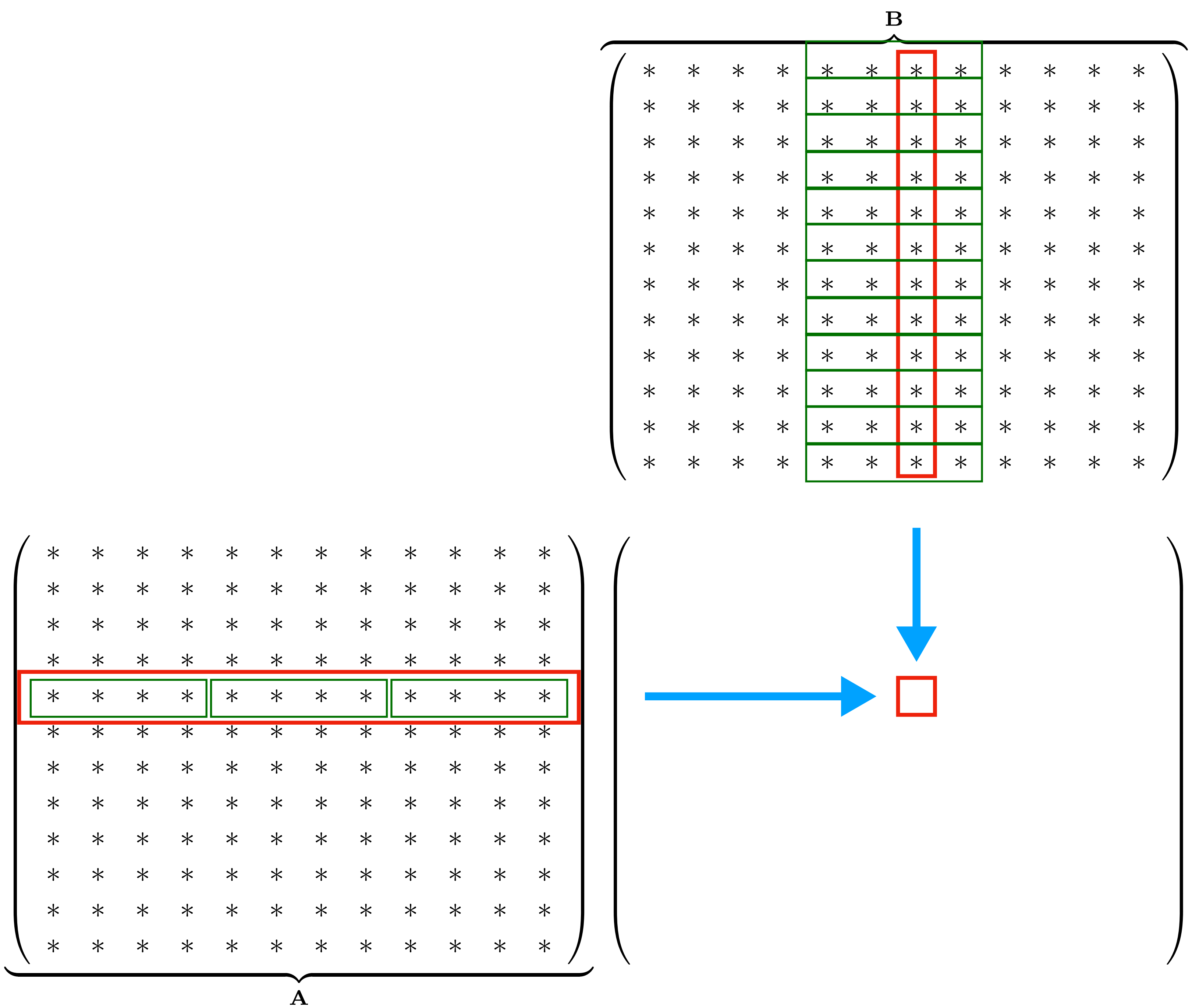
Causes of slowdown

$$
\begin{array}{l}
\texttt{for}\ i = 1 \ldots N \\
\quad \texttt{for}\ j = 1 \ldots N \\
\quad\quad C_{ij} \leftarrow 0 \\
\quad\quad \texttt{for}\ k = 1 \ldots N \\
\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} B_{kj}
\end{array}
$$



Causes of slowdown

$$
\begin{aligned}
&\texttt{for}\ \ i = 1 \ldots N\\
&\quad \texttt{for}\ \ j = 1 \ldots N\\
&\quad\quad C_{ij} \leftarrow 0\\
&\quad\quad \texttt{for}\ \ k = 1 \ldots N\\
&\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} B_{kj}
\end{aligned}
$$

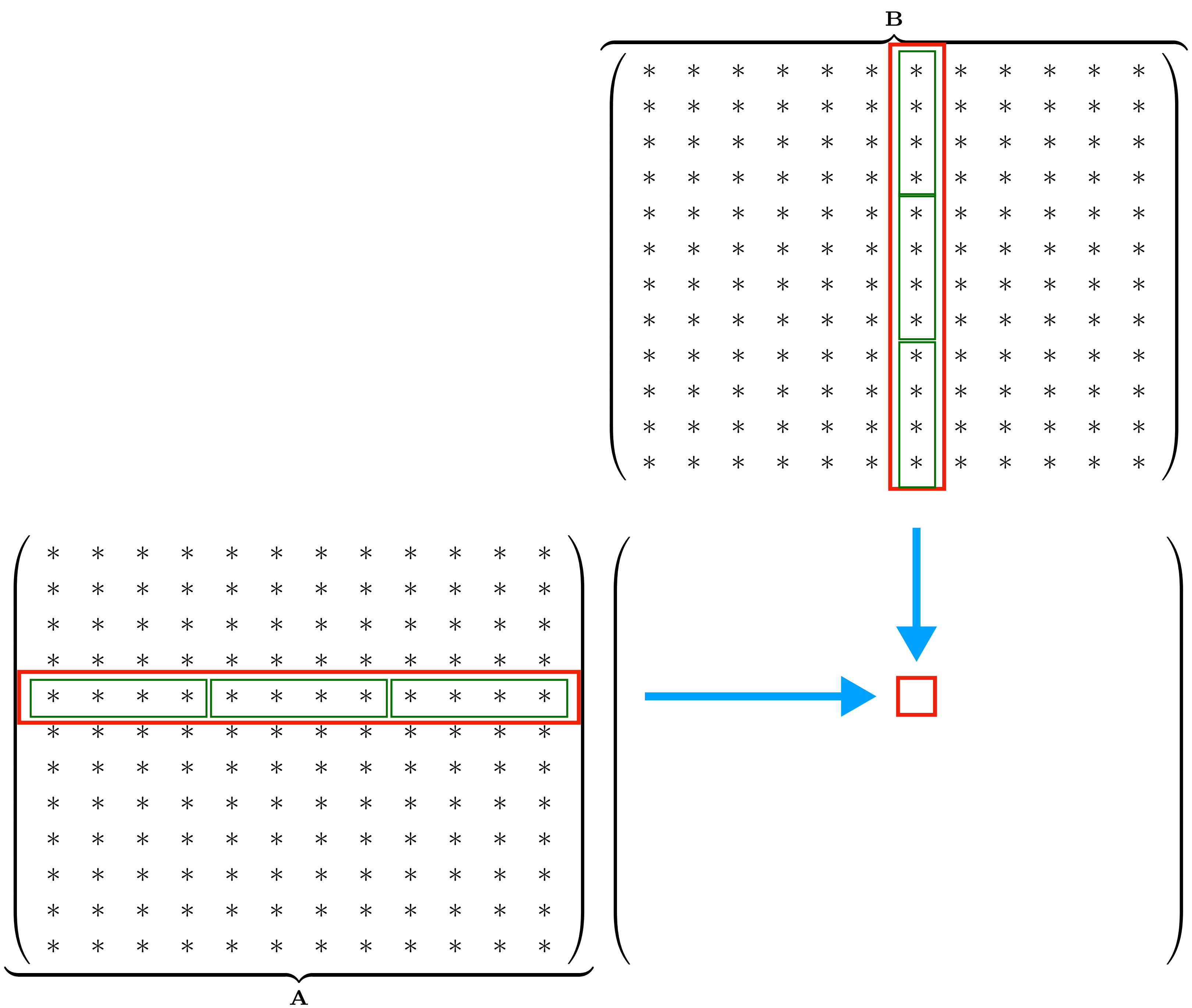


Shapes of cache lines (for row-major matrices)

*Memory bandwidth is being wasted while reading factor **B** …*

Causes of slowdown

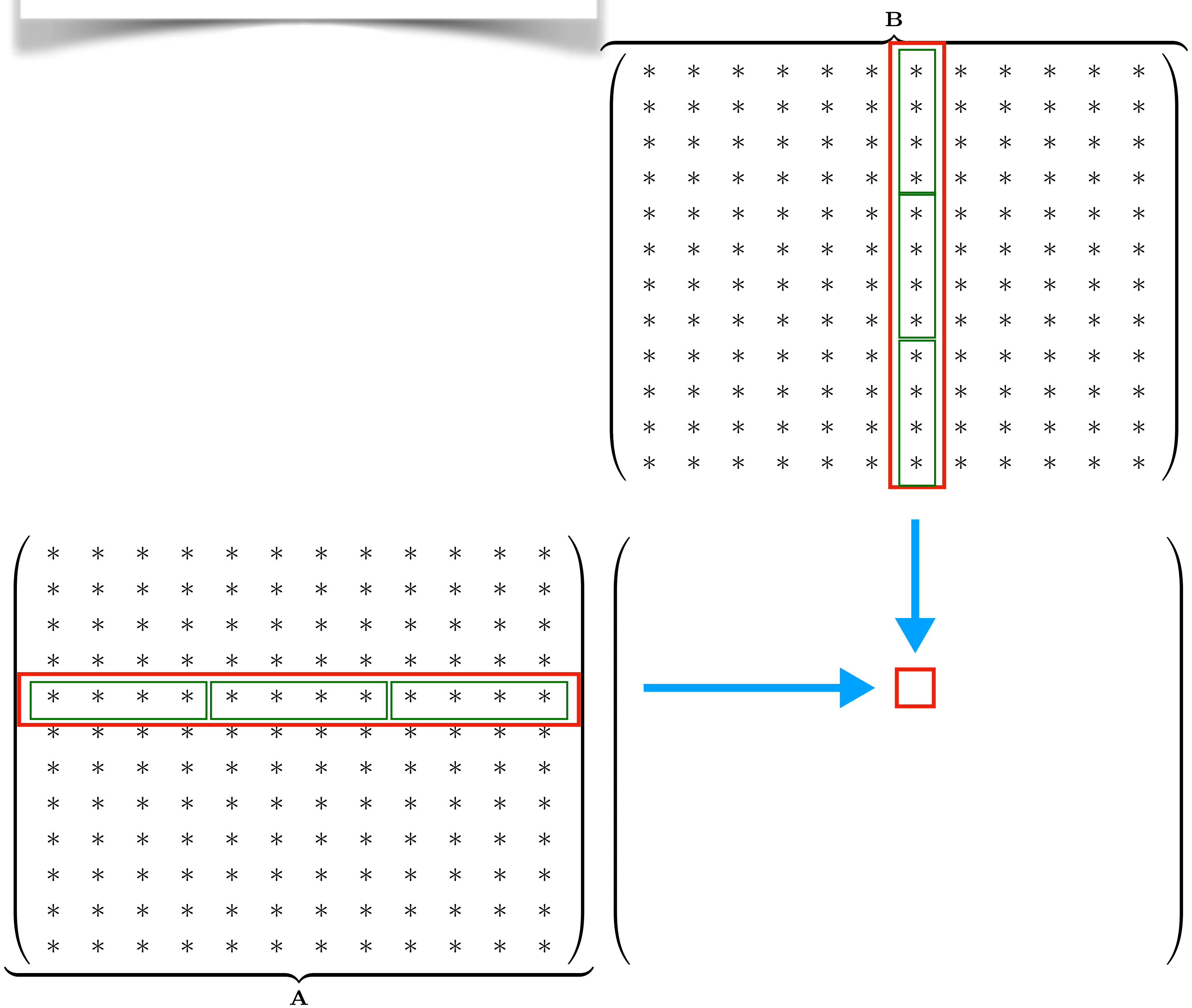
for $i = 1 \ldots N$
    for $j = 1 \ldots N$
        $C_{ij} \leftarrow 0$
        for $k = 1 \ldots N$
            $C_{ij} \leftarrow C_{ij} + A_{ik} B_{kj}$

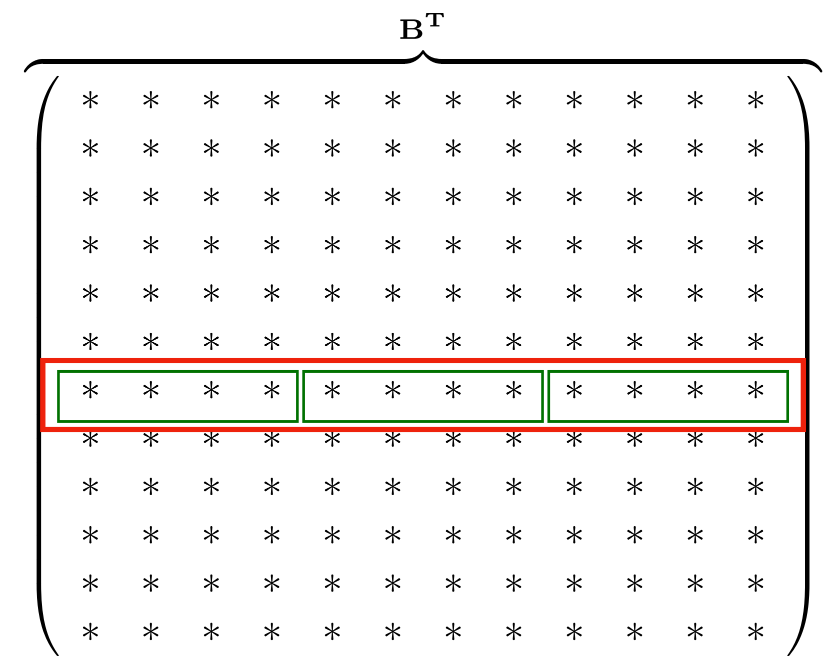


If, instead, ***B*** *was given as column-major …*

… cache lines are more effectively utilized

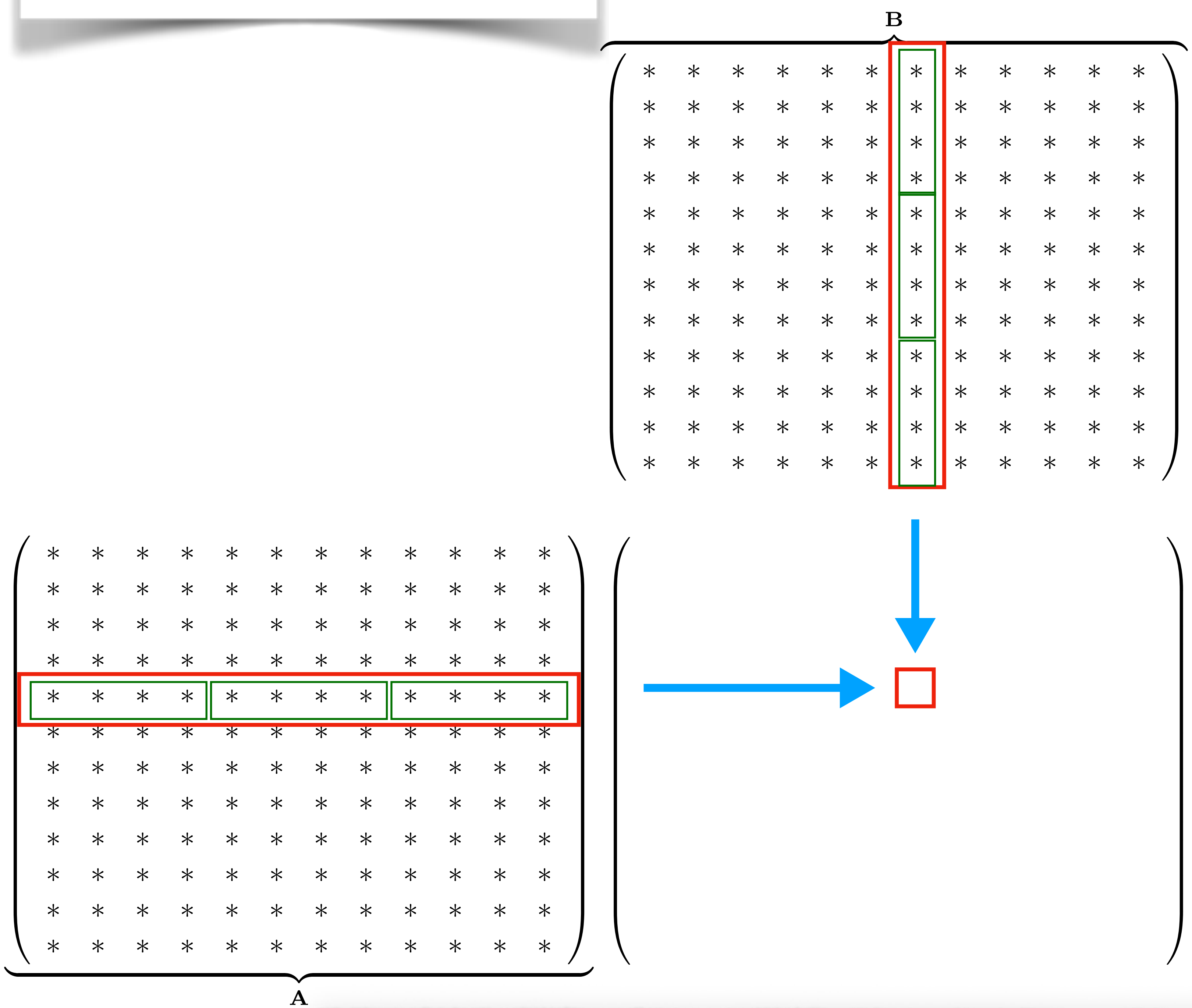
Use transpose?

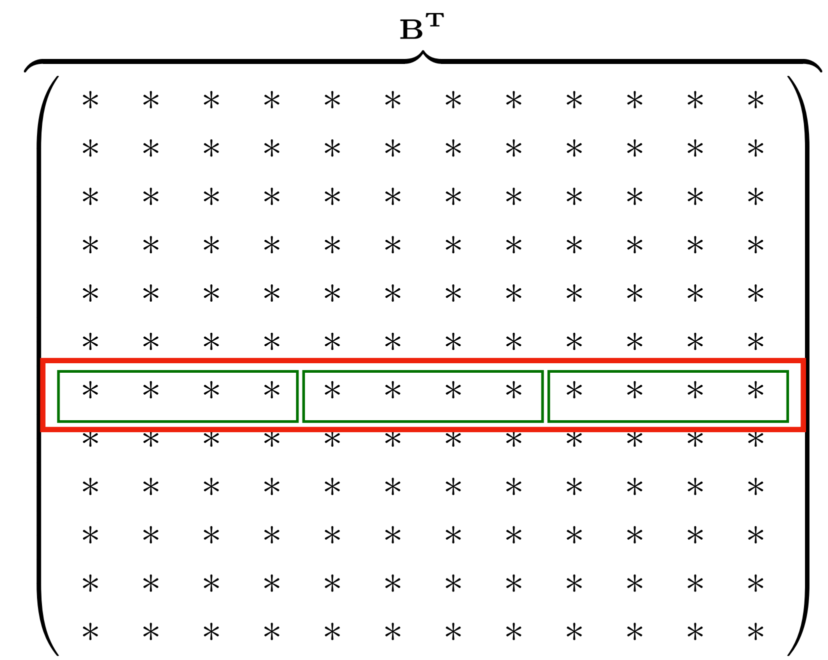




$$
\begin{aligned}
&\texttt{for}\ \ i = 1 \dots N\\
&\quad \texttt{for}\ \ j = 1 \dots N\\
&\qquad C_{ij} \leftarrow 0\\
&\qquad \texttt{for}\ \ k = 1 \dots N\\
&\qquad\quad C_{ij} \leftarrow C_{ij} + A_{ik} B_{kj}
\end{aligned}
$$

Use transpose?





$$
\begin{aligned}
&\texttt{for}\ \ i = 1 \ldots N\\
&\quad \texttt{for}\ \ j = 1 \ldots N\\
&\quad\quad C_{ij} \leftarrow 0\\
&\quad\quad \texttt{for}\ \ k = 1 \ldots N\\
&\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik}[B^T]_{jk}
\end{aligned}
$$

Multiplying with the transpose of B gives better cache utilization!

Two different interpretations:

(1) We multiply with B, stored in <u>column-major</u> format, or

(2) We multiply with B^T, stored in <u>row-major</u> format

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_4

```
[...]

int main(int argc, char *argv[])
{
    float *Araw=static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Braw=static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *BTraw=static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Craw=static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    [...]

    matrix_t A = reinterpret_cast<matrix_t>(*Araw);
    matrix_t B = reinterpret_cast<matrix_t>(*Braw);
    matrix_t BT = reinterpret_cast<matrix_t>(*BTraw);
    [...]

    InitializeMatrices(A, B);
    Timer timer;

    // Pre-transposing B
    std::cout << "Transposing second matrix factor ... " << std::flush;
    timer.Start();
    MatTranspose(B, BT);
    timer.Stop("Elapsed time : ");

    [...]
}
```

Build the matrix ***B^T*** *in advance …*

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_4

```
[...]

int main(int argc, char *argv[])
{
    [...]
    // Correctness test
    std::cout << "Running candidate kernel for correctness test ... " << std::flush;
    timer.Start();
    MatMatTransposeMultiply(A, BT, C);
    timer.Stop("Elapsed time : ");

    std::cout << "Running reference kernel for correctness test ... " << std::flush;
    timer.Start();
    MatMatMultiplyReference(A, B, referenceC);
    timer.Stop("Elapsed time : ");

    float discrepancy = MatrixMaxDifference(C, referenceC);
    std::cout << "Discrepancy between two methods : " << discrepancy << std::endl;

    for(int test = 1; test <= 20; test++)
    {
        std::cout << "Running kernel for performance run #" << std::setw(2) << test << " ... ";
        timer.Start();
        MatMatTransposeMultiply(A, BT, C);
        timer.Stop("Elapsed time : ");
    }

    return 0;
}
```

... and multiply with the transpose instead

Multiply w/Transpose (MatMatMultiply.cpp)

```
[...]
void MatTranspose(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    float (&AT)[MATRIX_SIZE][MATRIX_SIZE])
{
    mkl_somatcopy(
        'R',         // Matrix A is in row-major format
        'T',         // We are performing a transposition operation
        MATRIX_SIZE, // Dimensions of matrix -- rows ...
        MATRIX_SIZE, // ... and columns
        1.,          // No scaling
        &A[0][0],    // Input matrix
        MATRIX_SIZE, // Leading dimension (here, just the matrix dimension)
        &AT[0][0],   // Output matrix
        MATRIX_SIZE  // Leading dimension
    );
}

void MatMatTransposeMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
#pragma omp parallel for
    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++) {
        C[i][j] = 0.;
        for (int k = 0; k < MATRIX_SIZE; k++)
            C[i][j] += A[i][k] * B[j][k];
    }
}
[.. .]
```

Multiply w/Transpose (MatMatMultiply.cpp)

```
[...]
void MatTranspose(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    float (&AT)[MATRIX_SIZE][MATRIX_SIZE])
{
    mkl_somatcopy(
        'R',         // Matrix A is in row-major format
        'T',         // We are performing a transposition operation
        MATRIX_SIZE, // Dimensions of matrix -- rows ...
        MATRIX_SIZE, // ... and columns
        1.,          // No scaling
        &A[0][0],    // Input matrix
        MATRIX_SIZE, // Leading dimension (here, just the matrix dimension)
        &AT[0][0],   // Output matrix
        MATRIX_SIZE  // Leading dimension
    );
}

void MatMatTransposeMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
#pragma omp parallel for
    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++) {
        C[i][j] = 0.;
        for (int k = 0; k < MATRIX_SIZE; k++)
            C[i][j] += A[i][k] * B[j][k];
    }
}
[.. .]
```

Multiply w/Transpose (MatMatMultiply.cpp)

```
[...]
void MatTranspose(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    float (&AT)[MATRIX_SIZE][MATRIX_SIZE])
{
    mkl_somatcopy(
        'R',          // Matrix A is in row-major format
        'T',          // We are performing a transposition operation
        MATRIX_SIZE, // Dimensions of matrix -- rows ...
        MATRIX_SIZE, // ... and columns
        1.,          // No scaling
        &A[0][0],    // Input matrix
        MATRIX_SIZE, // Leading dimension (here, just the matrix dimension)
        &AT[0][0],   // Output matrix
        MATRIX_SIZE  // Leading dimension
    );
}

void MatMatTransposeMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
#pragma omp parallel for
    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++) {
        C[i][j] = 0.;
        for (int k = 0; k < MATRIX_SIZE; k++)
            C[i][j] += A[i][k] * B[j][k];
    }
}
[.. .]
```

$$
\begin{aligned}
&\texttt{for}\quad i = 1 \ldots N\\
&\quad \texttt{for}\quad j = 1 \ldots N\\
&\quad\quad C_{ij} \leftarrow 0\\
&\quad\quad \texttt{for}\quad k = 1 \ldots N\\
&\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik}[B^T]_{jk}
\end{aligned}
$$

DenseAlgebra/GEMM_Test_0_4

Multiply w/Transpose (MatMatMultiply.cpp)

```
[...]
void MatTranspose(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    float (&AT)[MATRIX_SIZE][MATRIX_SIZE])
{
    mkl_somatcopy(
        'R',         // Matrix A is in row-major format
        'T',         // We are performing a transposition operation
        MATRIX_SIZE, // Dimensions of matrix -- rows ...
        MATRIX_SIZE, // ... and columns
        1.,          // No scaling
        &A[0][0],    // Input matrix
        MATRIX_SIZE, // Leading dimension (here, just the matrix dimension)
        &AT[0][0],   // Output matrix
        MATRIX_SIZE  // Leading dimension
    );
}

void MatMatTranspo
    const float (&
{
#pragma omp parall
    for (int i = 0
    for (int j = 0
        C[i][j] =
        for (int k
            C[i][j
    }
}
[.. .]
```

At matrix size = 1024

Execution:

```
Transposing second matrix factor ... [Elapsed time : 16.4232ms]
Running candidate kernel for correctness test ... [Elapsed time : 45.558ms]
Running reference kernel for correctness test ... [Elapsed time : 1.96817ms]
Discrepancy between two methods : 6.86646e-05
Running kernel for performance run # 1 ... [Elapsed time : 34.7349ms]
Running kernel for performance run # 2 ... [Elapsed time : 35.4725ms]
Running kernel for performance run # 3 ... [Elapsed time : 37.0109ms]
Running kernel for performance run # 4 ... [Elapsed time : 36.4638ms]
Running kernel for performance run # 5 ... [Elapsed time : 36.53ms]
Running kernel for performance run # 6 ... [Elapsed time : 36.6595ms]
Running kernel for performance run # 7 ... [Elapsed time : 36.5089ms]
[...]
```

DenseAlgebra/GEMM_Test_0_4

Multiply w/Transpose (MatMatMultiply.cpp)

```
[...]
void MatTranspose(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    float (&AT)[MATRIX_SIZE][MATRIX_SIZE])
{
    mkl_somatcopy(
        'R',          // Matrix A is in row-major format
        'T',          // We are performing a transposition operation
        MATRIX_SIZE, // Dimensions of matrix -- rows ...
        MATRIX_SIZE, // ... and columns
        1.,           // No scaling
        &A[0][0],     // Input matrix
        MATRIX_SIZE, // Leading dimension (here, just the matrix dimension)
        &AT[0][0],    // Output matrix
        MATRIX_SIZE  // Leading dimension
    );
}

void MatMatTranspo
    const float (&
{
#pragma omp parall
    for (int i = 0
    for (int j = 0
        C[i][j] =
        for (int k
            C[i][j
    }
}
[.. .]
```

At matrix size = 2048

Execution:

```
Transposing second matrix factor ... [Elapsed time : 28.3228ms]
Running candidate kernel for correctness test ... [Elapsed time : 413.998ms]
Running reference kernel for correctness test ... [Elapsed time : 16.1733ms]
Discrepancy between two methods : 0.000152588
Running kernel for performance run # 1 ... [Elapsed time : 391.771ms]
Running kernel for performance run # 2 ... [Elapsed time : 394.115ms]
Running kernel for performance run # 3 ... [Elapsed time : 395.299ms]
Running kernel for performance run # 4 ... [Elapsed time : 388.921ms]
Running kernel for performance run # 5 ... [Elapsed time : 396.476ms]
Running kernel for performance run # 6 ... [Elapsed time : 403.584ms]
Running kernel for performance run # 7 ... [Elapsed time : 396.318ms]
[...]
```