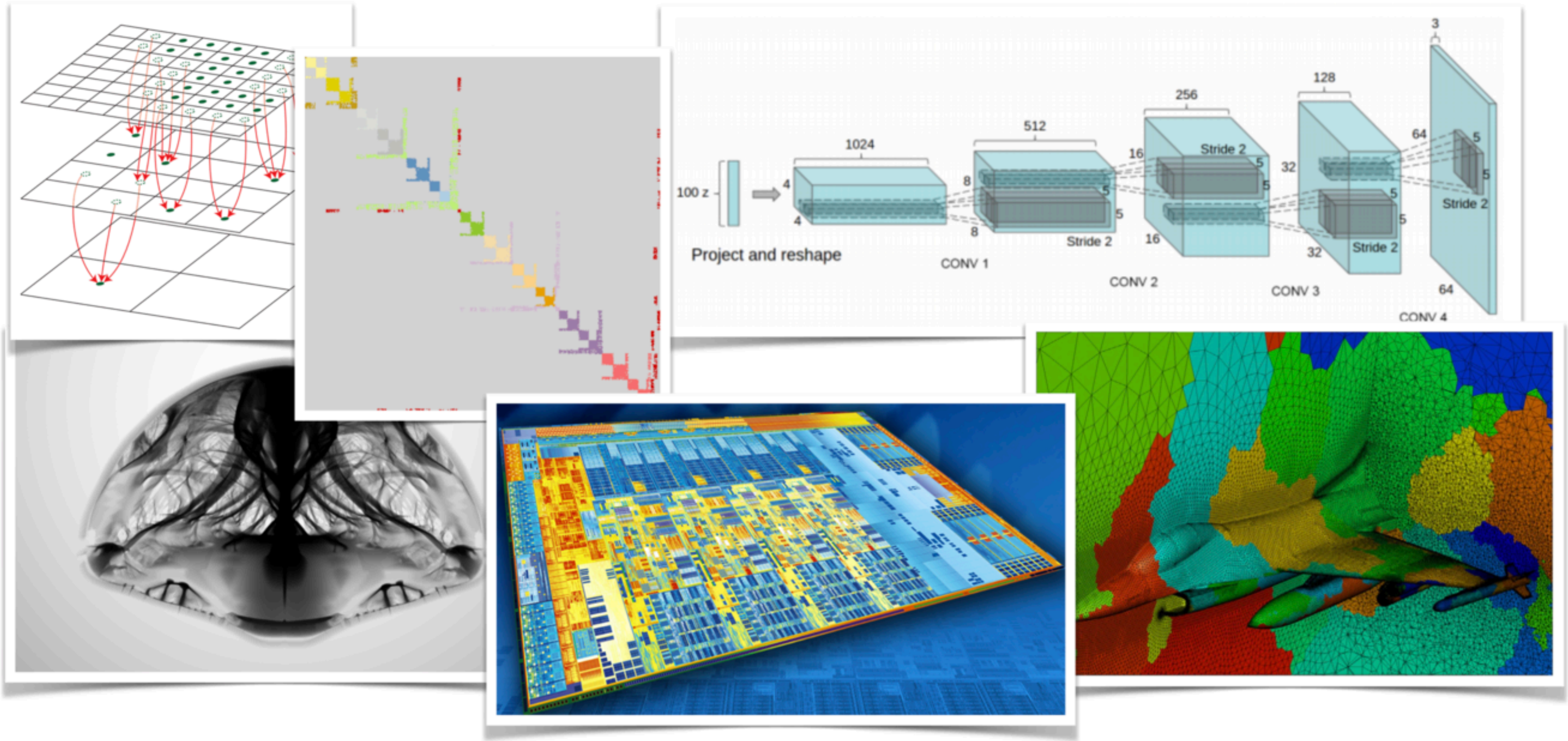


Lecture 11: Continued optimizations on General Matrix-Matrix multiplication (GEMM)

Tuesday February 28th 2023

Logistics

- HW #2 due Friday
- You will get a practice midterm by end of week (not due, just for your own study)
- We'll go over the midterm next Tuesday

Today’s lecture

- Flash review of GEMM theory from last lecture
- Additional optimizations: Blocking, transposition, cache line optimizations, etc.
- Some notes on correctness checking (essential for developing nontrivial optimizations)

RECAP

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

RECAP

*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:

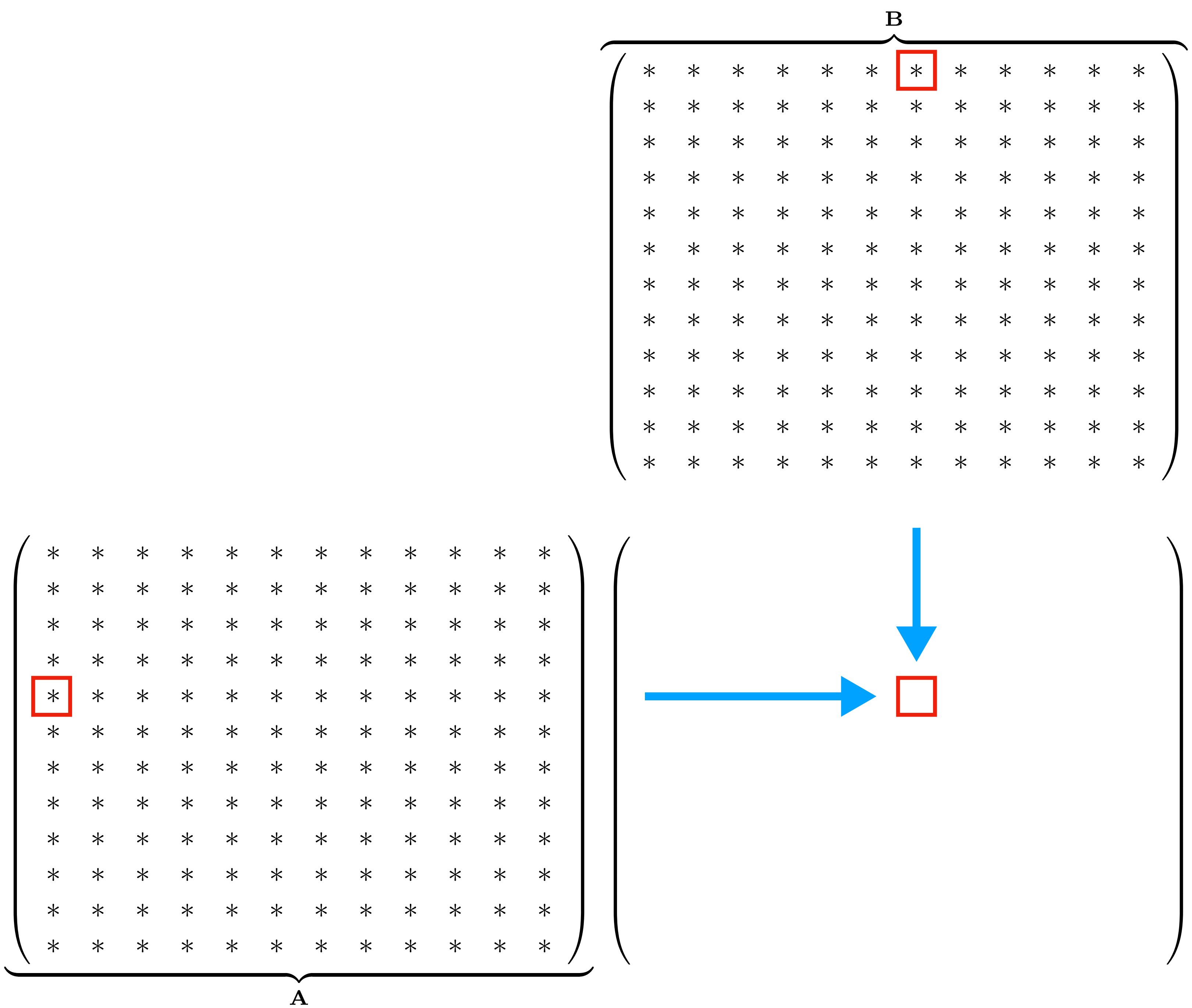
N^2 data and N^3 computation

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

Theory of GEMM operation

RECAP

A visual illustration …

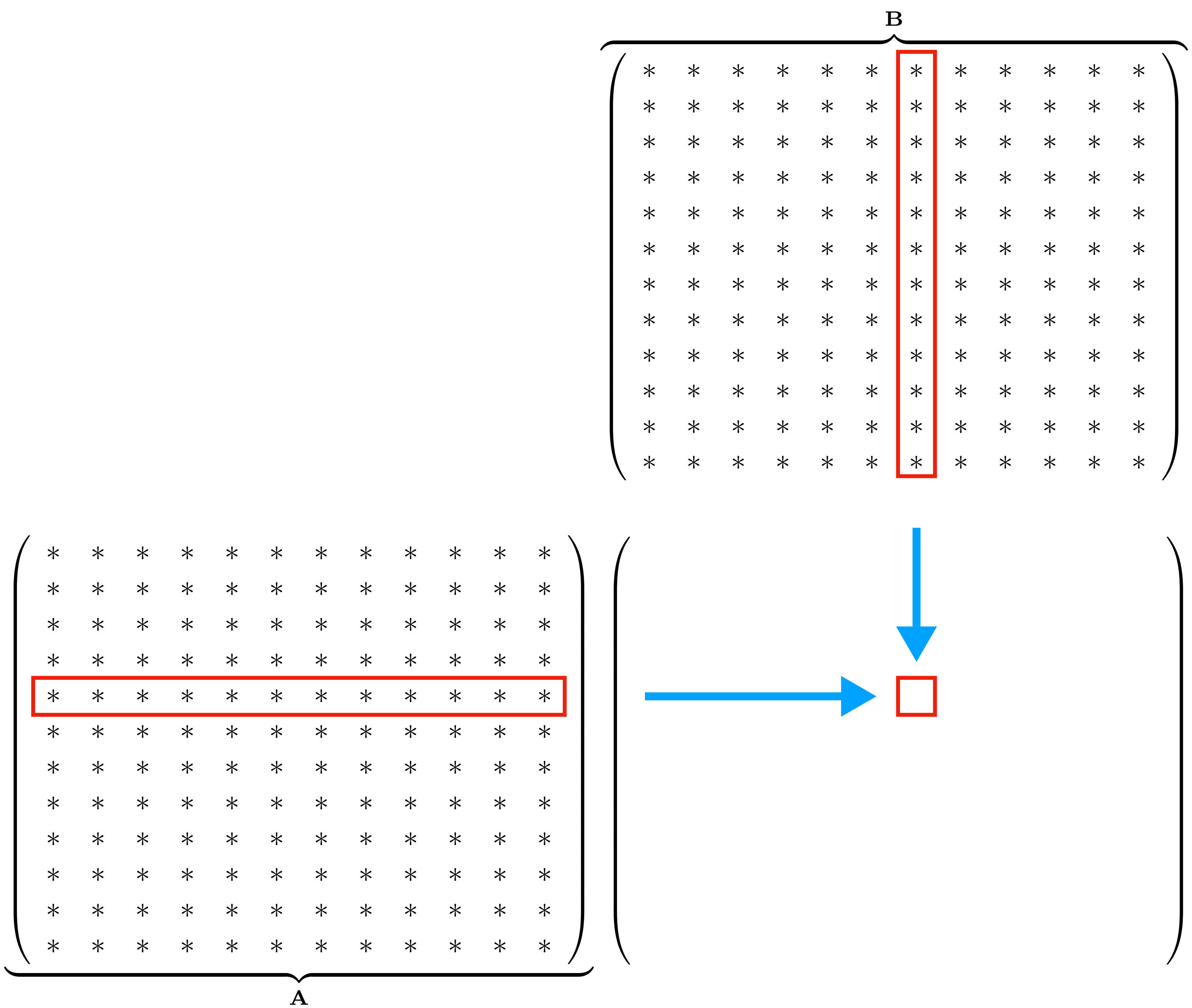


*Multiply respective entries of **A** & **B**,*
*accumulate on highlighted entry of **C=A*B***

Theory of GEMM operation

RECAP

A visual illustration …



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

GEMM routine (MatMatMultiply.cpp)

RECAP

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]
```

GEMM routine (MatMatMultiply.cpp)

RECAP

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)

RECAP

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]
```

GEMM routine (MatMatMultiply.cpp)

RECAP

DenseAlgebra/GEMM_Test_0_2

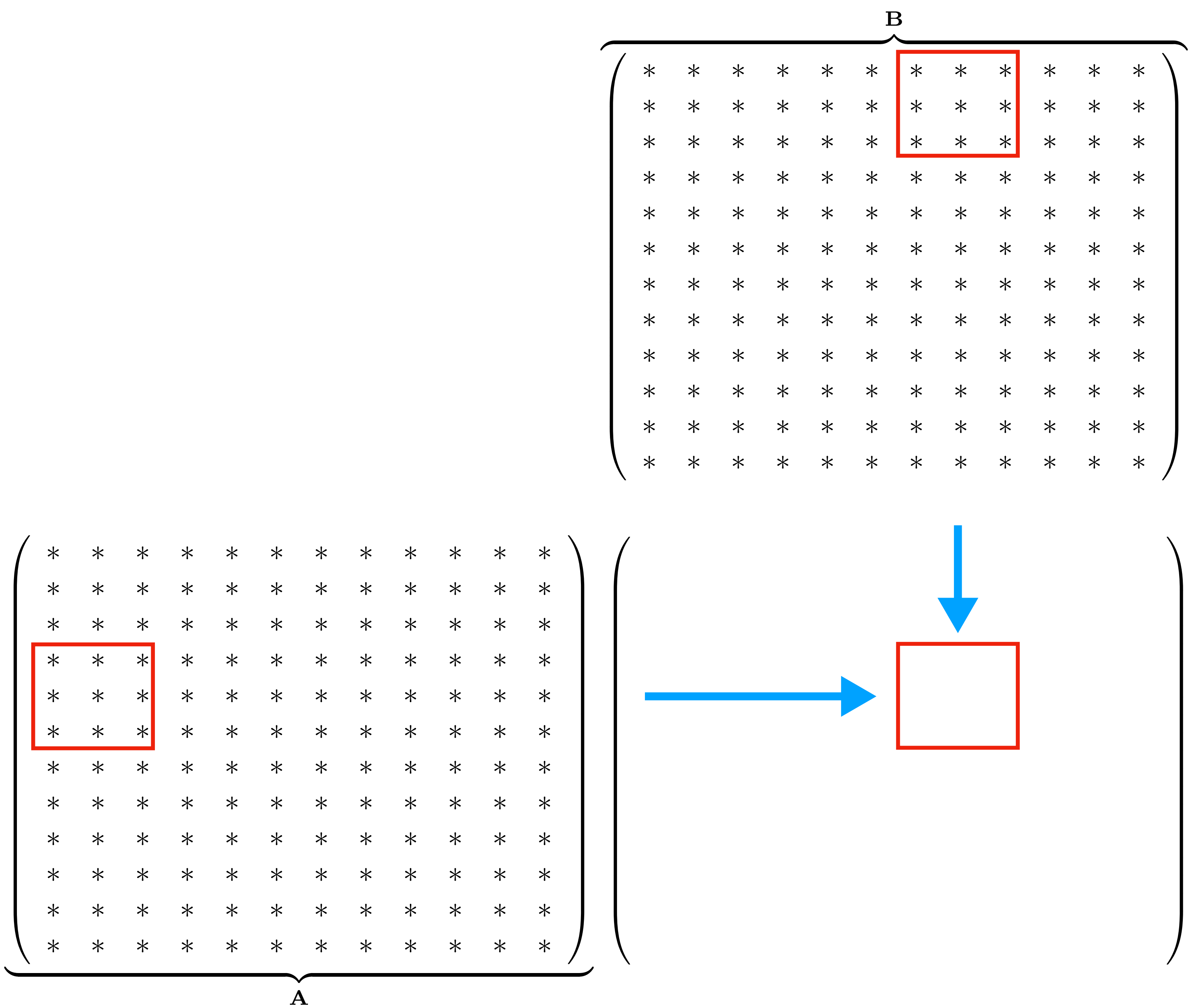
```
#include "MatMatMultiply.h"

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

RECAP

A visual illustration …

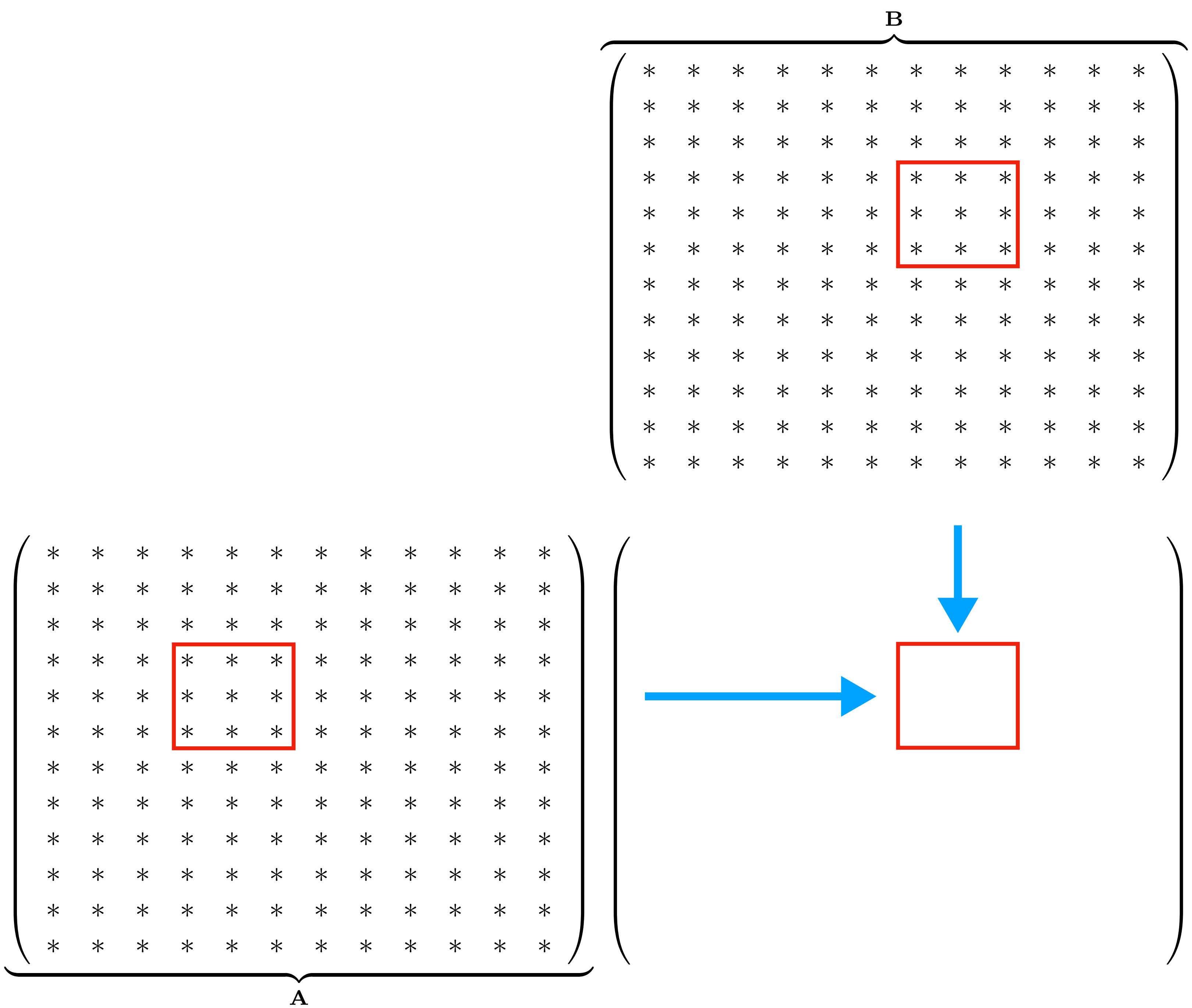


Multiply respective ***sub-matrices (blocks)*** *of* ***A*** *&* ***B****,*
accumulate on highlighted ***block*** *of* ***C=A*B***

Theory of GEMM operation

RECAP

A visual illustration …

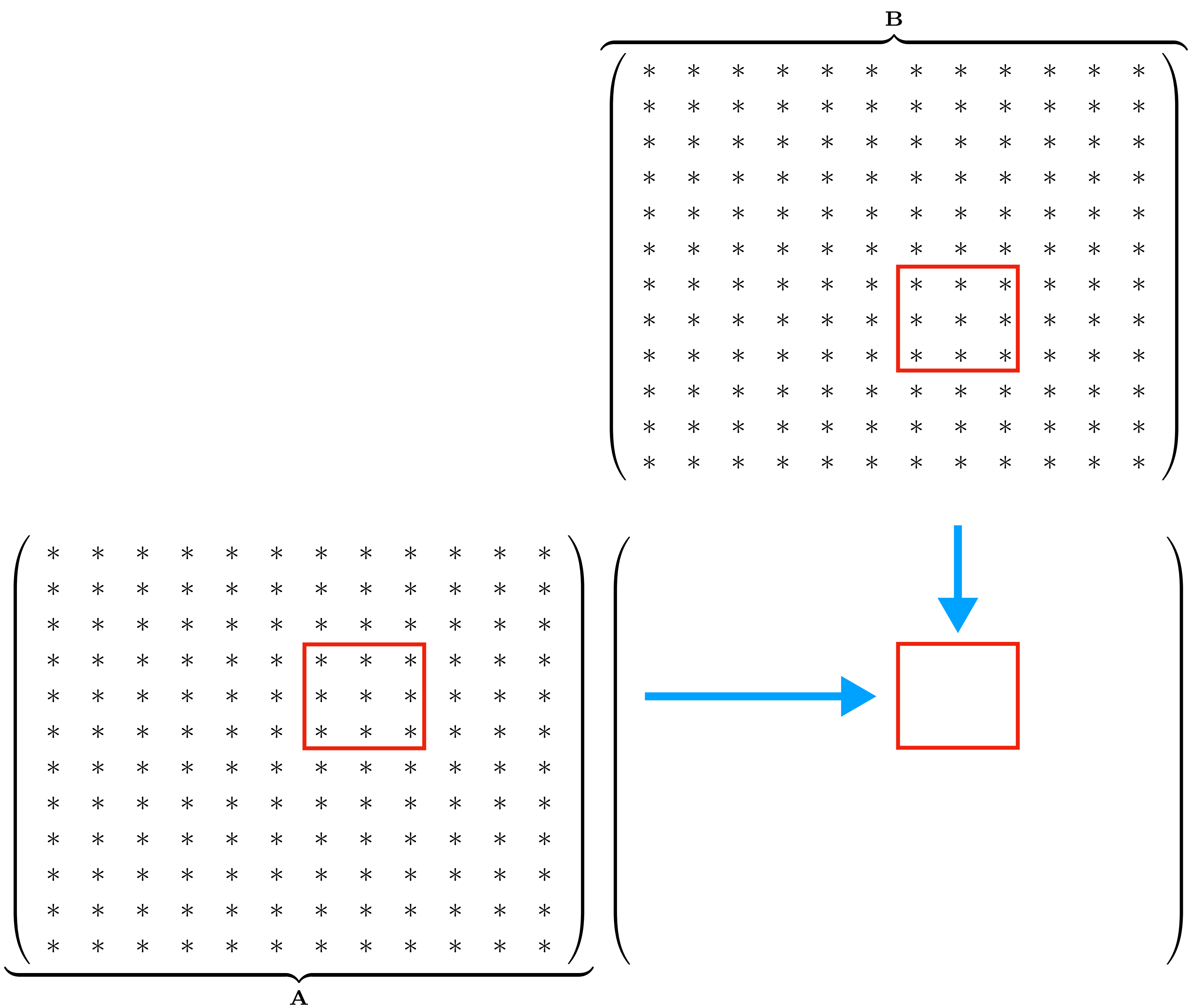


*Multiply respective **sub-matrices (blocks)** of **A** & **B**, accumulate on highlighted **block** of **C=A*B***

Theory of GEMM operation

RECAP

A visual illustration …

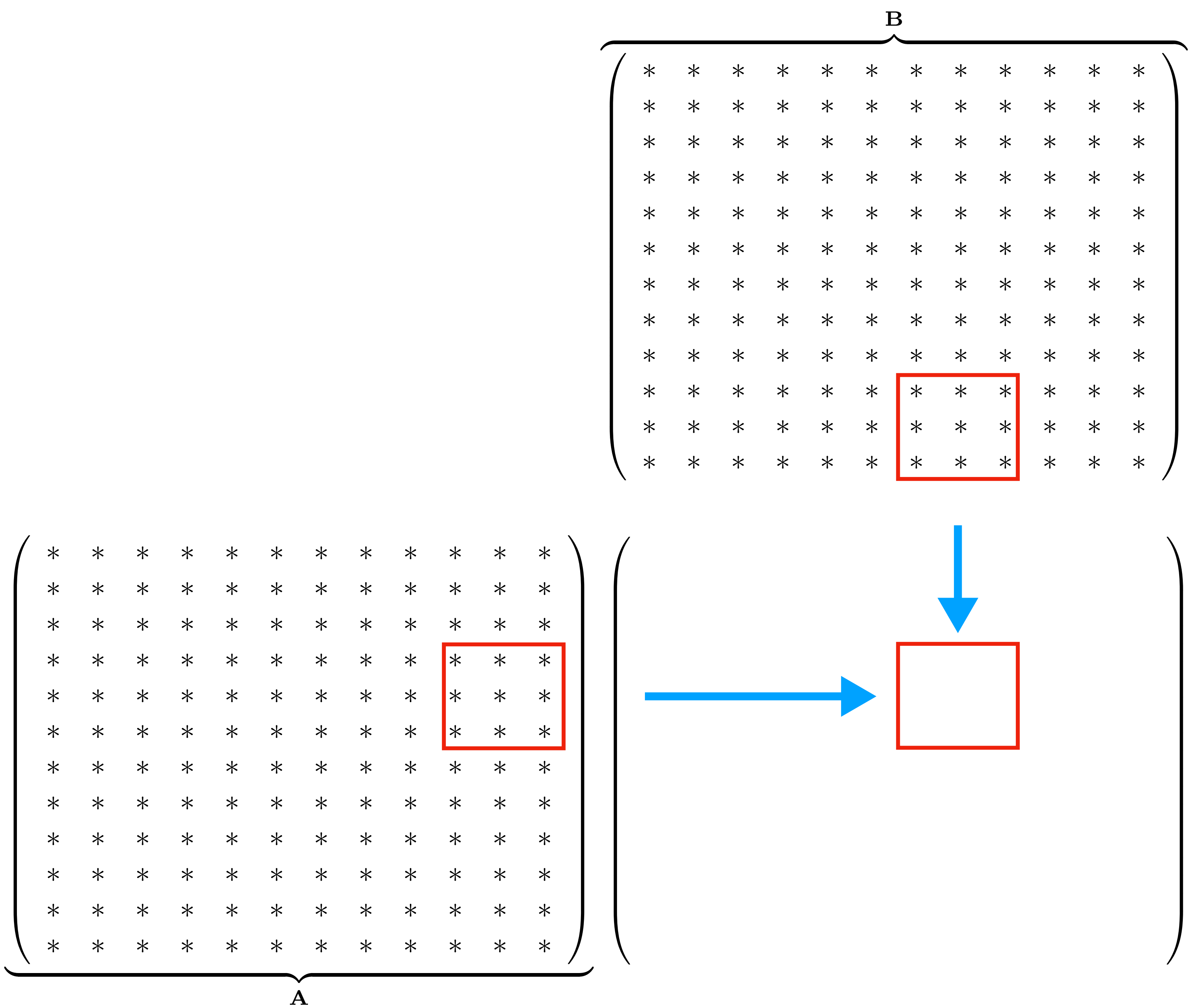


*Multiply respective **sub-matrices (blocks)** of **A** & **B**, accumulate on highlighted **block** of **C=A*B***

Theory of GEMM operation

RECAP

A visual illustration …



Tiling, blocking promotes cache reuse
with a limited working set size

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

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

“Blocked” indexing of matrix entries

RECAP

j=8

i=4

$$\underbrace{\left(\begin{array}{ccc|ccc|ccc|ccc}
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & * \\
\hline
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & \boxed{*} & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & * \\
\hline
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & * \\
\hline
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * & * & *
\end{array}\right)}_{\mathbf{A}}$$

bi=1 (block index)
ii=1 (subindex in block)

bj=2 (block index)
jj=2 (subindex in block)

RECAP

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]

RECAP

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];

        }
}
```

Use 6-way (instead of 3-way) for-loop

RECAP

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

Correctness checking logic

```
#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

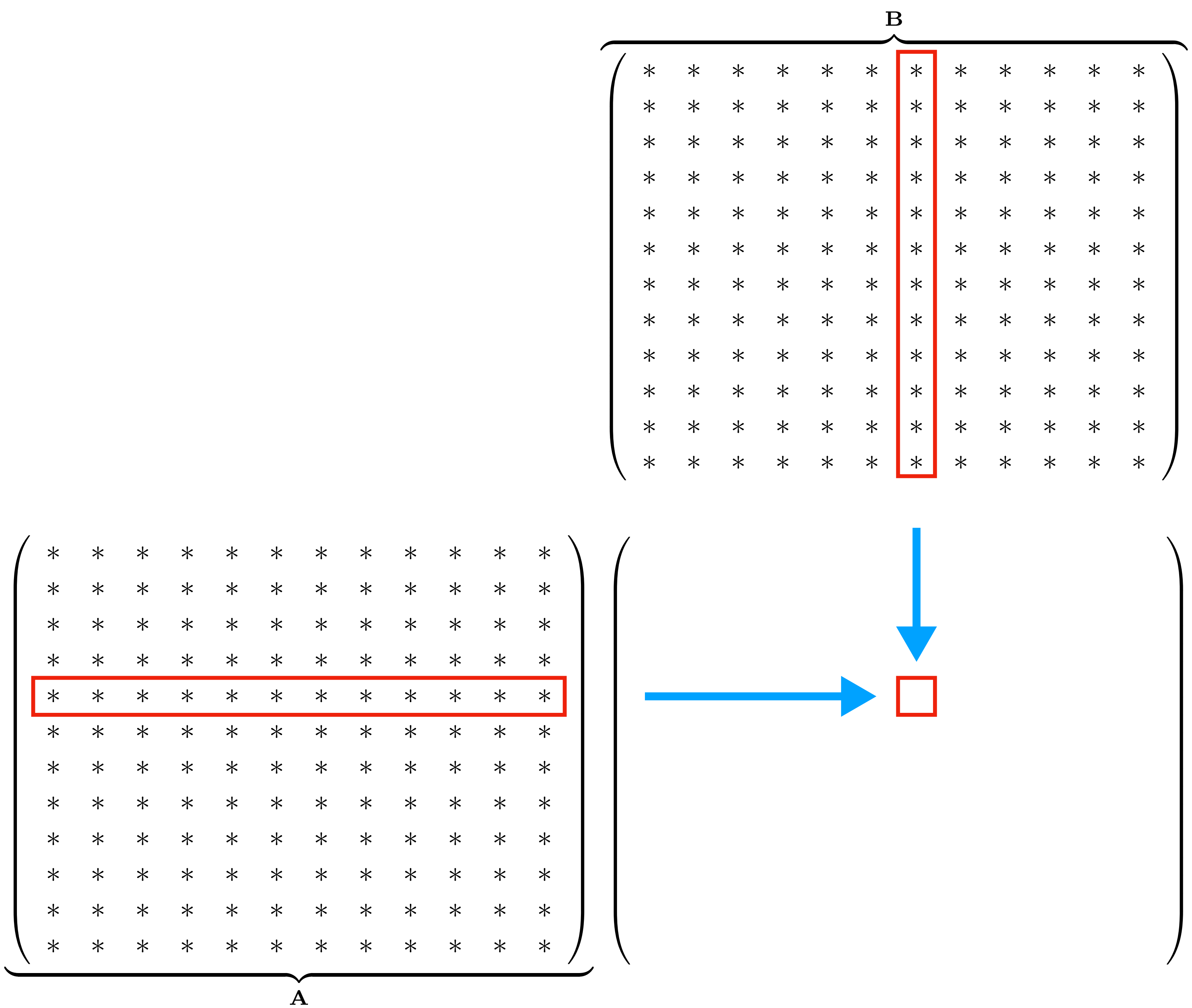
```
[...]

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
        MatMatMulti
        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]
[...]
```

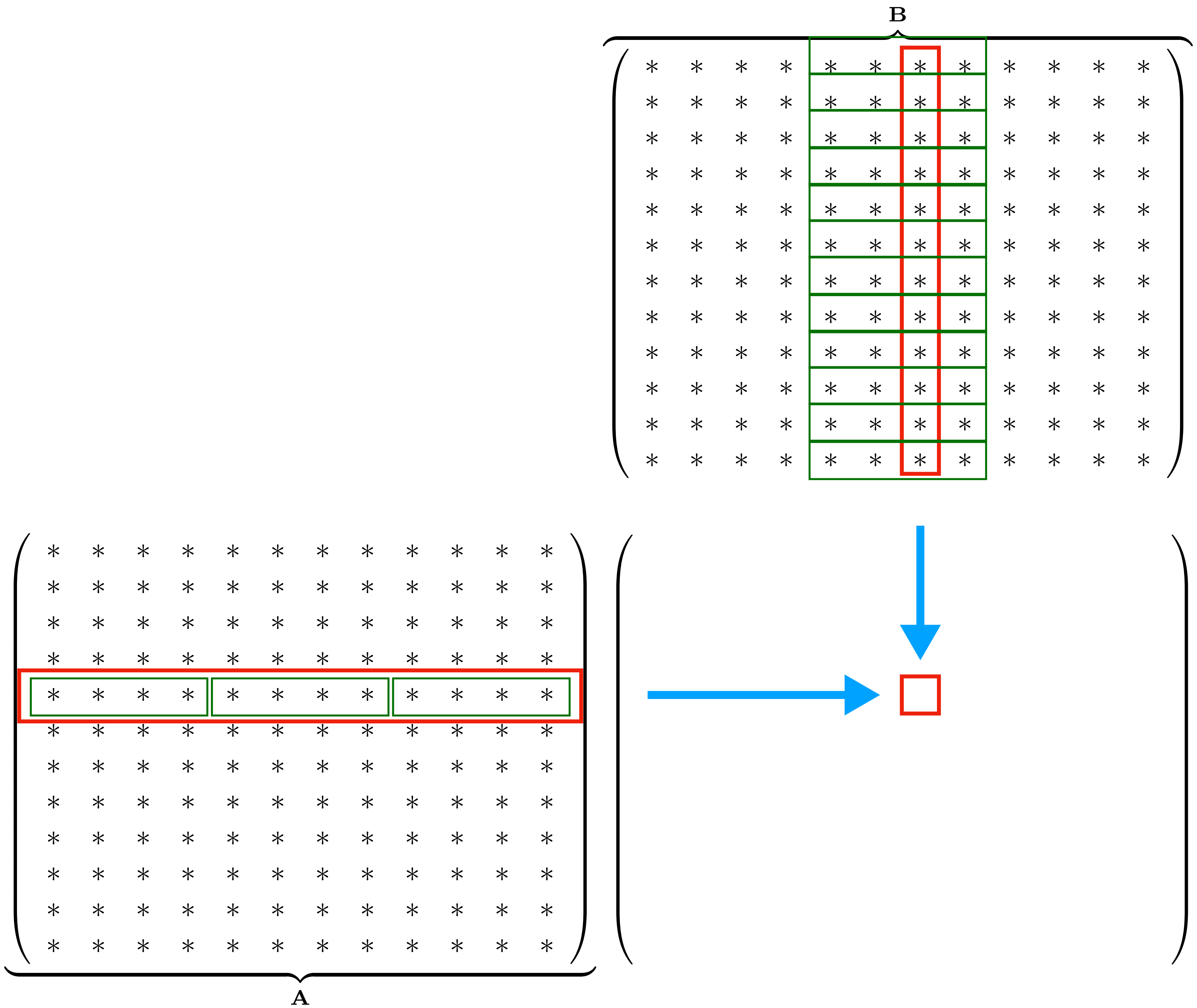
Causes of slowdown

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



Causes of slowdown

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

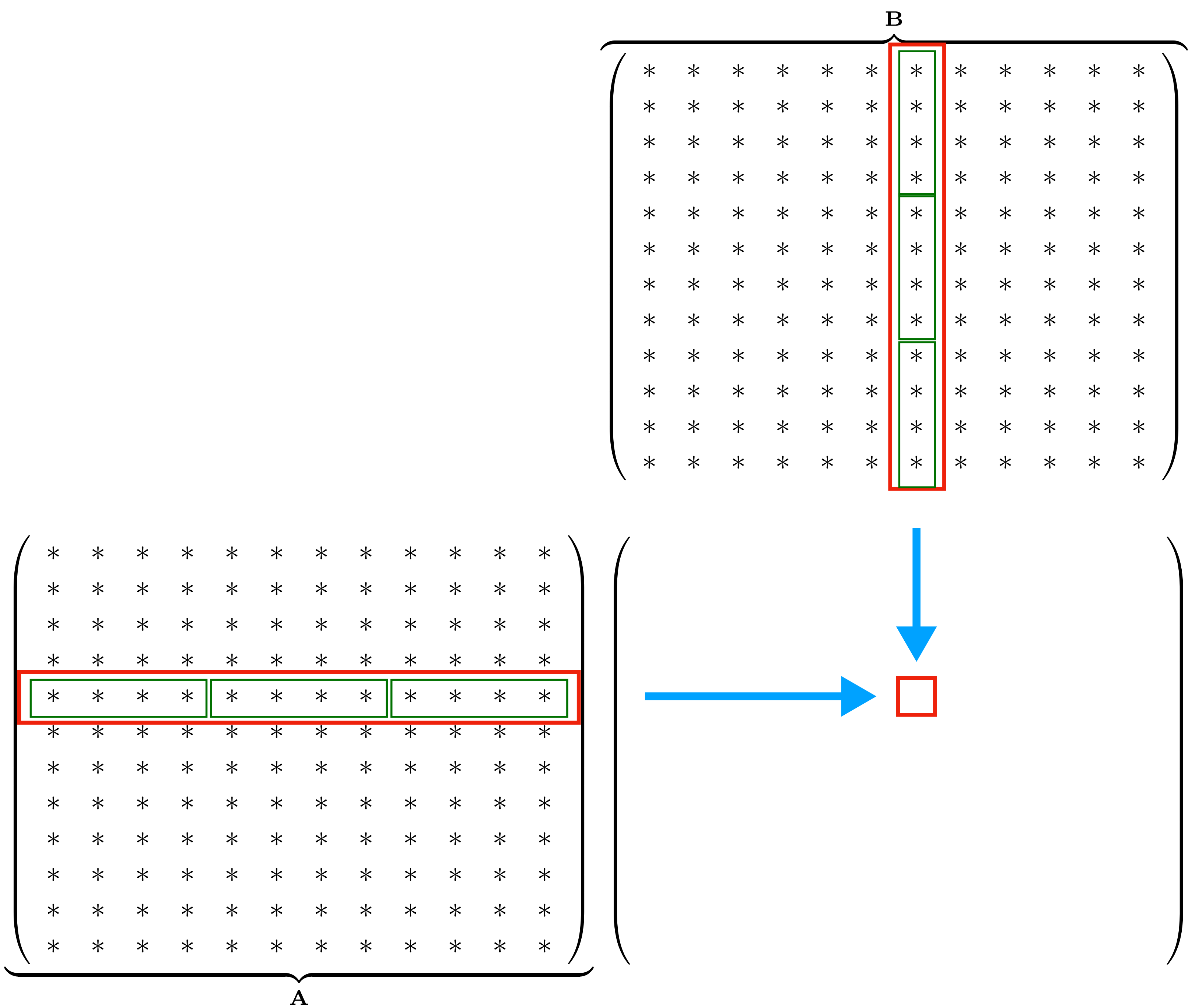


Shapes of cache lines (for row-major matrices)

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

Causes of slowdown

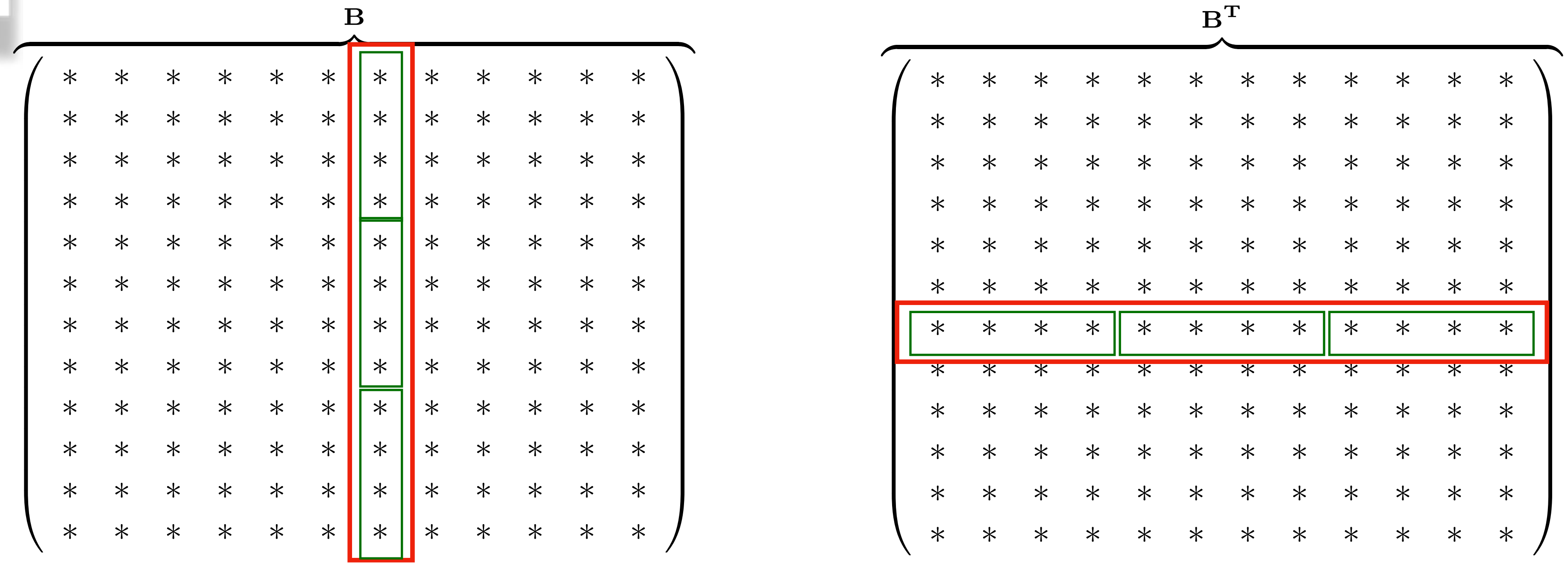
for $i = 1 \ldots N$
  for $j = 1 \ldots N$
    $C_{ij} \leftarrow 0$
    for $k = 1 \ldots N$
      $C_{ij} \leftarrow C_{ij} + A_{ik} B_{kj}$

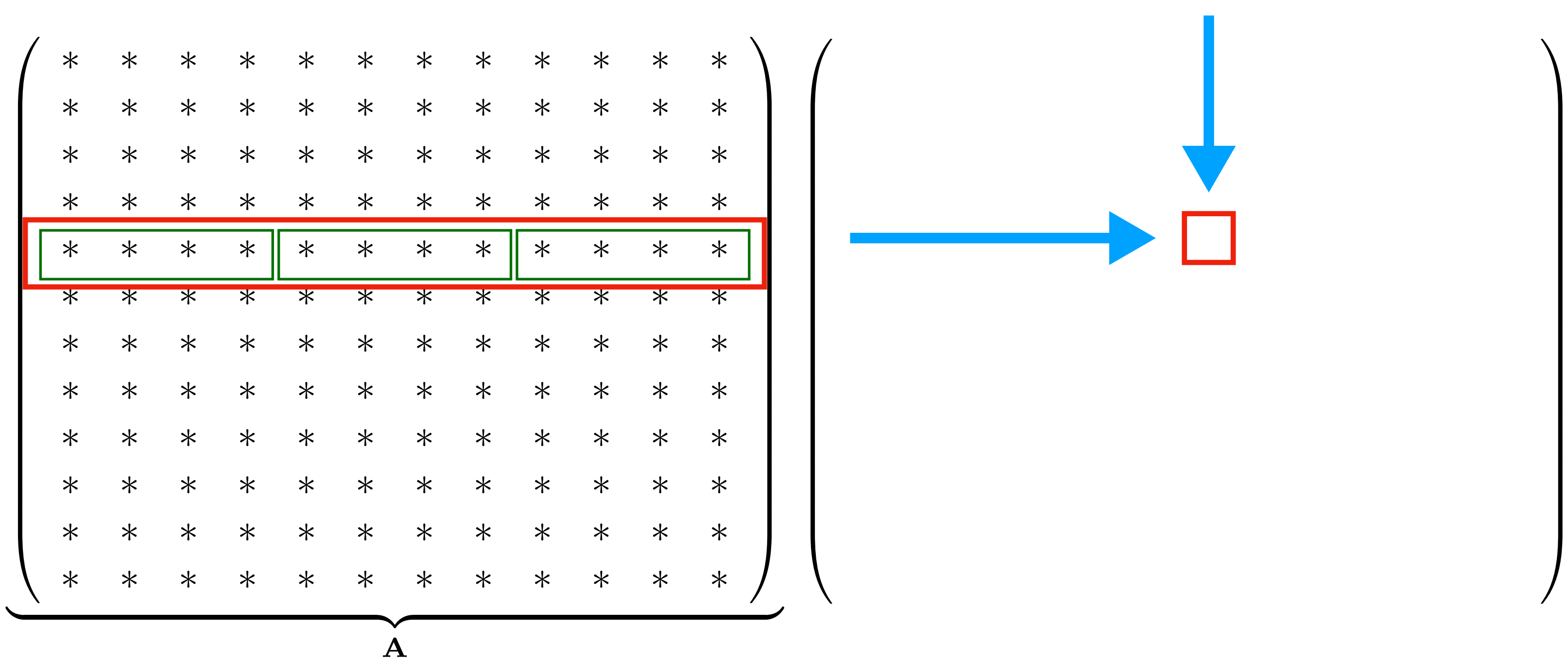


*If, instead, **B** was given as column-major …*

… cache lines are more effectively utilized

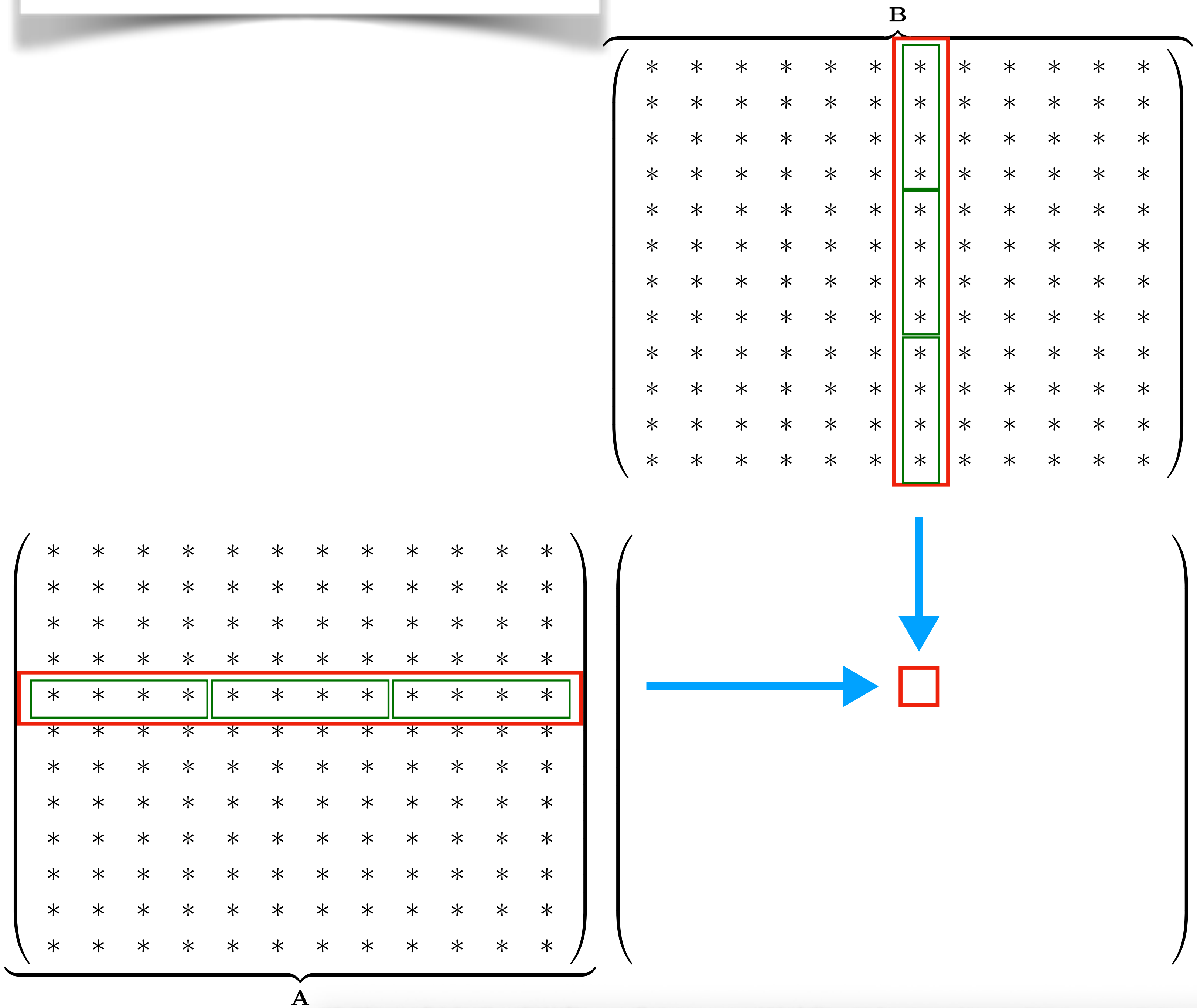
Use transpose?





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

Use transpose?



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

Multiplying with the transpose of B gives better cache utilization!

Two different interpretations:
(1) We multiply with B, stored in column-major format, or
(2) We multiply with B^T, stored in row-major 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 …*

Storage overhead is higher than compute`

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];
    }
}
[.. .]
```

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

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)

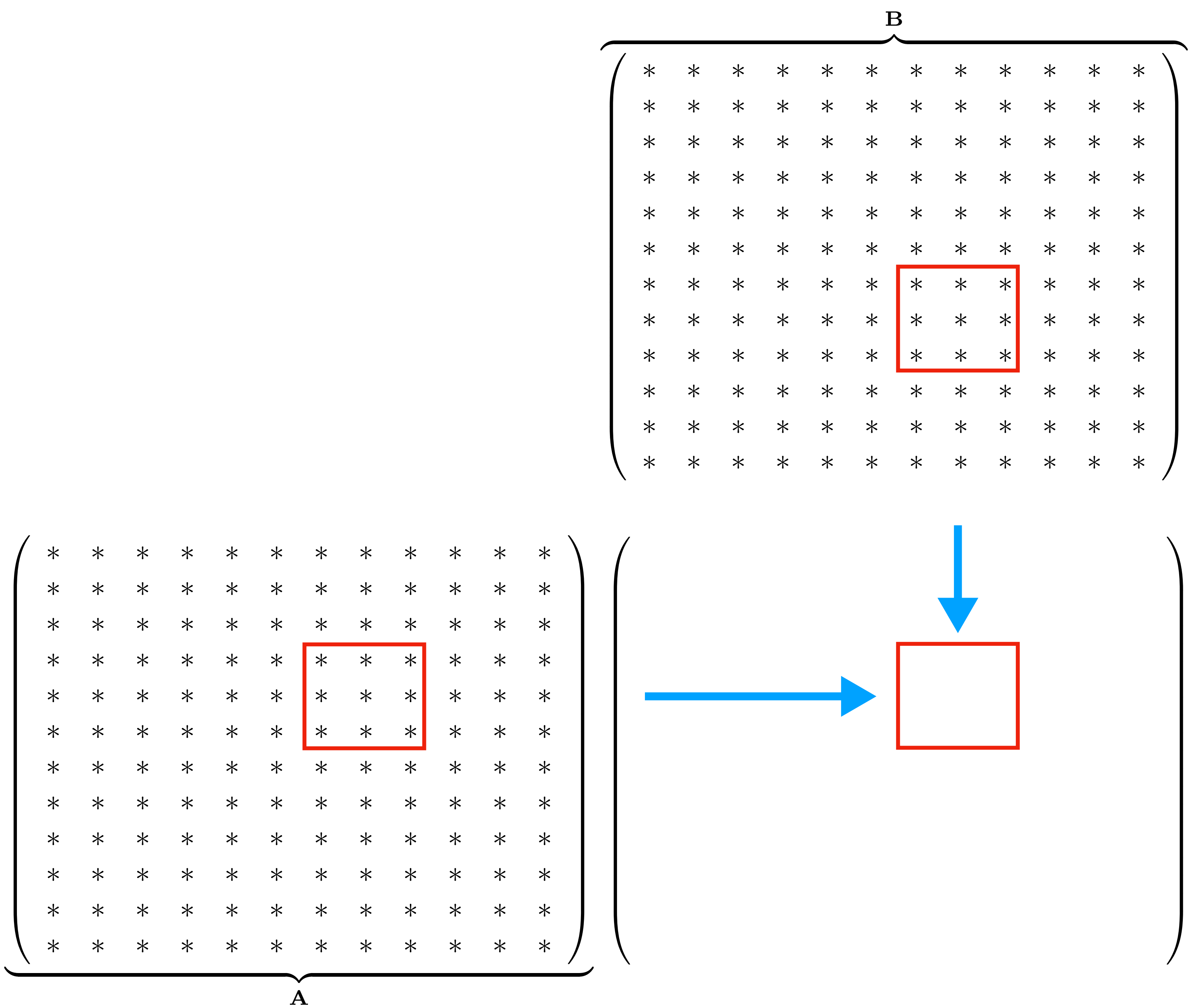
```
[...]
void MatTranspose(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    float (&AT)[MATRIX_SIZE][MATRIX_SIZE])
{
    mkl_somatcopy(
        'R',         // Matrix A is in row-major format
        'T',         // We are performing a transposition operation
        MATRIX_SIZE, // Dimensions of matrix -- 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:

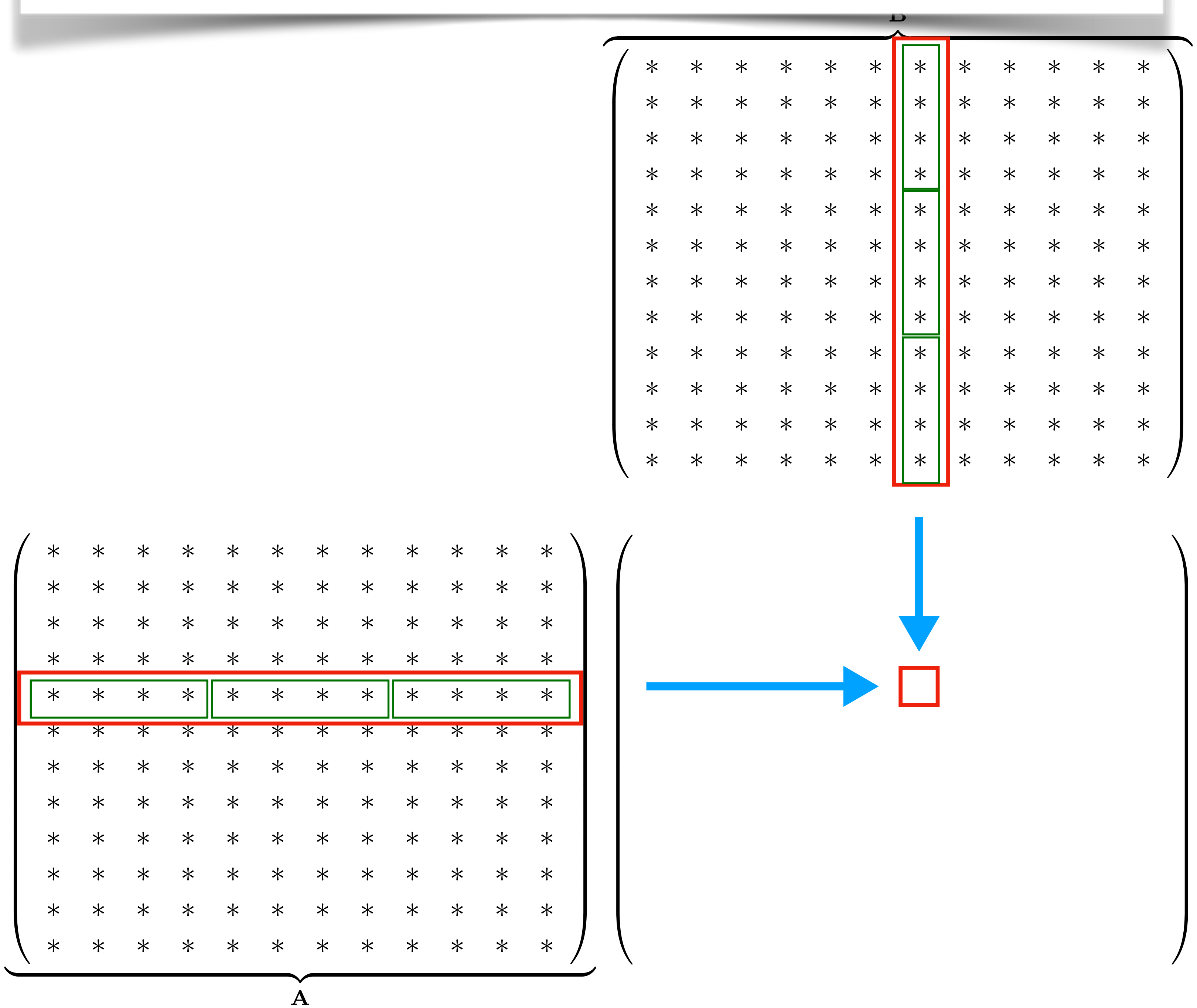
```
Transposing second matrix factor ... [Elapsed time : 28.3228ms]
Running candidate kernel for correctness test ... [Elapsed time : 413.998ms]
Running reference kernel for correctness test ... [Elapsed time : 16.1733ms]
Discrepancy between two methods : 0.000152588
Running kernel for performance run # 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]
[...]
```

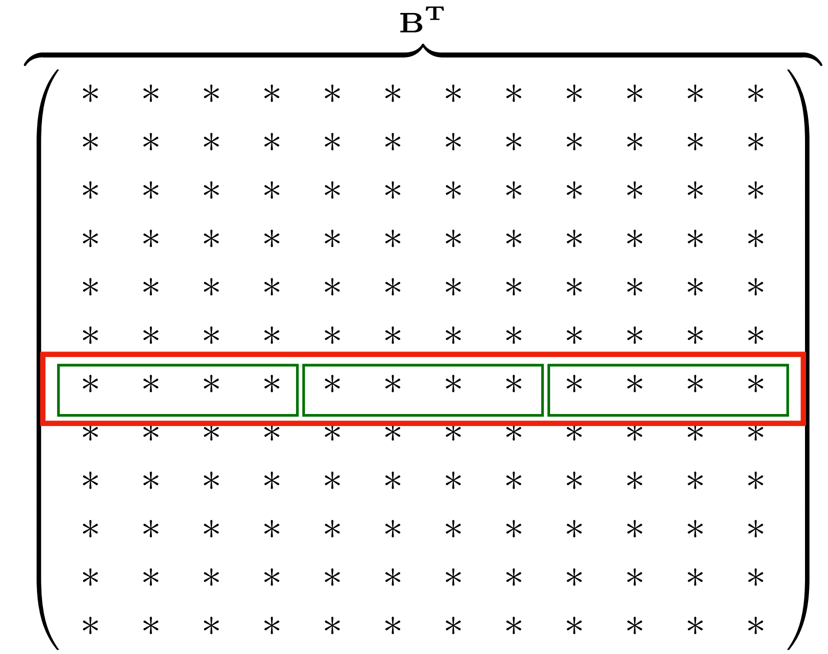
Promising leads: Blocking?



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

Promising leads: Use transpose?

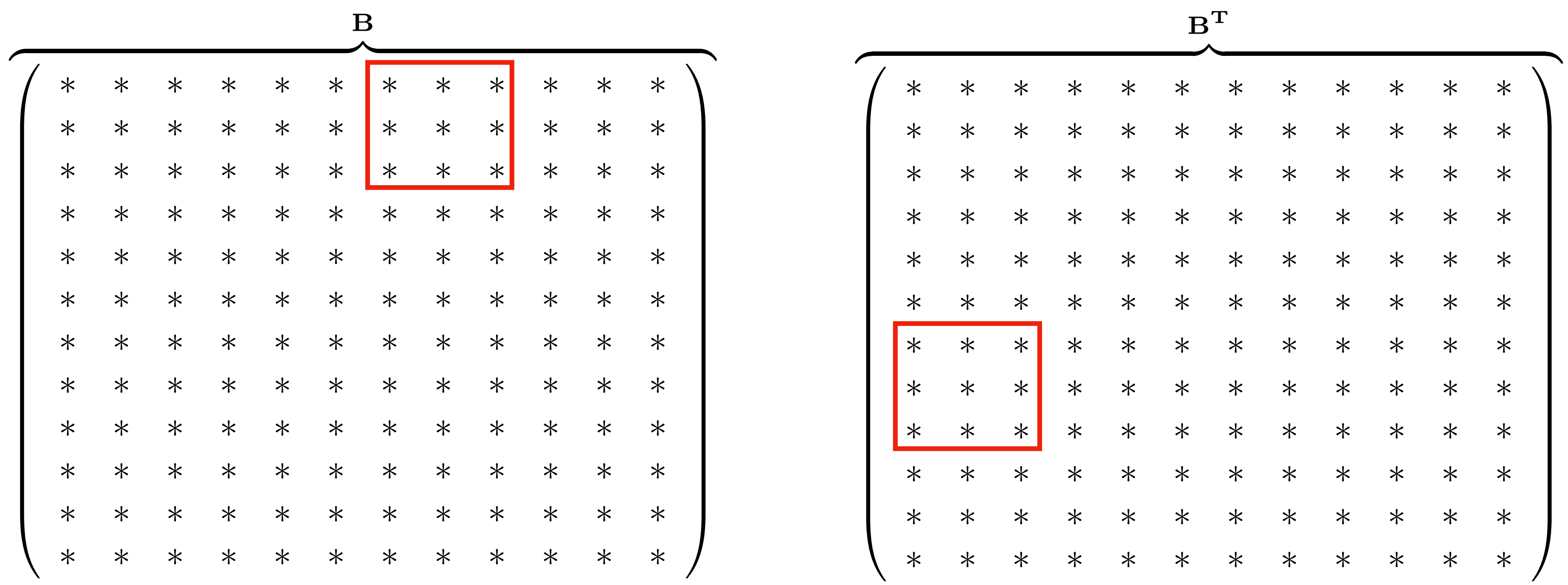


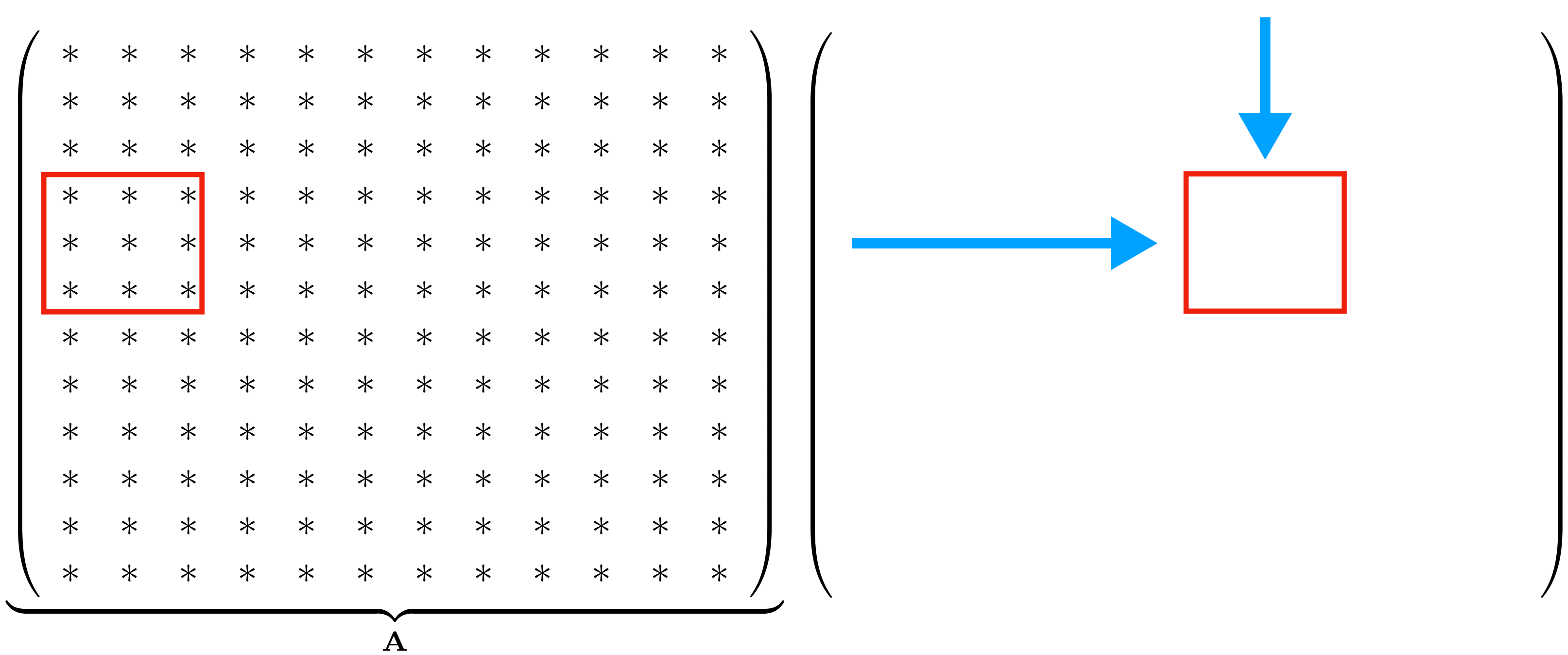


$$
\begin{aligned}
&\texttt{for } i = 1 \ldots N \\
&\quad \texttt{for } j = 1 \ldots N \\
&\quad\quad C_{ij} \leftarrow 0 \\
&\quad\quad \texttt{for } k = 1 \ldots N \\
&\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} [B^T]_{jk}
\end{aligned}
$$

Two different interpretations:
(1) We multiply with B, stored in column-major format, or
(2) We multiply with B^T, stored in row-major format

Combining blocking & pre-transposed B (or col-major B)

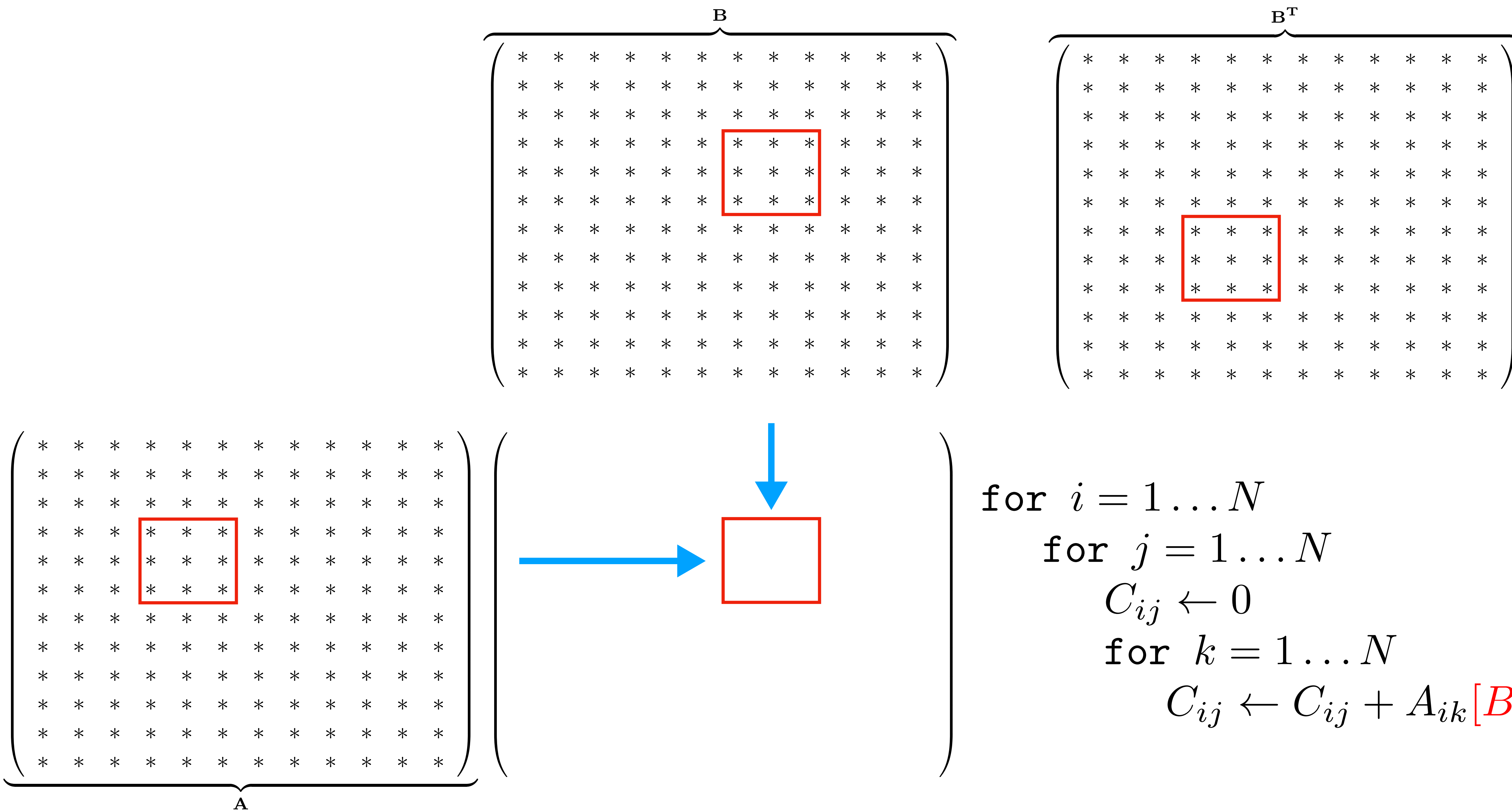




$$
\begin{aligned}
&\texttt{for}\ \ i = 1 \ldots N \\
&\quad \texttt{for}\ \ j = 1 \ldots N \\
&\quad\quad C_{ij} \leftarrow 0 \\
&\quad\quad \texttt{for}\ \ k = 1 \ldots N \\
&\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} [B^T]_{jk}
\end{aligned}
$$

***C_{ij}, A_{ik}** and **$B^T{}_{jk}$** represent <u>block 3x3 sub-matrices</u>*

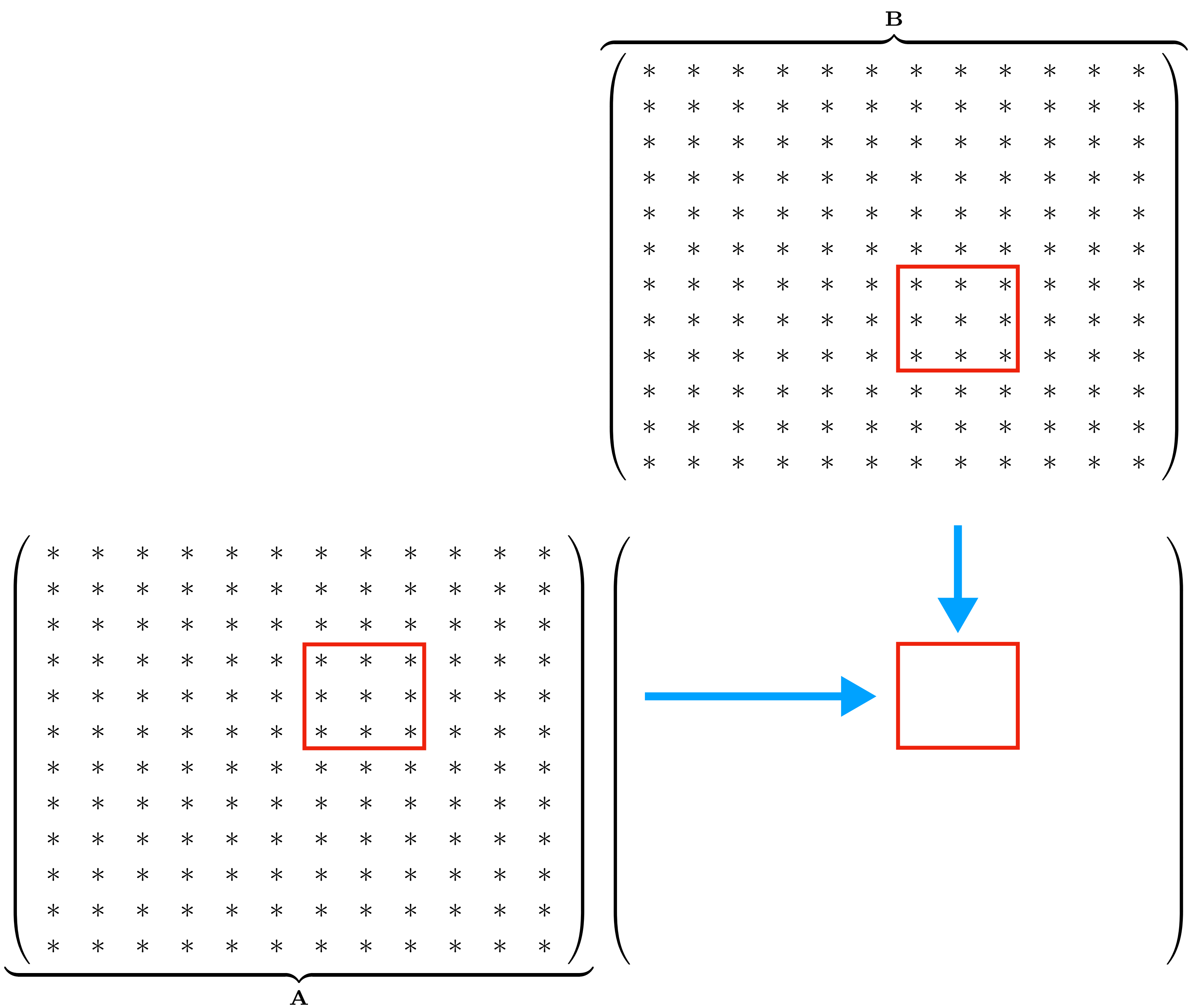
Combining blocking & pre-transposed B (or col-major B)

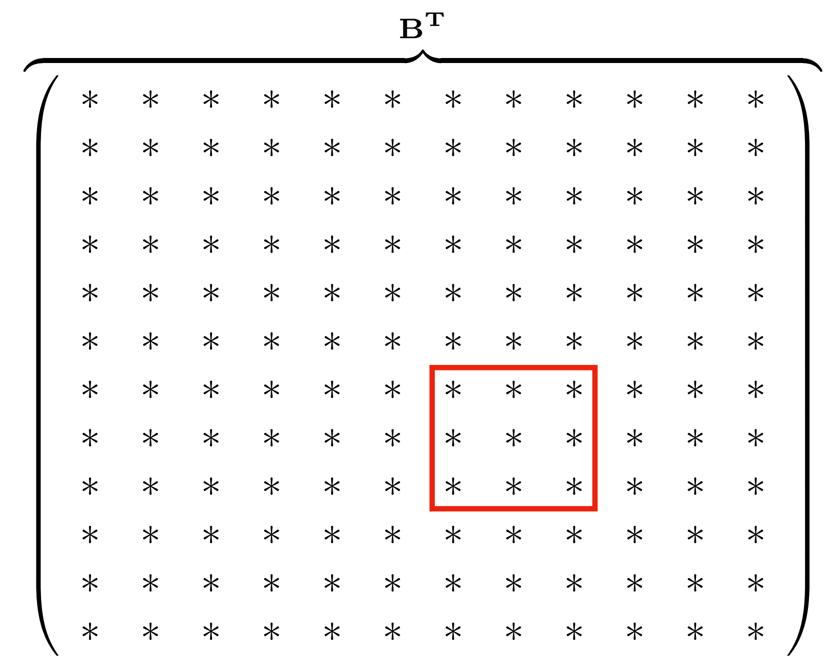


for $i = 1 \ldots N$
    for $j = 1 \ldots N$
        $C_{ij} \leftarrow 0$
        for $k = 1 \ldots N$
            $C_{ij} \leftarrow C_{ij} + A_{ik}[B^T]_{jk}$

***C_{ij}, A_{ik}** and **B^T_{jk}** represent <u>block 3x3 sub-matrices</u>*

Combining blocking & pre-transposed B (or col-major B)

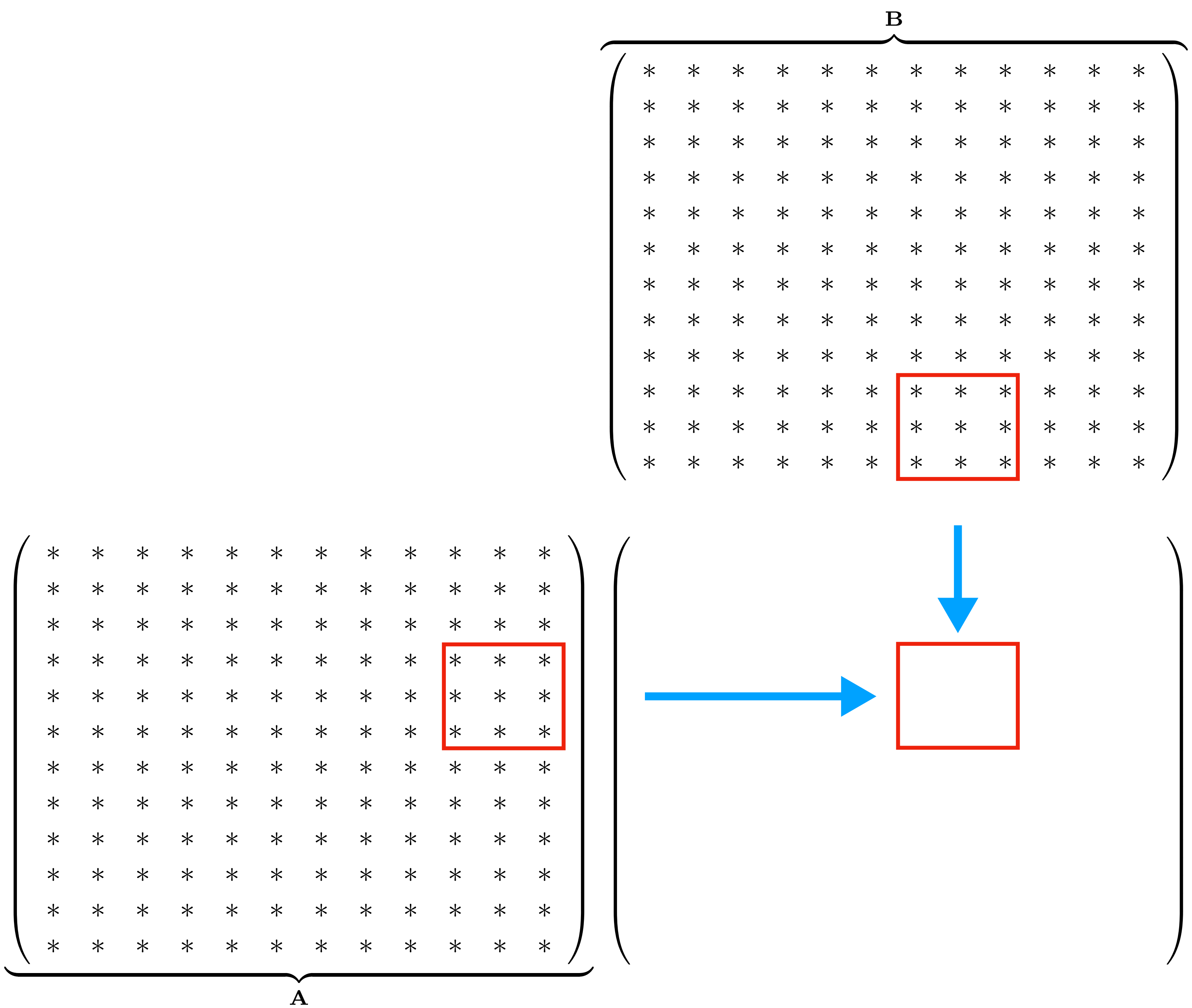




$$
\begin{array}{l}
\texttt{for}\ \ i = 1 \ldots N \\
\quad \texttt{for}\ \ j = 1 \ldots N \\
\quad\quad C_{ij} \leftarrow 0 \\
\quad\quad \texttt{for}\ \ k = 1 \ldots N \\
\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} [B^T]_{jk}
\end{array}
$$

C_{ij}*, *A_{ik} *and* ***B^T_{jk}*** *represent* *block 3x3 sub-matrices*

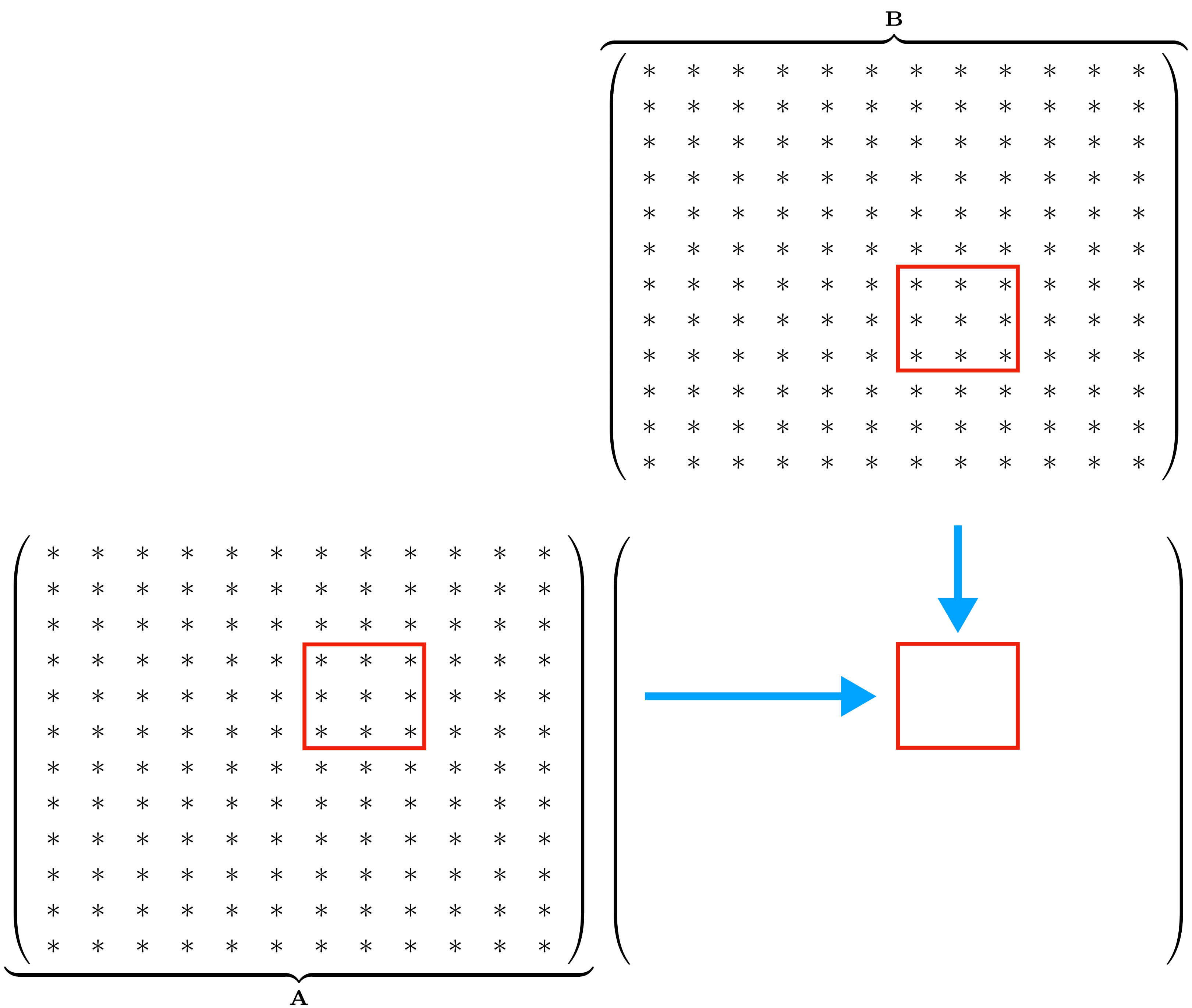
Combining blocking & pre-transposed B (or col-major B)

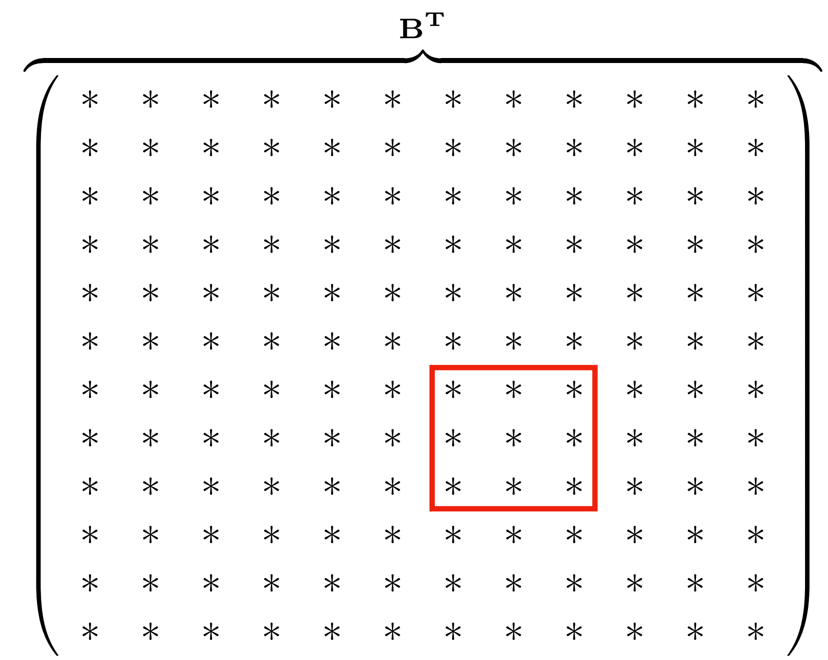


$$
\begin{array}{l}
\texttt{for}\ \ i = 1 \dots N \\
\quad \texttt{for}\ \ j = 1 \dots N \\
\quad\quad C_{ij} \leftarrow 0 \\
\quad\quad \texttt{for}\ \ k = 1 \dots N \\
\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} [B^T]_{jk}
\end{array}
$$

***C_{ij}, A_{ik}** and **$B^T{}_{jk}$** represent <u>block 3x3 sub-matrices</u>*

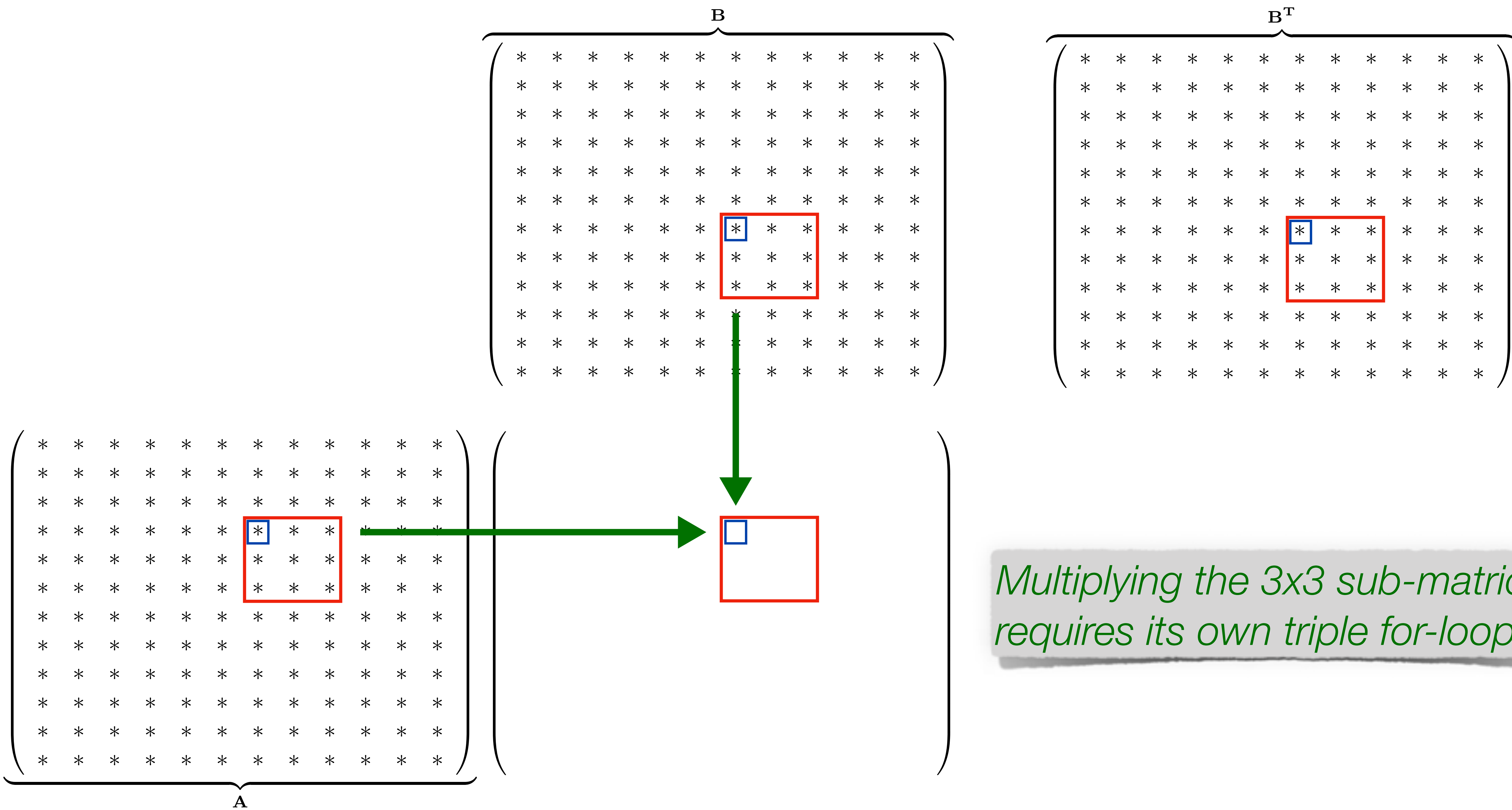
Combining blocking & pre-transposed B (or col-major B)





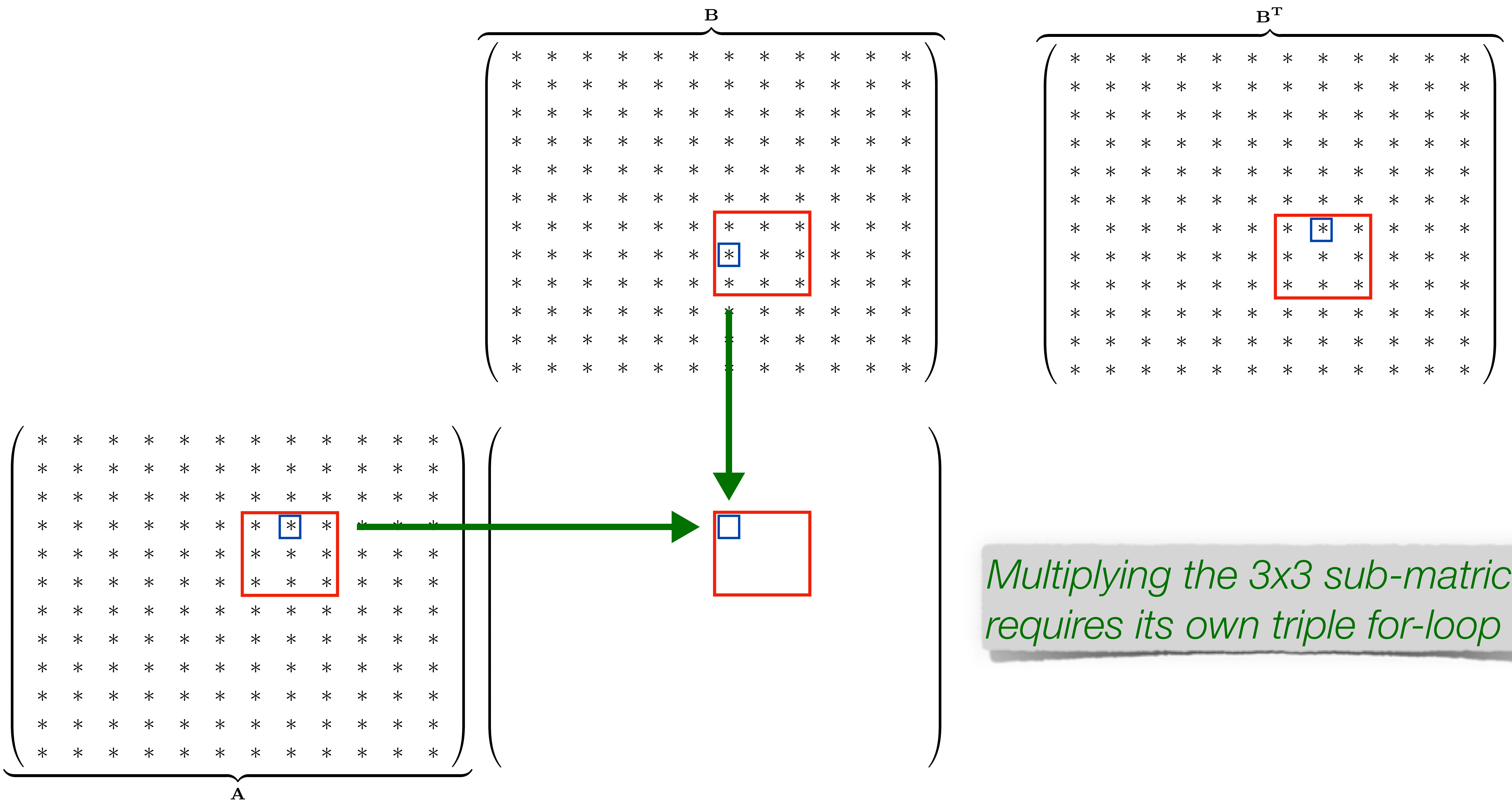
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

Combining blocking & pre-transposed B (or col-major B)



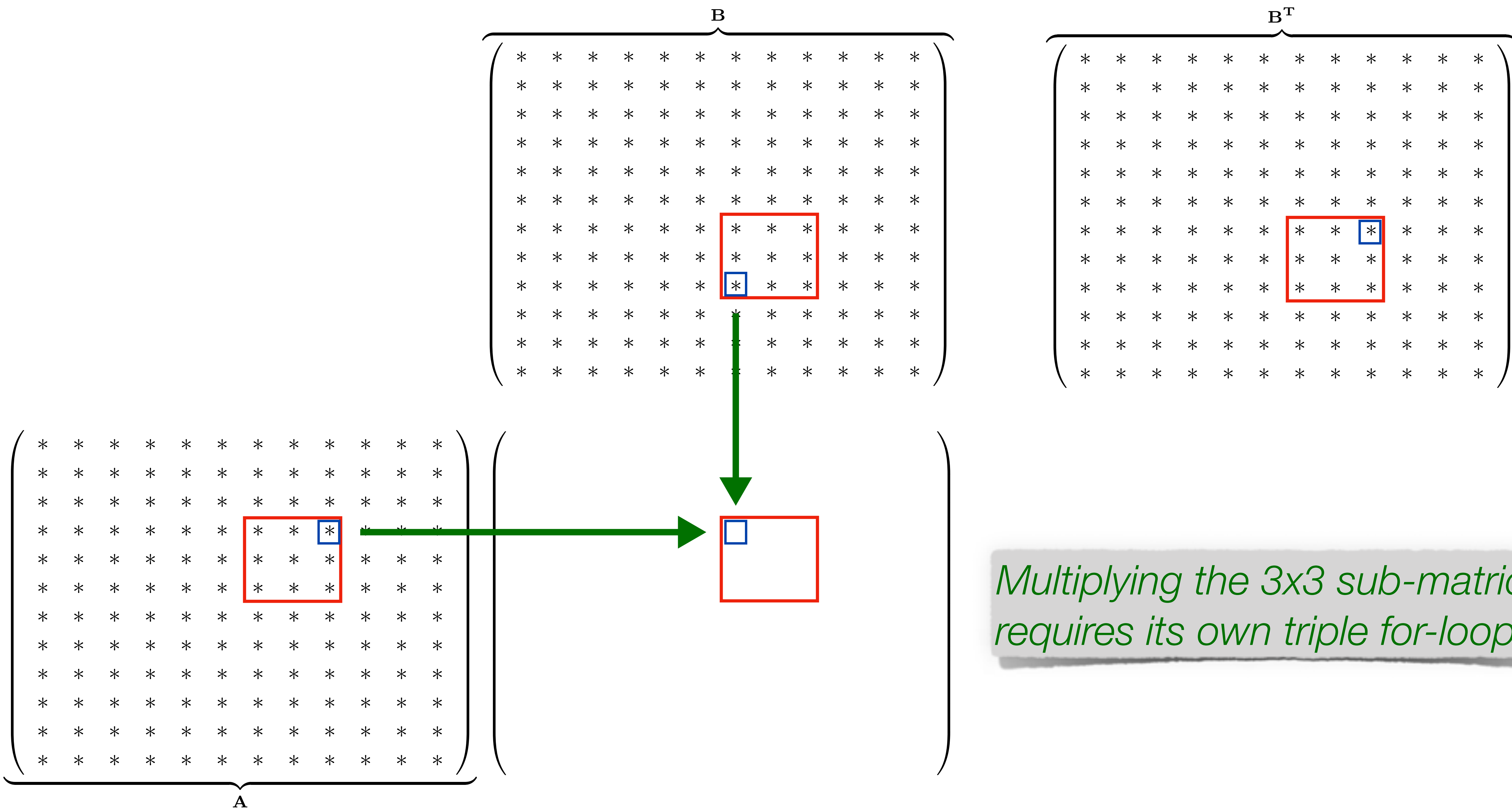
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

Combining blocking & pre-transposed B (or col-major B)



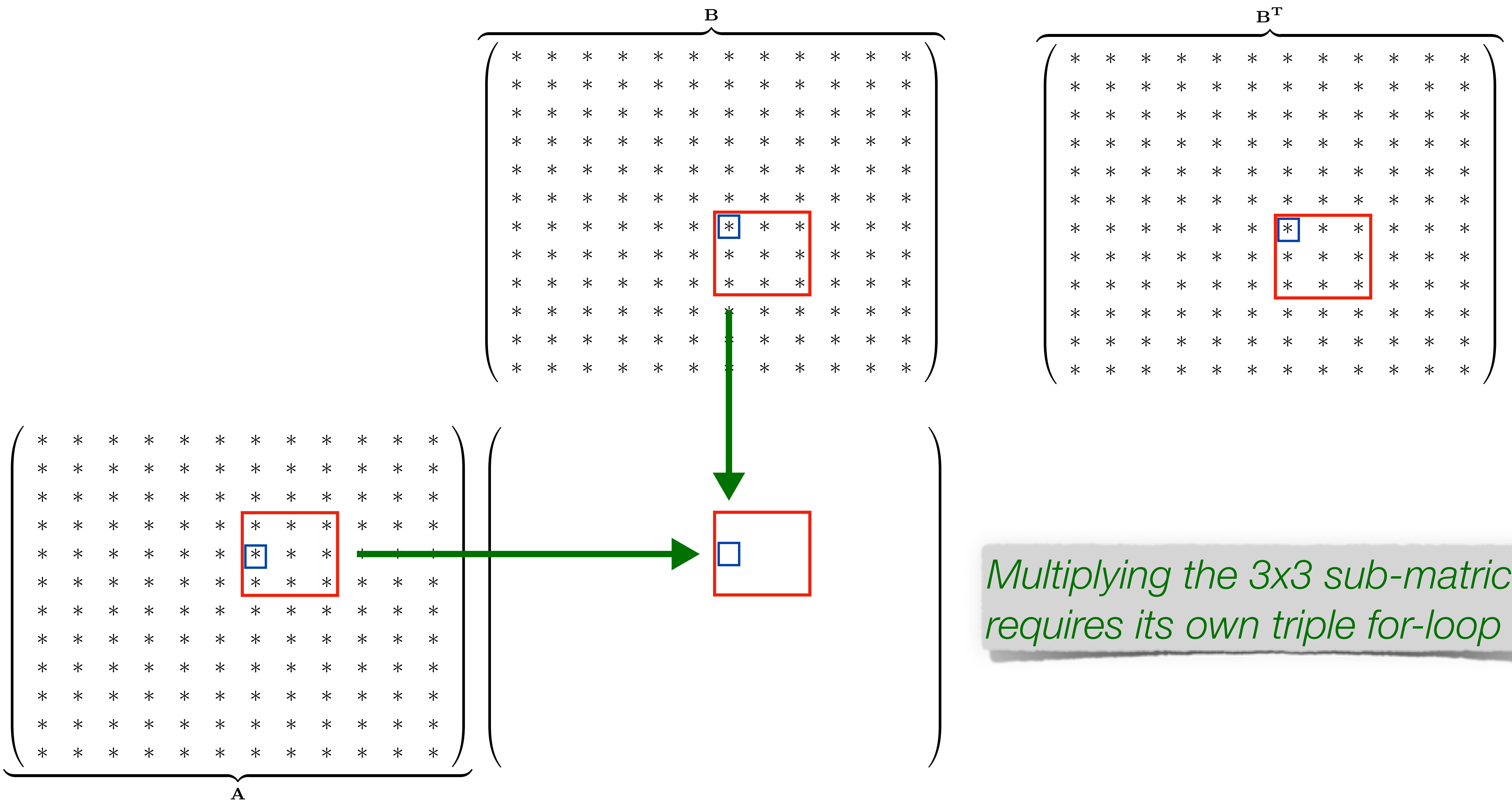
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

Combining blocking & pre-transposed B (or col-major B)



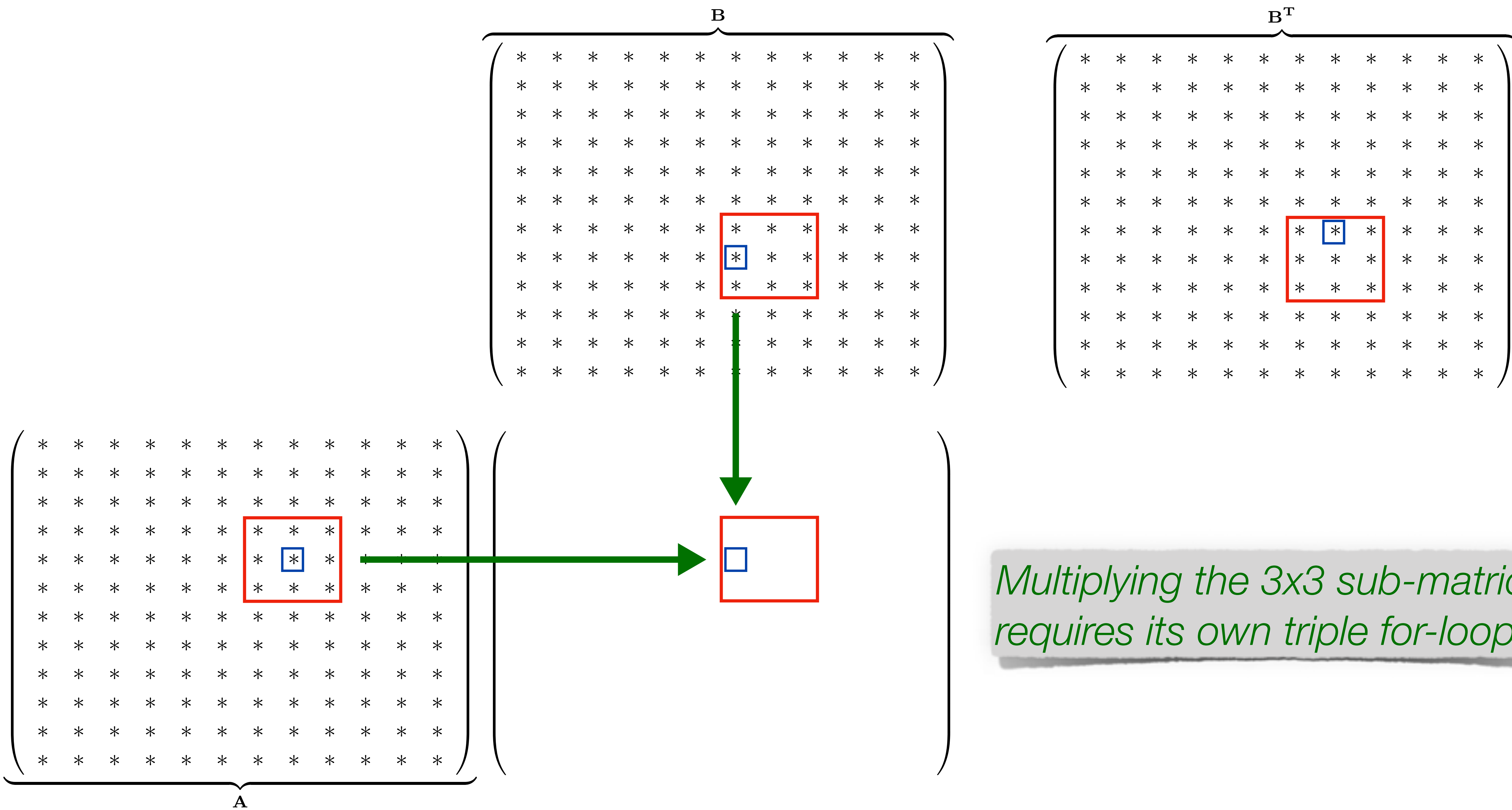
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

Combining blocking & pre-transposed B (or col-major B)



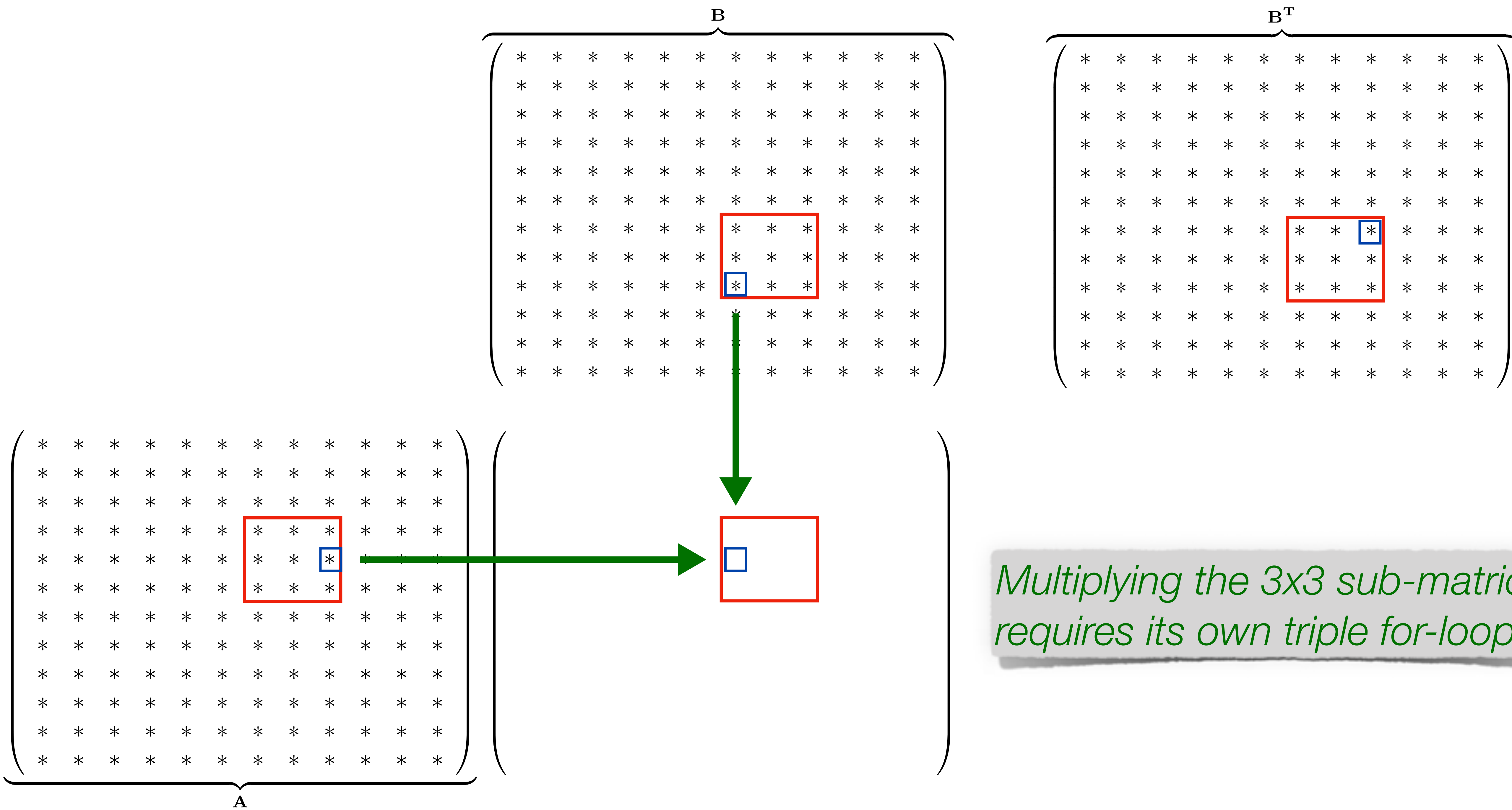
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

Combining blocking & pre-transposed B (or col-major B)



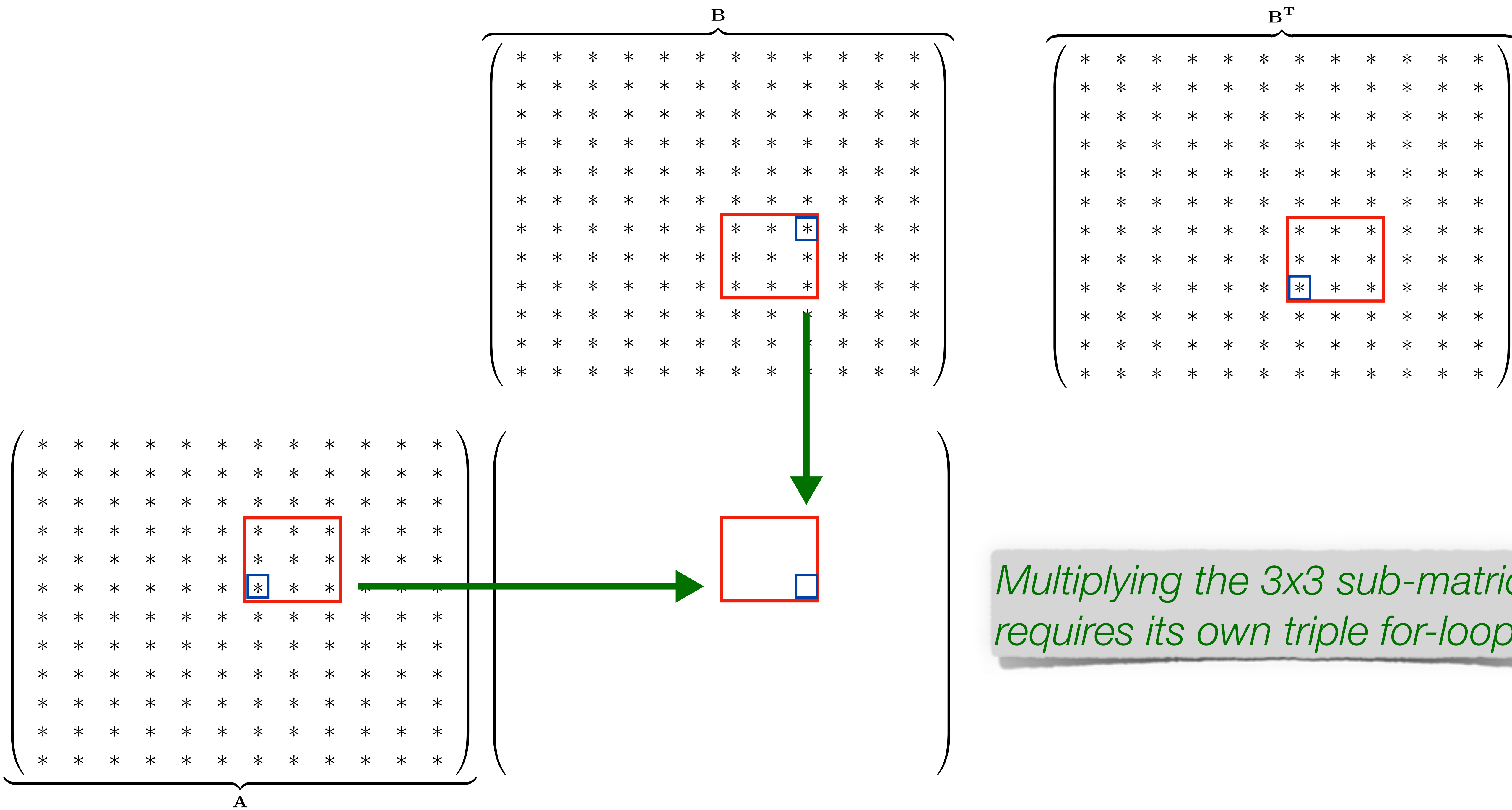
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

Combining blocking & pre-transposed B (or col-major B)



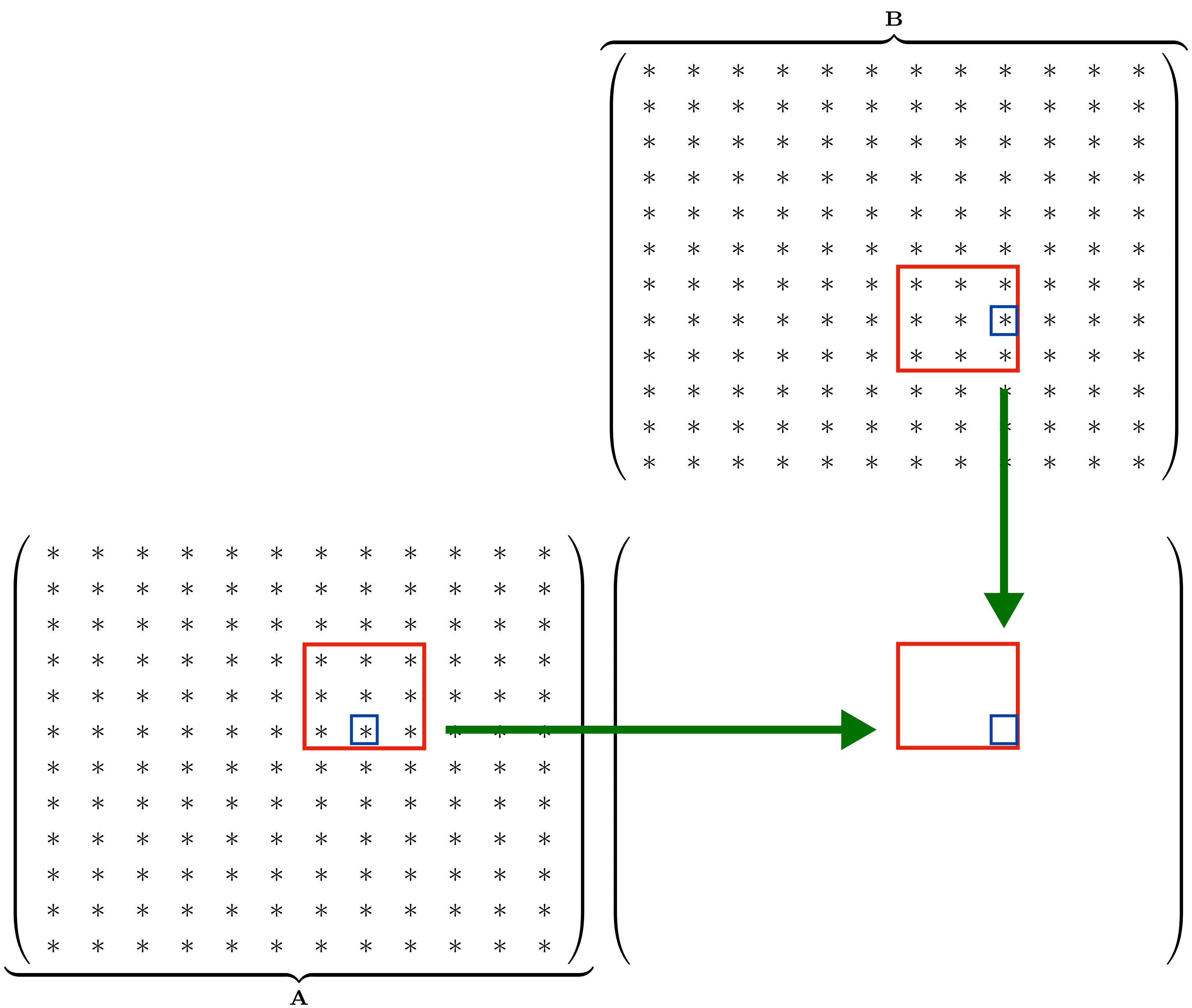
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

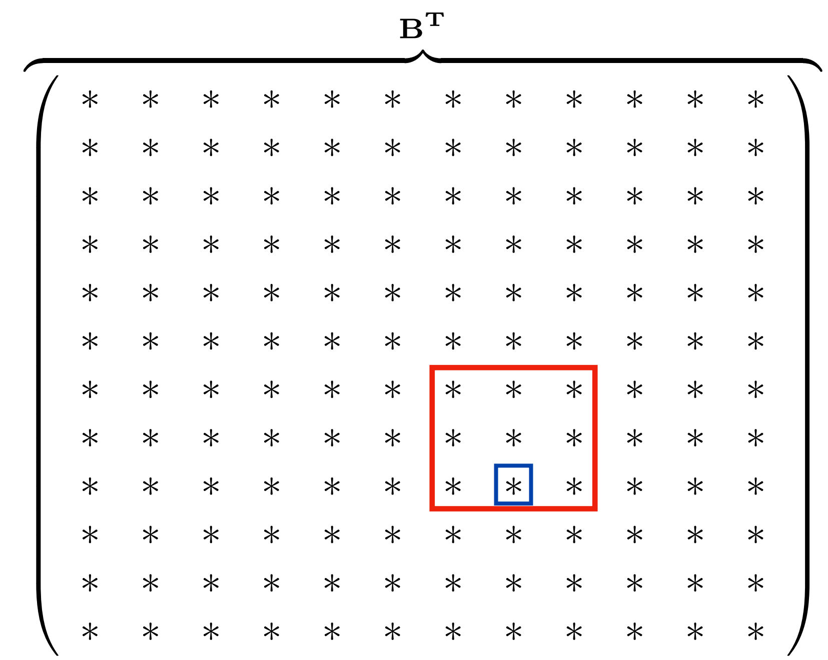
Combining blocking & pre-transposed B (or col-major B)



Multiplying the 3x3 sub-matrices requires its own triple for-loop …

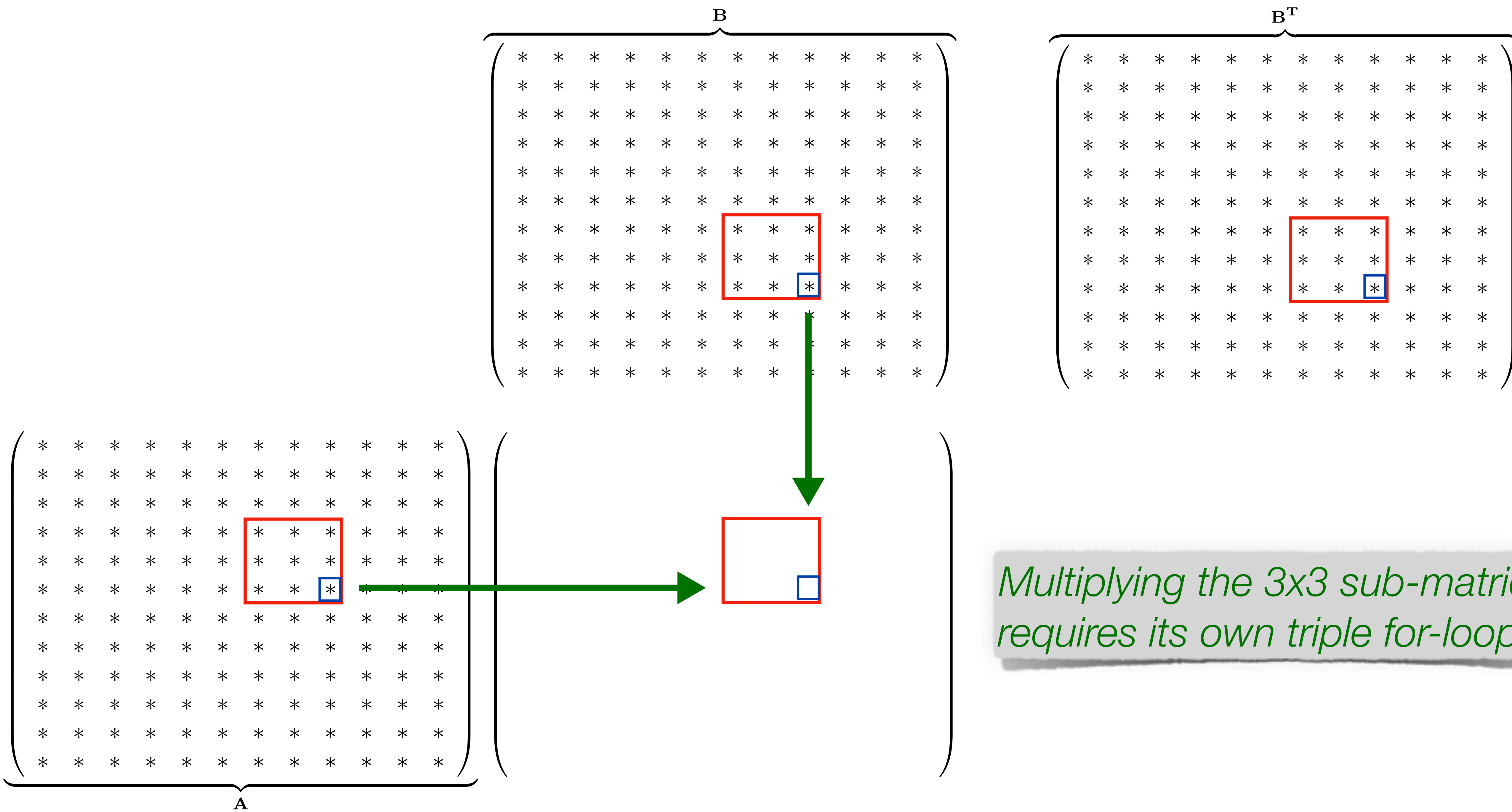
Combining blocking & pre-transposed B (or col-major B)





Multiplying the 3x3 sub-matrices requires its own triple for-loop …

Combining blocking & pre-transposed B (or col-major B)



Multiplying the 3x3 sub-matrices requires its own triple for-loop …

Kernel parameters (Parameters.h)

DenseAlgebra/GEMM_Test_0_5

```
#pragma once

#ifndef MATRIX_SIZE
#define MATRIX_SIZE 1024
#endif

#ifndef BLOCK_SIZE
#define BLOCK_SIZE 32
#endif
```

Kernel parameters (Parameters.h)

DenseAlgebra/GEMM_Test_0_5

```
#pragma once

#ifndef MATRIX_SIZE
#define MATRIX_SIZE 1024
#endif

#ifndef BLOCK_SIZE
#define BLOCK_SIZE 32
#endif
```

We presume we know, at compile-time, both the matrix size and the size of the sub-matrix blocks

Kernel parameters (Parameters.h)

DenseAlgebra/GEMM_Test_0_5

```
#pragma once

#ifndef MATRIX_SIZE
#define MATRIX_SIZE 1024
#endif

#ifndef BLOCK_SIZE
#define BLOCK_SIZE 32
#endif
```

*#define guards make it easy to override dimensions via compiler options, for testing (e.g. **-DMATRIX_SIZE=1024 -DBLOCK_SIZE=32**)*

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_5

```
#include “MatMatMultiply.h"
[.. .]

void MatMatTransposeMultiply(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]);

    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; 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++)
            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[bj][jj][bk][kk];
}
```

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_5

```
#include "MatMatMultiply.h"
[.. .]

void MatMatTransposeMultiply(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]);

    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; 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++)
            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[bj][jj][bk][kk];
}
```

Multiply using pre-transposed matrix B (which is treated as column-major) … just like GEMM_Test_0_4

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_5

```
#include “MatMatMultiply.h"
[.. .]

void MatMatTransposeMultiply(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]);

    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; 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++)
            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[bj][jj][bk][kk];
}
```

Re-cast the input/output matrices so we can index them with blockID/subelementID … just like GEMM_Test_0_2

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_5

```
#include “MatMatMultiply.h"
[.. .]

void MatMatTransposeMultiply(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]);

    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; 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++)
            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[bj][jj][bk][kk];
}
```

*Zero out the matrix **C** in the beginning*
(easier to do, only N^2 operations/accesses)

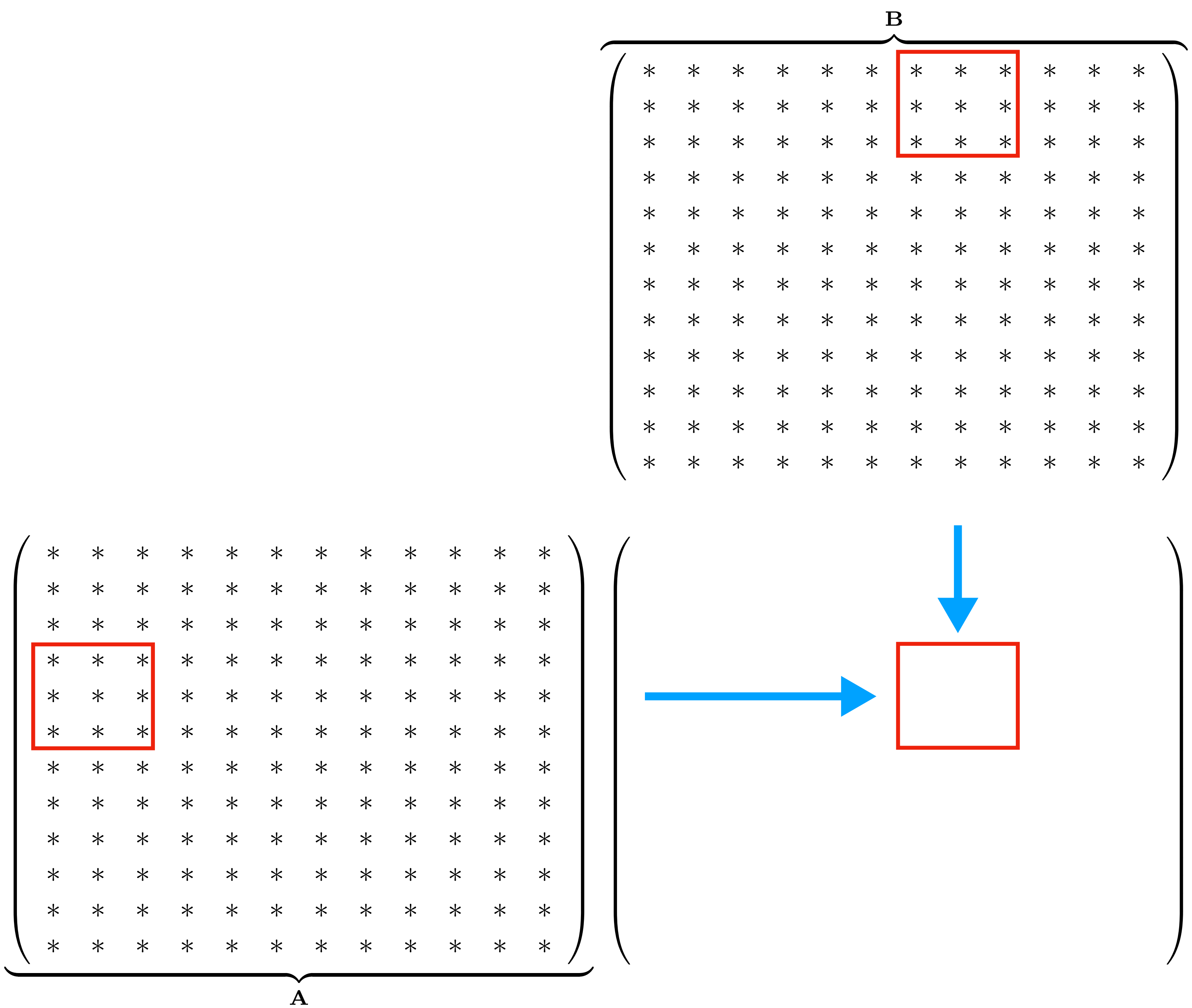
GEMM routine (MatMatMultiply.cpp)

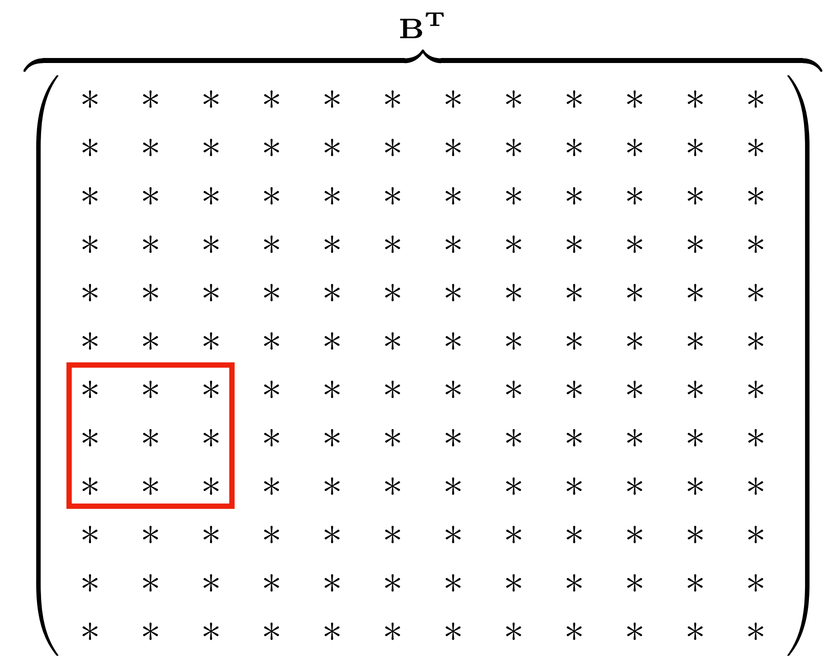
DenseAlgebra/GEMM_Test_0_5

```
#include “MatMatMultiply.h"
[.. .]

void MatMatTransposeMultiply(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]);

    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; 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++)
            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[bj][jj][bk][kk];
}
```

Iterate to do multiplication of ***blocks*** *…*

Combining blocking & pre-transposed B (or col-major B)

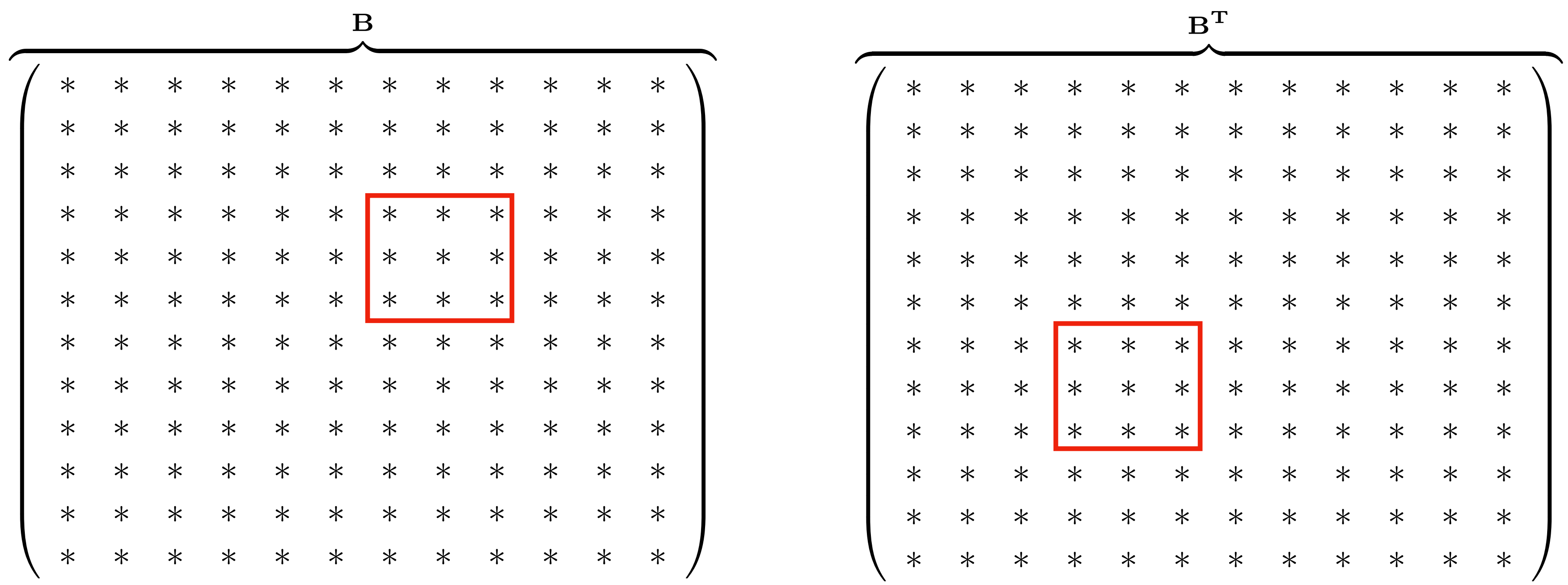


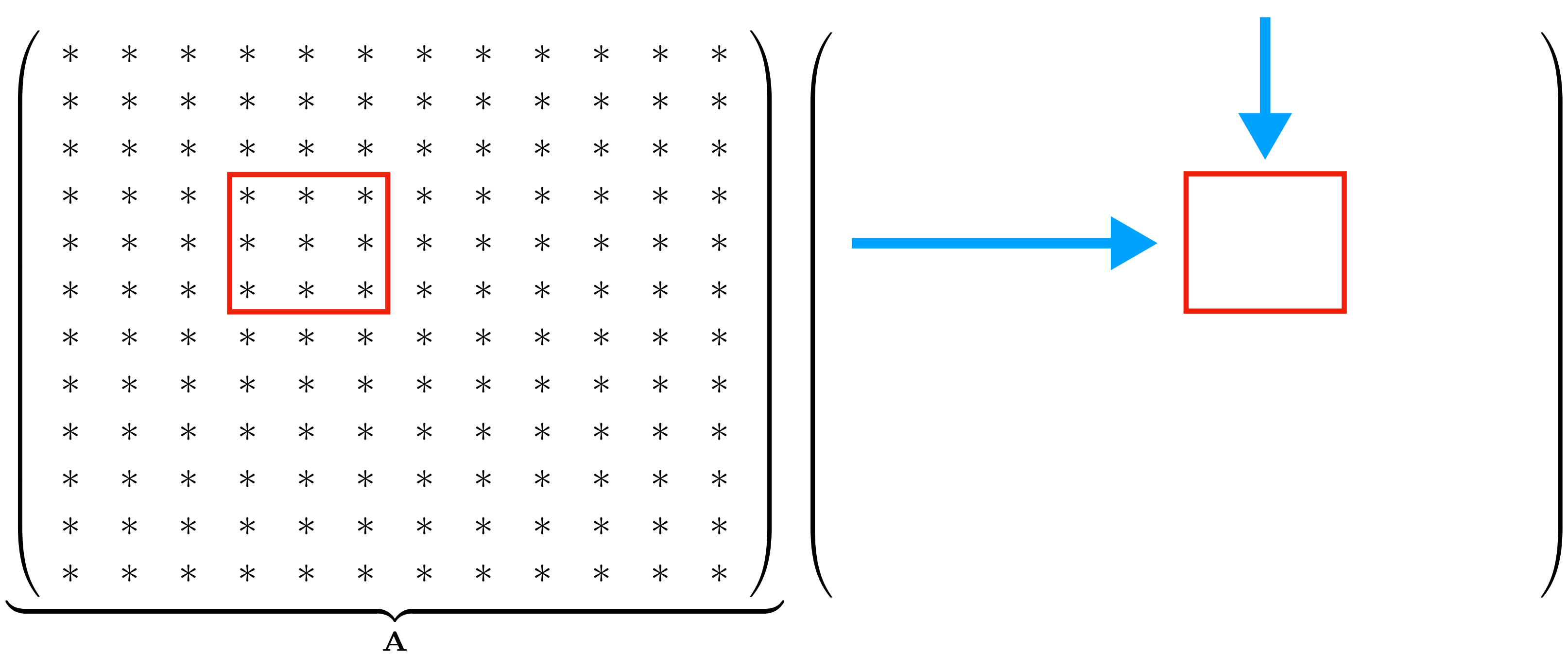


for $i = 1 \ldots N$
    for $j = 1 \ldots N$
        $C_{ij} \leftarrow 0$
        for $k = 1 \ldots N$
            $C_{ij} \leftarrow C_{ij} + A_{ik}[B^T]_{jk}$

***C_{ij}, A_{ik}** and **$B^T{}_{jk}$** represent <u>block 3x3 sub-matrices</u>*

Combining blocking & pre-transposed B (or col-major B)

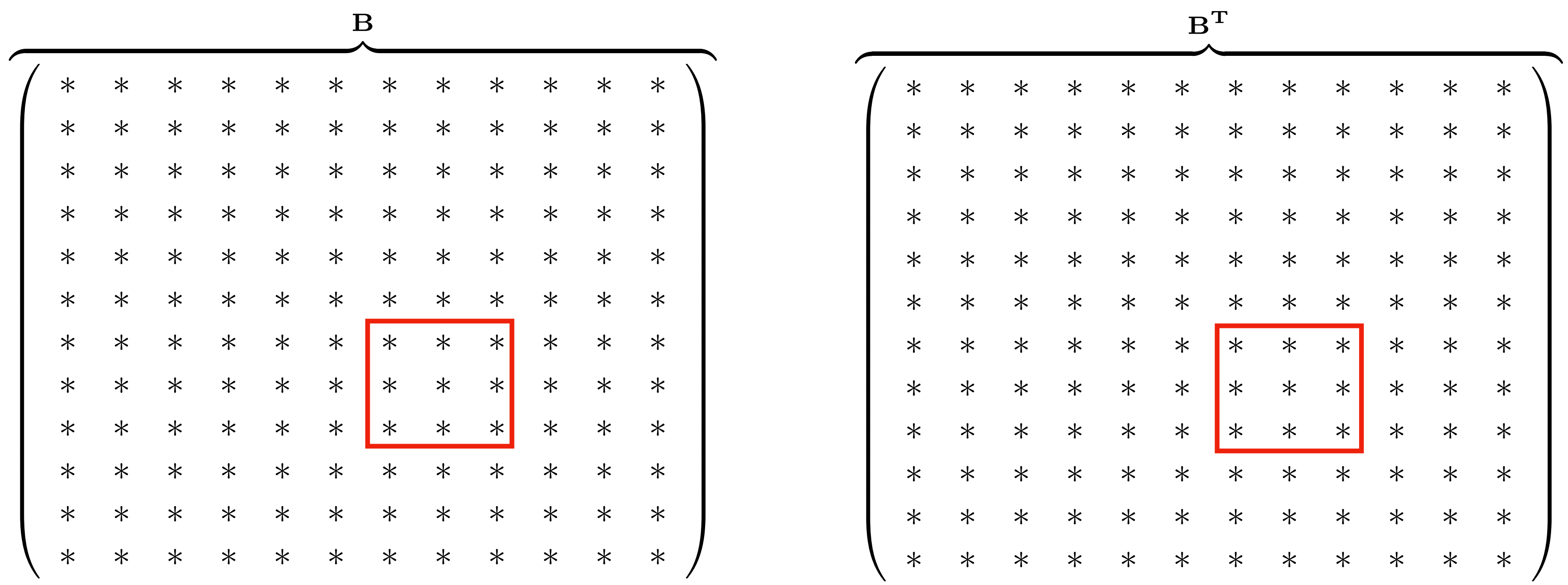


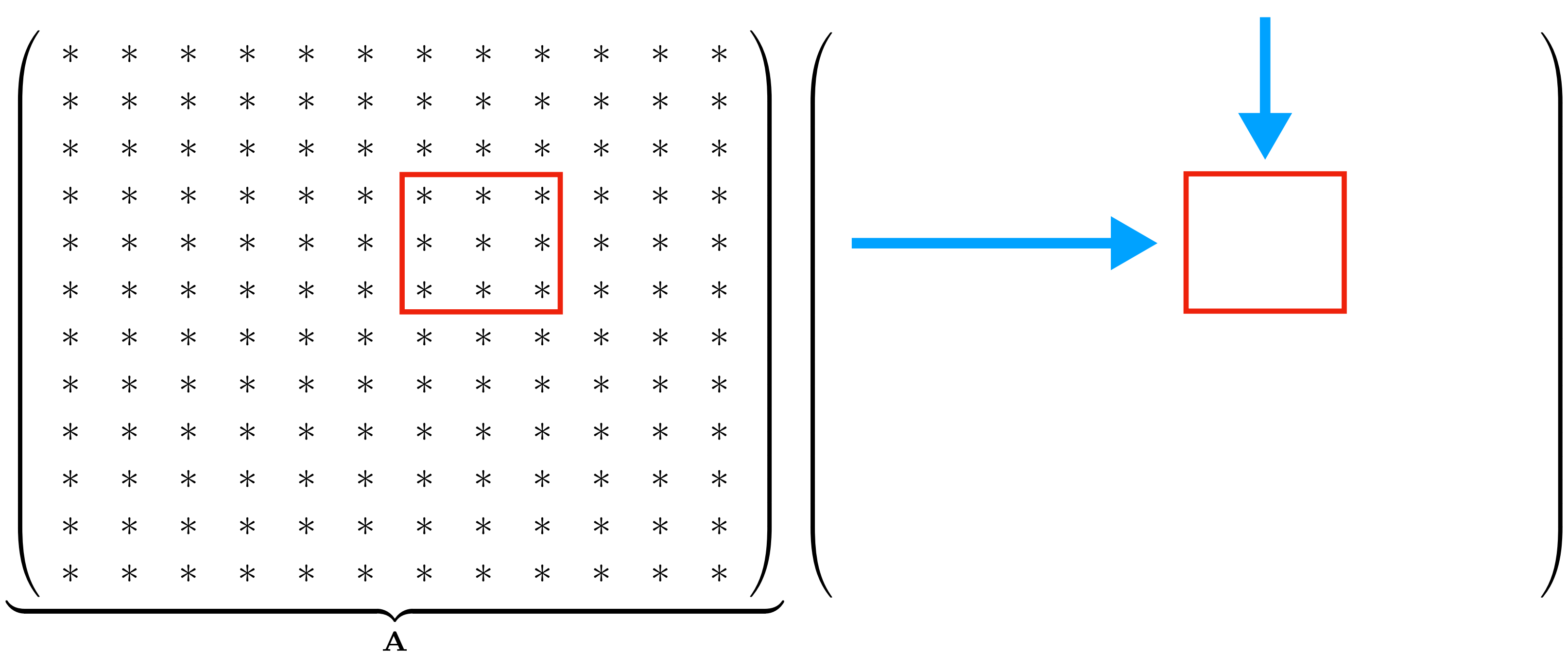


$$
\begin{array}{l}
\texttt{for } i = 1 \ldots N \\
\quad \texttt{for } j = 1 \ldots N \\
\quad\quad C_{ij} \leftarrow 0 \\
\quad\quad \texttt{for } k = 1 \ldots N \\
\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik}[B^T]_{jk}
\end{array}
$$

***C_{ij}, A_{ik}** and **$B^T{}_{jk}$** represent <u>block 3x3 sub-matrices</u>*

Combining blocking & pre-transposed B (or col-major B)

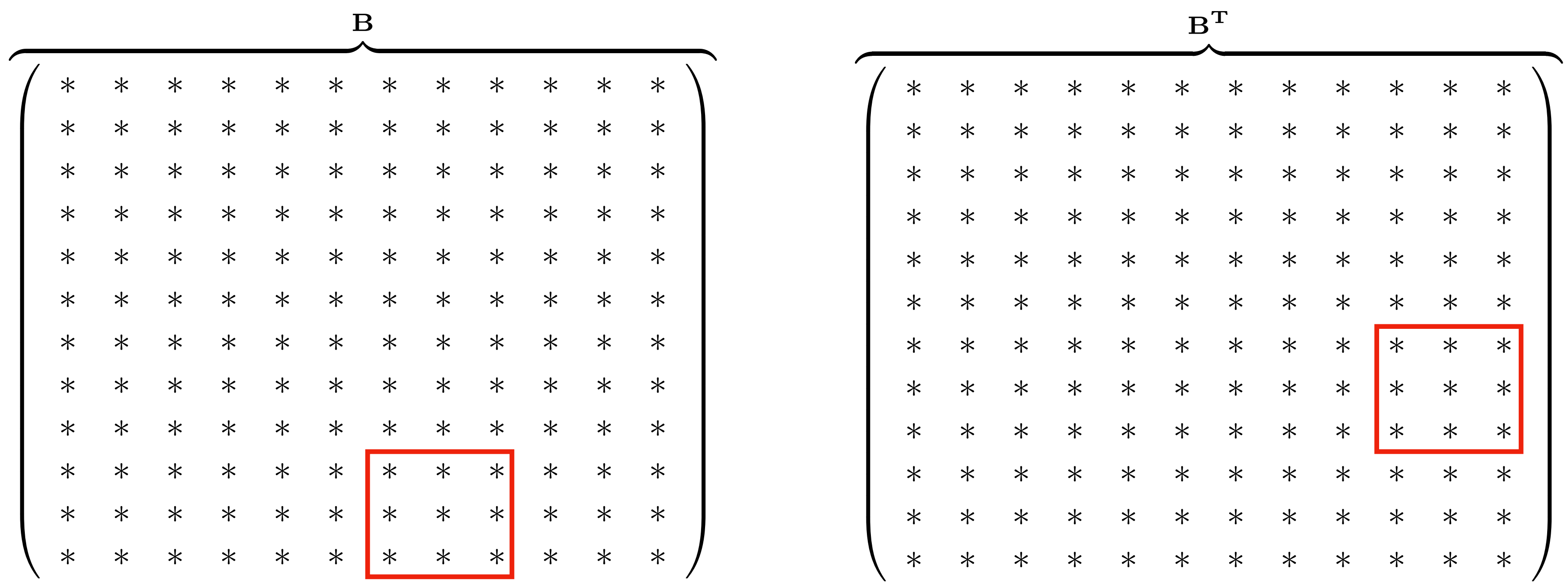


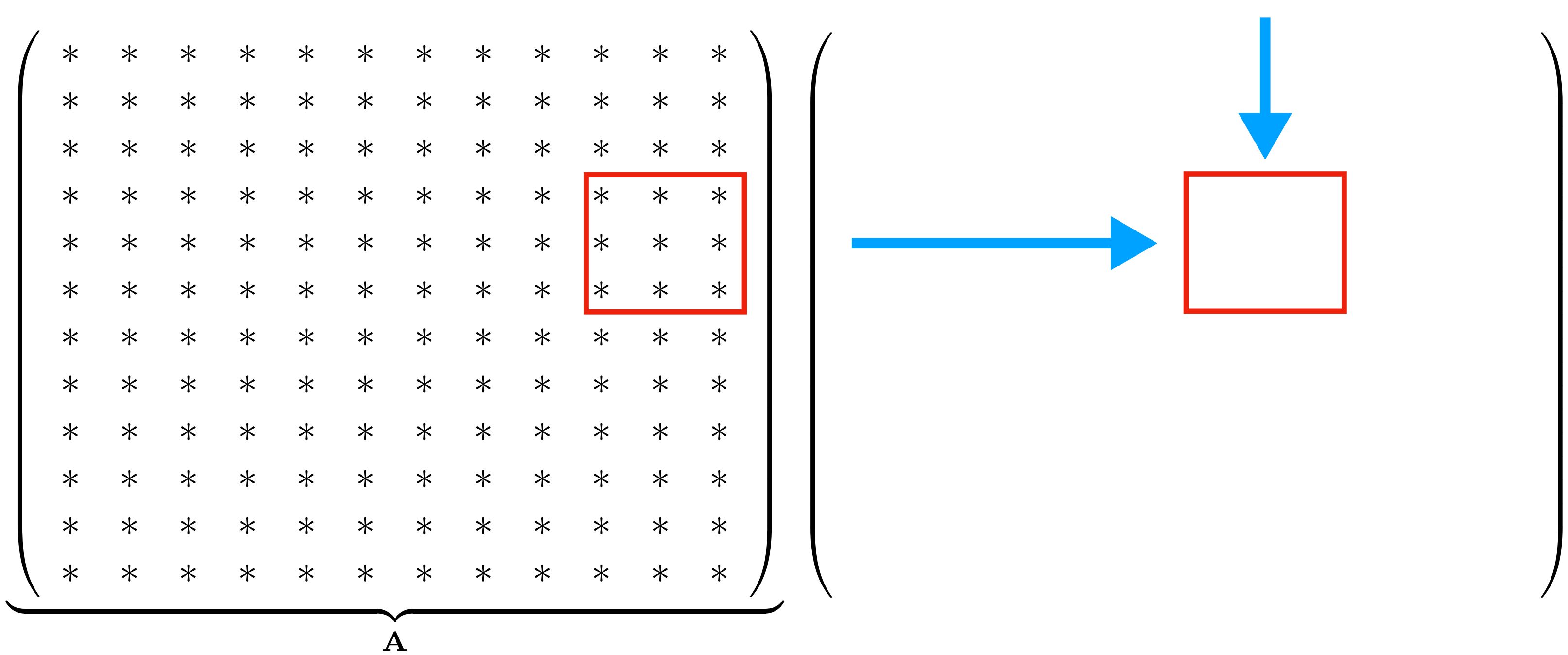


$$
\begin{array}{l}
\texttt{for}\ \ i = 1 \ldots N \\
\quad \texttt{for}\ \ j = 1 \ldots N \\
\quad\quad C_{ij} \leftarrow 0 \\
\quad\quad \texttt{for}\ \ k = 1 \ldots N \\
\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik}[B^T]_{jk}
\end{array}
$$

***C_{ij}, A_{ik}** and **$B^T{}_{jk}$** represent <u>block 3x3 sub-matrices</u>*

Combining blocking & pre-transposed B (or col-major B)





for $i = 1 \dots N$
    for $j = 1 \dots N$
        $C_{ij} \leftarrow 0$
        for $k = 1 \dots N$
            $C_{ij} \leftarrow C_{ij} + A_{ik}[B^T]_{jk}$

***C_{ij}, A_{ik}** and **B^T_{jk}** represent <u>block 3x3 sub-matrices</u>*

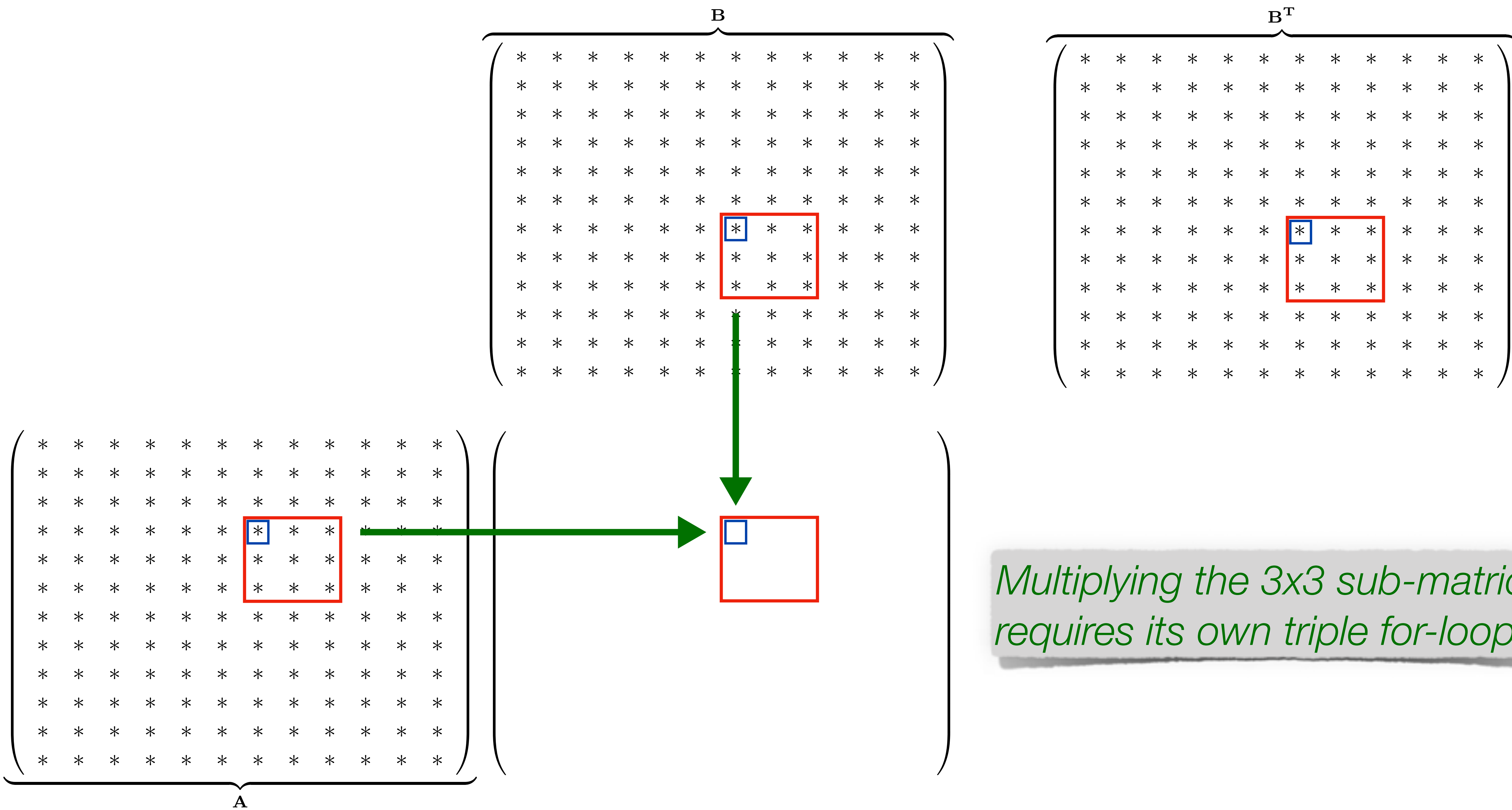
GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_5

```
#include “MatMatMultiply.h"
[.. .]

void MatMatTransposeMultiply(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]);

    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; 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++)
            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[bj][jj][bk][kk];
}
```

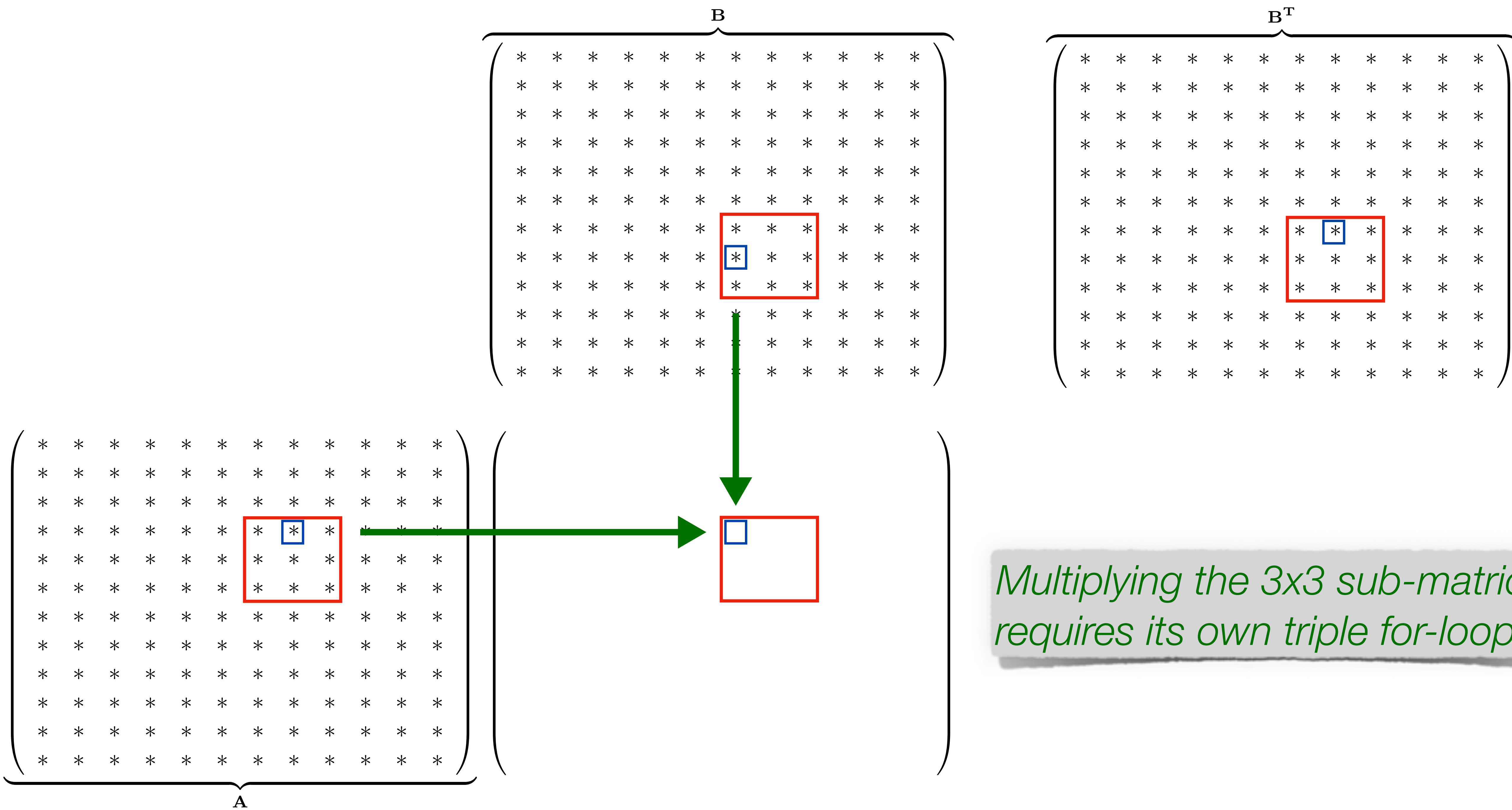
*… and iterate again **within** the blocks, to multiply them together*

Combining blocking & pre-transposed B (or col-major B)



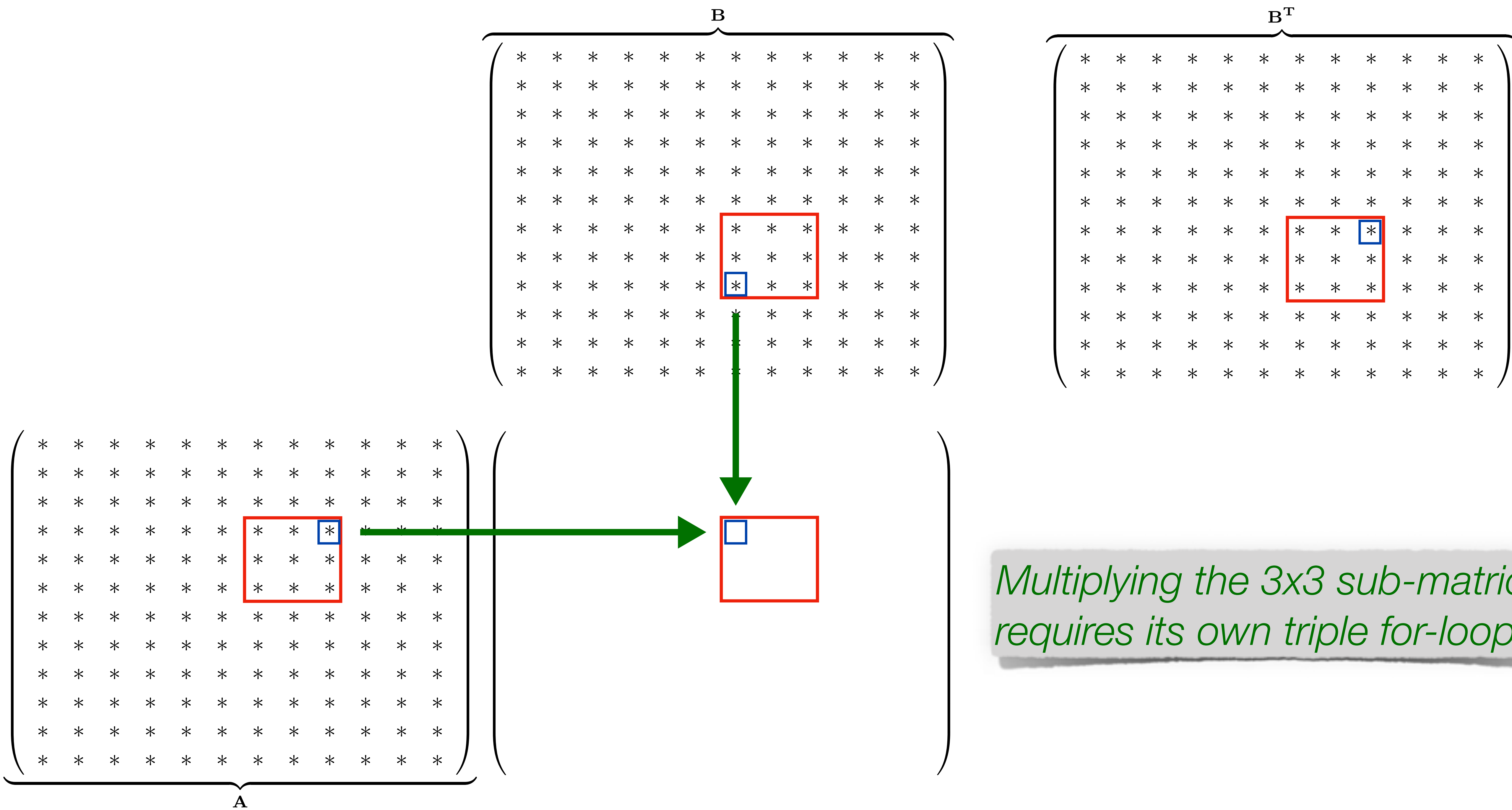
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

Combining blocking & pre-transposed B (or col-major B)



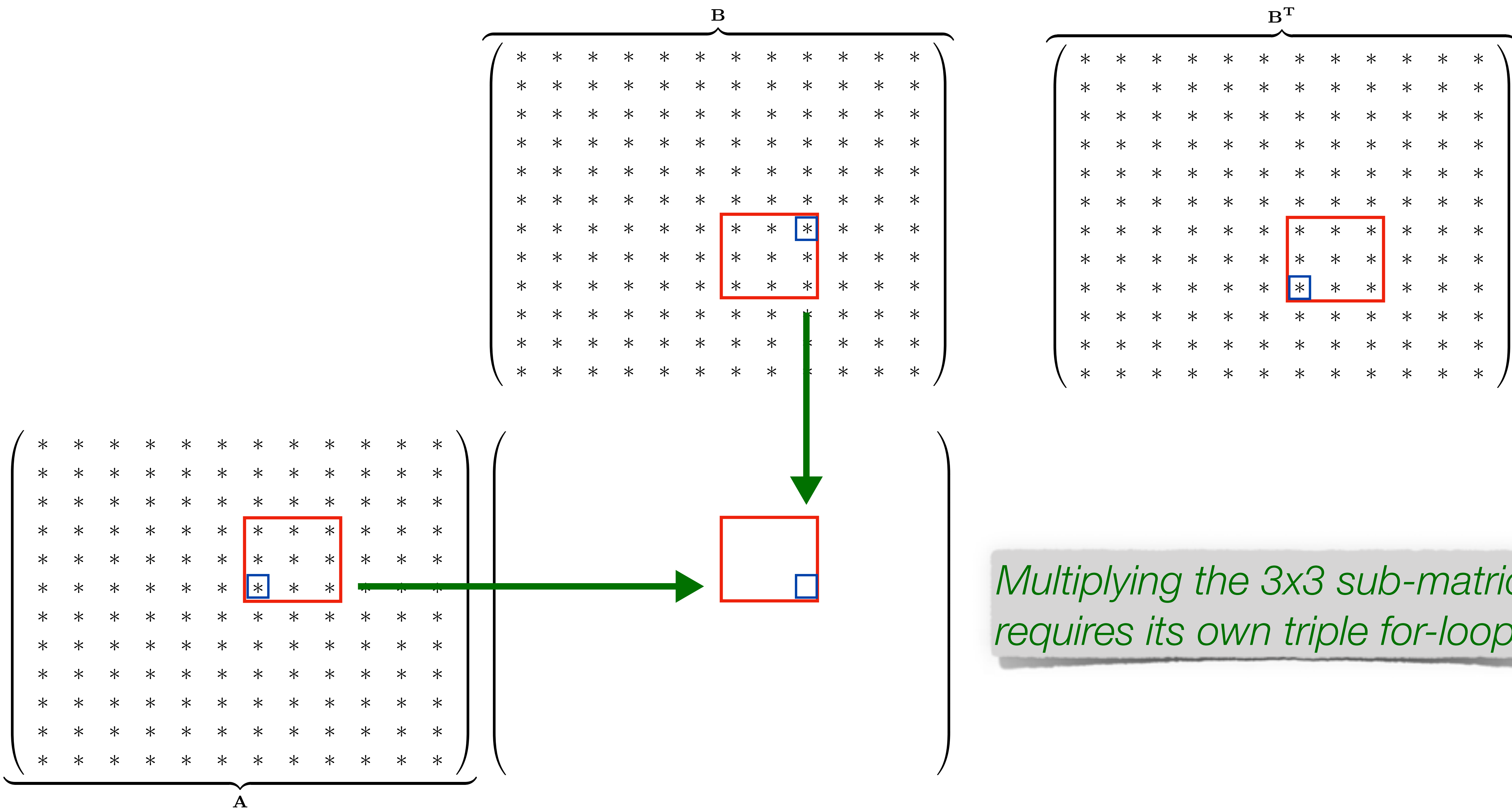
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

Combining blocking & pre-transposed B (or col-major B)



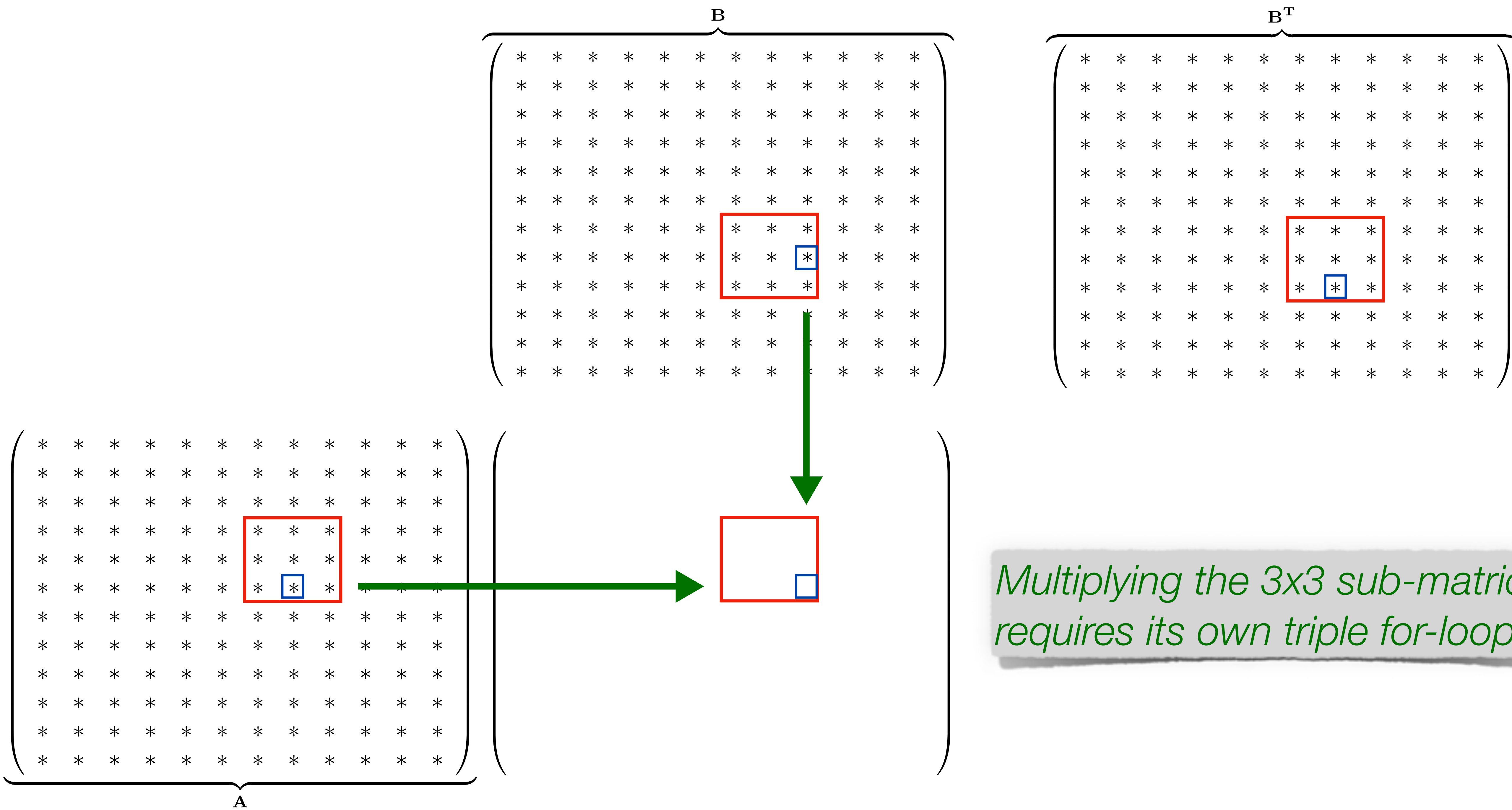
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

Combining blocking & pre-transposed B (or col-major B)



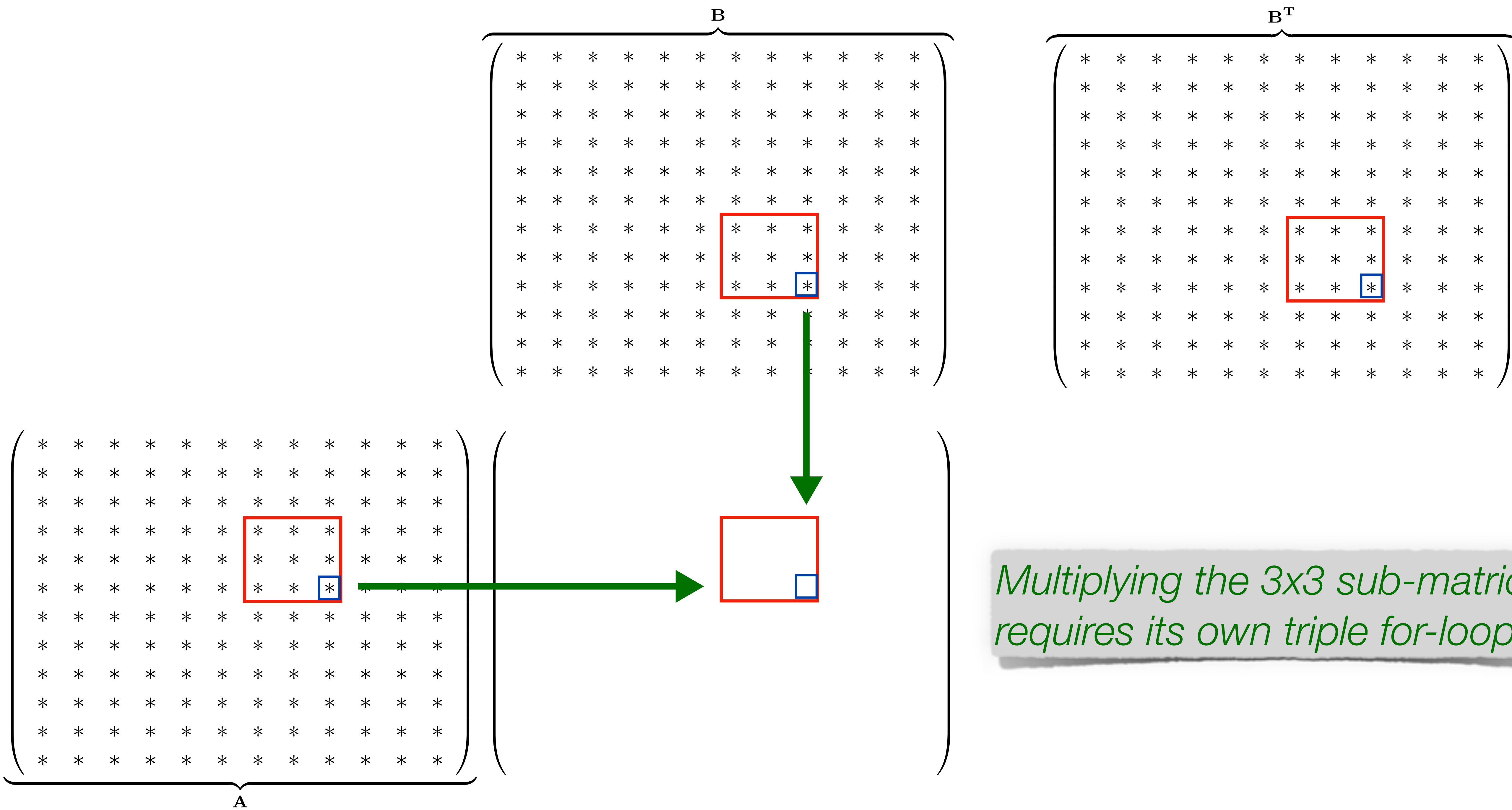
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

Combining blocking & pre-transposed B (or col-major B)



Multiplying the 3x3 sub-matrices requires its own triple for-loop …

Combining blocking & pre-transposed B (or col-major B)



Multiplying the 3x3 sub-matrices requires its own triple for-loop …

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_5

```
#include “MatMatMultiply.h"
[.. .]

void MatMatTransposeMultiply(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]);

    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; 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++)
            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[bj][jj][bk][kk];
}
```

*… and iterate again **within** the blocks, to multiply them together*

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_5

```
#include “MatMatMultiply.h"
[.. .]

void MatMatTransposeMultiply(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]);

    for (int i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; 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++)
            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[bj][jj][bk][kk];
}
```

No parallelization yet!

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_5

```
#include “MatMatMultiply.h"
[.. .]

void MatMatTransposeMultiply(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 =

    for (int i = 0
    for (int j = 0
        C[i][j] =

    for (int bi =
    for (int bj =
        for (int b
            for (i
            for (i
                fo

}
```

At matrix size = 1024

Execution:

```
Transposing second matrix factor ... [Elapsed time : 37.801ms]
Running candidate kernel for correctness test ... [Elapsed time : 860.326ms]
Running reference kernel for correctness test ... [Elapsed time : 3.38718ms]
Discrepancy between two methods : 7.24792e-05
Running kernel for performance run # 1 ... [Elapsed time : 765.428ms]
Running kernel for performance run # 2 ... [Elapsed time : 686.641ms]
Running kernel for performance run # 3 ... [Elapsed time : 687.419ms]
Running kernel for performance run # 4 ... [Elapsed time : 685.17ms]
Running kernel for performance run # 5 ... [Elapsed time : 687.214ms]
Running kernel for performance run # 6 ... [Elapsed time : 686.913ms]
Running kernel for performance run # 7 ... [Elapsed time : 685.123ms]
Running kernel for performance run # 8 ... [Elapsed time : 686.015ms]
[...]
```

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_6

```
[...]
void MatMatTransposeMultiply(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 < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++)
        C[i][j] = 0.;

#pragma omp parallel for
    for (int bi = 0; bi < NBLOCKS; bi++)
    for (int bj = 0; bj < NBLOCKS; bj++)
        for (int bk = 0; bk < NBLOCKS; bk++)
            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++) {
                float partial_result = 0.; // Needed by some compilers for correctness
#pragma omp simd reduction (+:partial_result)
                for (int kk = 0; kk < BLOCK_SIZE; kk++)
                    partial_result += blockA[bi][ii][bk][kk] * blockB[bj][jj][bk][kk];
                blockC[bi][ii][bj][jj] += partial_result;
            }
}
```

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_6

```
[...]
void MatMatTransposeMultiply(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 < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++)
        C[i][j] = 0.;

#pragma omp parallel for
    for (int bi = 0; bi < NBLOCKS; bi++)
    for (int bj = 0; bj < NBLOCKS; bj++)
        for (int bk = 0; bk < NBLOCKS; bk++)
            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++) {
                float partial_result = 0.; // Needed by some compilers for correctness
#pragma omp simd reduction (+:partial_result)
                for (int kk = 0; kk < BLOCK_SIZE; kk++)
                    partial_result += blockA[bi][ii][bk][kk] * blockB[bj][jj][bk][kk];
                blockC[bi][ii][bj][jj] += partial_result;
            }
}
```

*Use multithreading across rows of **A***
*(or rows of blocks of **A**)*

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_6

```
[...]
void MatMatTransposeMultiply(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 < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++)
        C[i][j] = 0.;

#pragma omp parallel for
    for (int bi = 0; bi < NBLOCKS; bi++)
    for (int bj = 0; bj < NBLOCKS; bj++)
        for (int bk = 0; bk < NBLOCKS; bk++)
            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++) {
                float partial_result = 0.; // Needed by some compilers for correctness
#pragma omp simd reduction (+:partial_result)
                for (int kk = 0; kk < BLOCK_SIZE; kk++)
                    partial_result += blockA[bi][ii][bk][kk] * blockB[bj][jj][bk][kk];
                blockC[bi][ii][bj][jj] += partial_result;
            }
}
```

Use SIMD to accelerate the "dot-product-like" reduction in matrix-matrix multiply

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_6

```
[...]
void MatMatTransposeMultiply(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 < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++)
        C[i][j] = 0.;

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

Note: It should have been sufficient to write it like this: (and this does work correctly in most compilers …)

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_6

```
[...]
void MatMatTransposeMultiply(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 < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++)
        C[i][j] = 0.;

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

Note the pattern that suggests SIMD …

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_6

```
[...]
void MatMatTransposeMultiply(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 < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++)
        C[i][j] = 0.;

#pragma omp parallel for
    for (int bi = 0; bi < NBLOCKS; bi++)
    for (int bj = 0; bj < NBLOCKS; bj++)
        for (int bk = 0; bk < NBLOCKS; bk++)
            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++) {
                float partial_result = 0.; // Needed by some compilers for correctness
#pragma omp simd reduction (+:partial_result)
                for (int kk = 0; kk < BLOCK_SIZE; kk++)
                    partial_result += blockA[bi][ii][bk][kk] * blockB[bj][jj][bk][kk];
                blockC[bi][ii][bj][jj] += partial_result;
            }
}
```

... but some versions of gcc/g++ seem to engage in unsafe optimizations (leading to errors) if you don't use an intermediate variable for reduction

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_6

```
[...]
void MatMatTransposeMultiply(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 < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++)
        C[i][j] = 0

#pragma omp parall
    for (int bi =
    for (int bj =
        for (int b
            for (i
            for (i
                fl
#pragma omp simd r
                fo

                bl
            }
}
```

Matrix size 1024 x 1024

Execution:

```
Transposing second matrix factor ... [Elapsed time : 39.6778ms]
Running candidate kernel for correctness test ... [Elapsed time : 23.9182ms]
Running reference kernel for correctness test ... [Elapsed time : 2.6098ms]
Discrepancy between two methods : 3.8147e-05
Running kernel for performance run # 1 ... [Elapsed time : 14.682ms]
Running kernel for performance run # 2 ... [Elapsed time : 14.4771ms]
Running kernel for performance run # 3 ... [Elapsed time : 14.4331ms]
Running kernel for performance run # 4 ... [Elapsed time : 14.7571ms]
Running kernel for performance run # 5 ... [Elapsed time : 14.6737ms]
Running kernel for performance run # 6 ... [Elapsed time : 14.5883ms]
Running kernel for performance run # 7 ... [Elapsed time : 14.6881ms]
Running kernel for performance run # 8 ... [Elapsed time : 13.9368ms]
[...]
```

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_6

```
[...]
void MatMatTransposeMultiply(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 < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++)
        C[i][j] = 0.;

#pragma omp parall
    for (int bi =
    for (int bj =
        for (int b
            for (i
            for (i
                fl
#pragma omp simd r
                fo

                bl
            }
}
```

Matrix size 2048 x 2048

Execution:

```
Transposing second matrix factor ... [Elapsed time : 40.4148ms]
Running candidate kernel for correctness test ... [Elapsed time : 116.462ms]
Running reference kernel for correctness test ... [Elapsed time : 16.2658ms]
Discrepancy between two methods : 4.57764e-05
Running kernel for performance run # 1 ... [Elapsed time : 105.37ms]
Running kernel for performance run # 2 ... [Elapsed time : 104.903ms]
Running kernel for performance run # 3 ... [Elapsed time : 104.987ms]
Running kernel for performance run # 4 ... [Elapsed time : 108.066ms]
Running kernel for performance run # 5 ... [Elapsed time : 110.45ms]
Running kernel for performance run # 6 ... [Elapsed time : 111.708ms]
Running kernel for performance run # 7 ... [Elapsed time : 110.166ms]
Running kernel for performance run # 8 ... [Elapsed time : 109.819ms]
[...]
```

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
    );
}
```

(compare with MKL)

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]
```