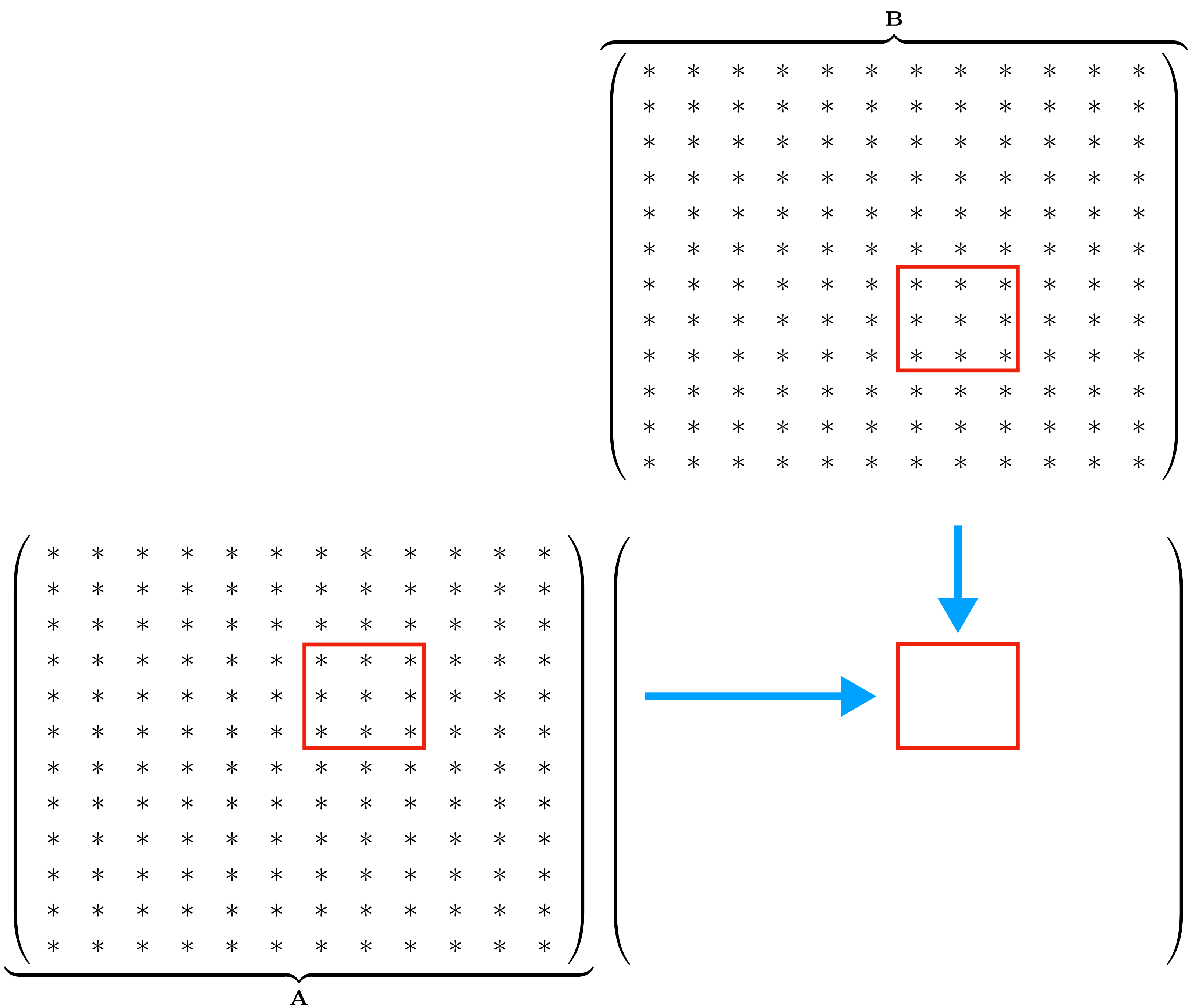
Dense Matrix Computations

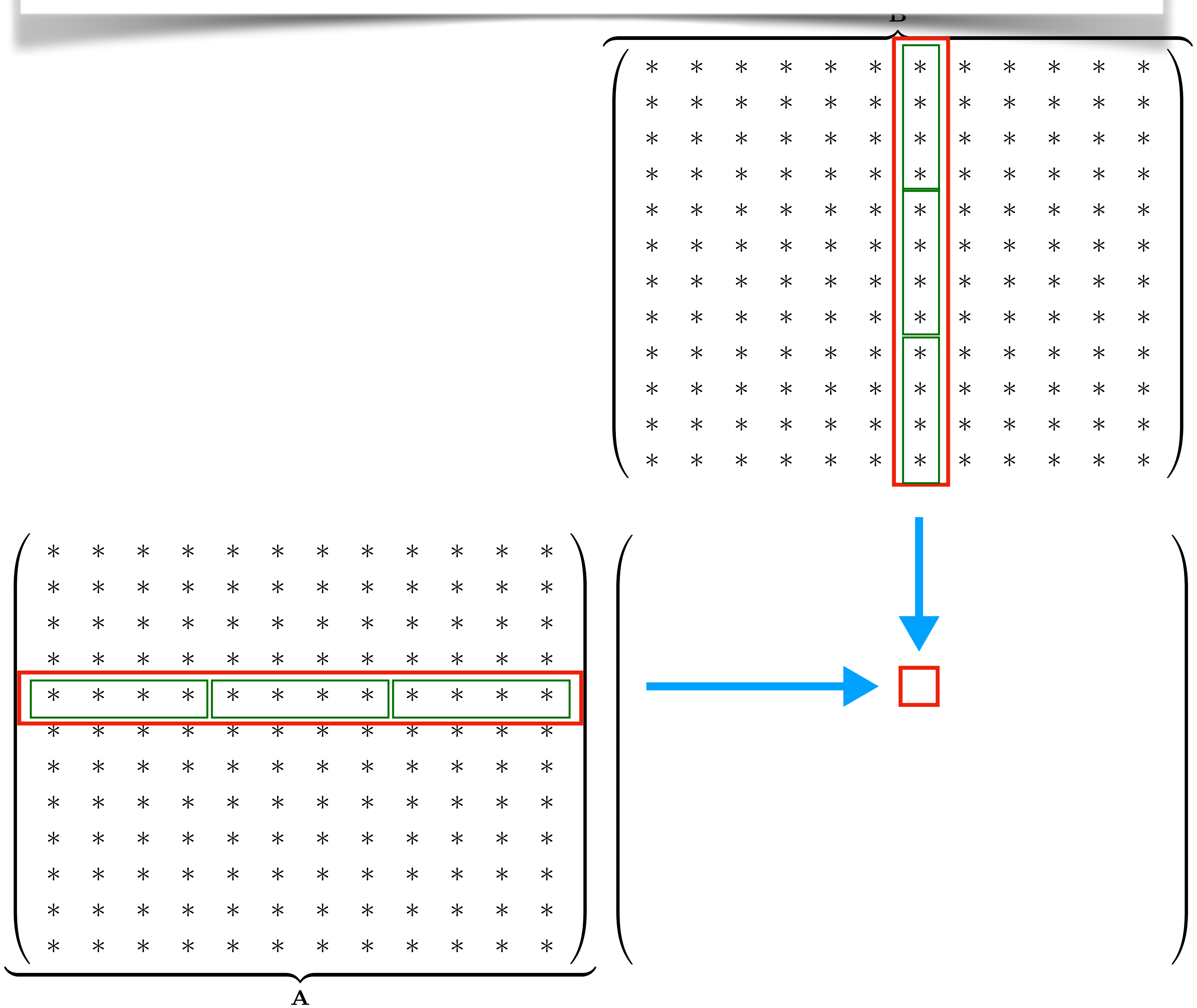
Optimizing GEMM Operations in OpenMP (Part#2)

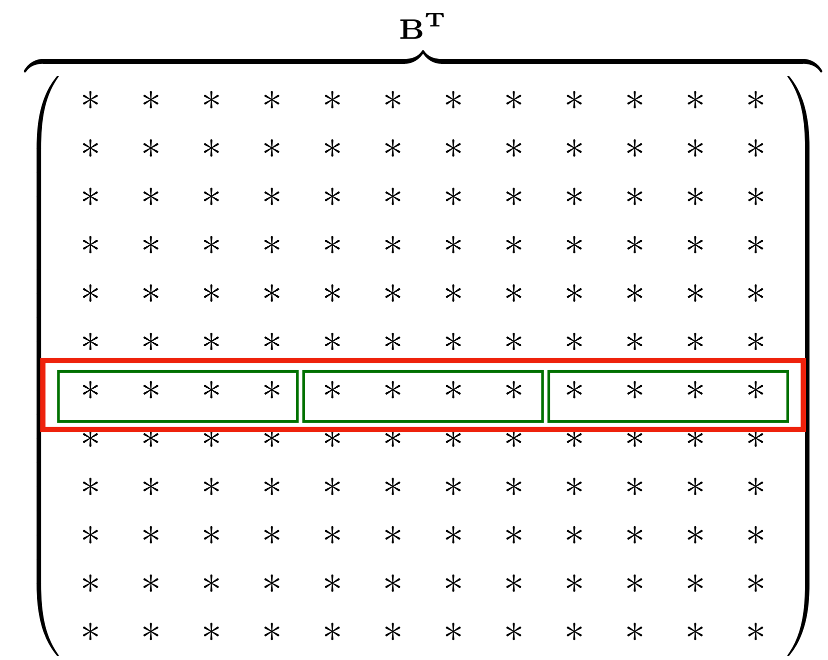
Promising leads: Blocking?



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

Promising leads: Use transpose?

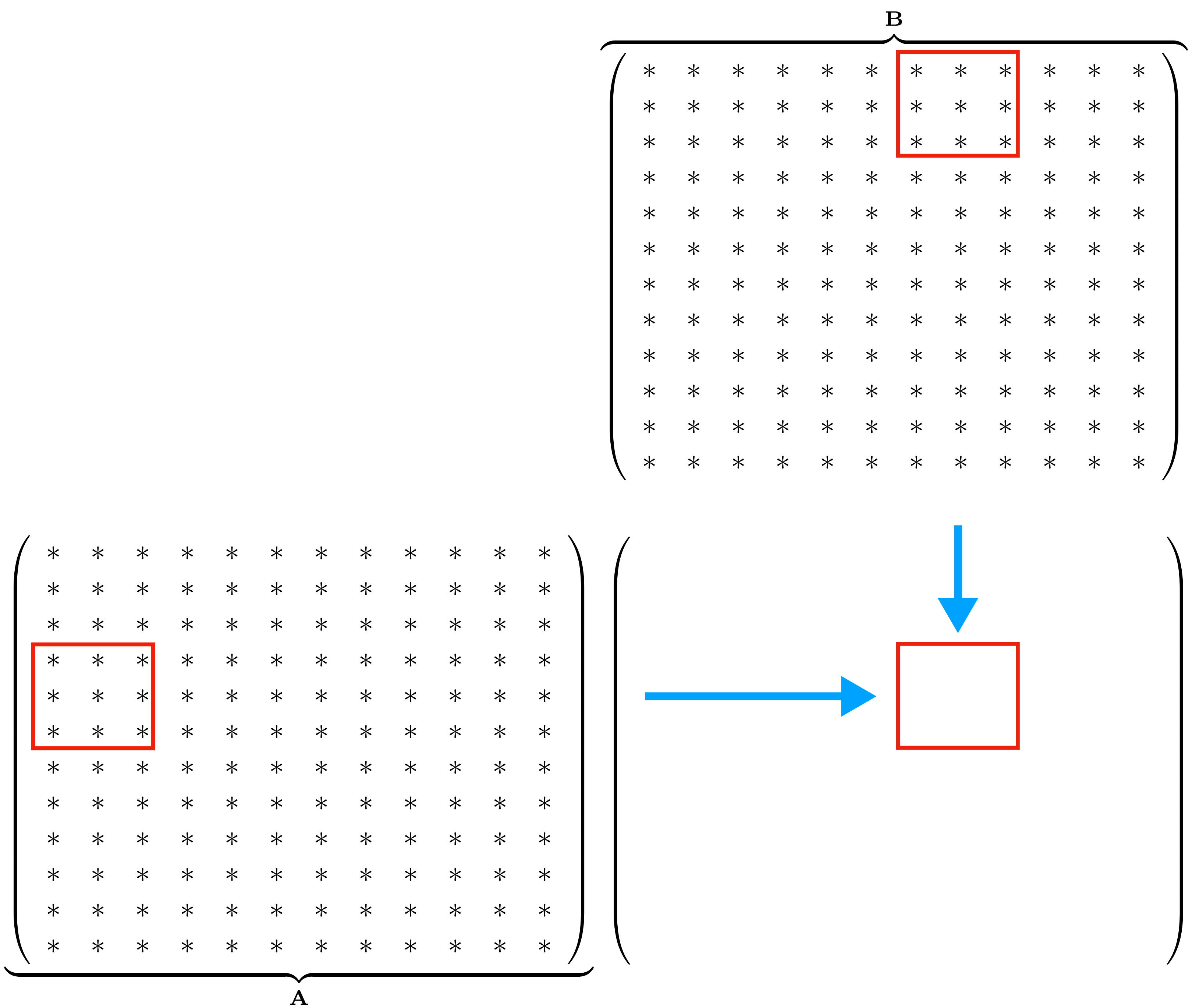


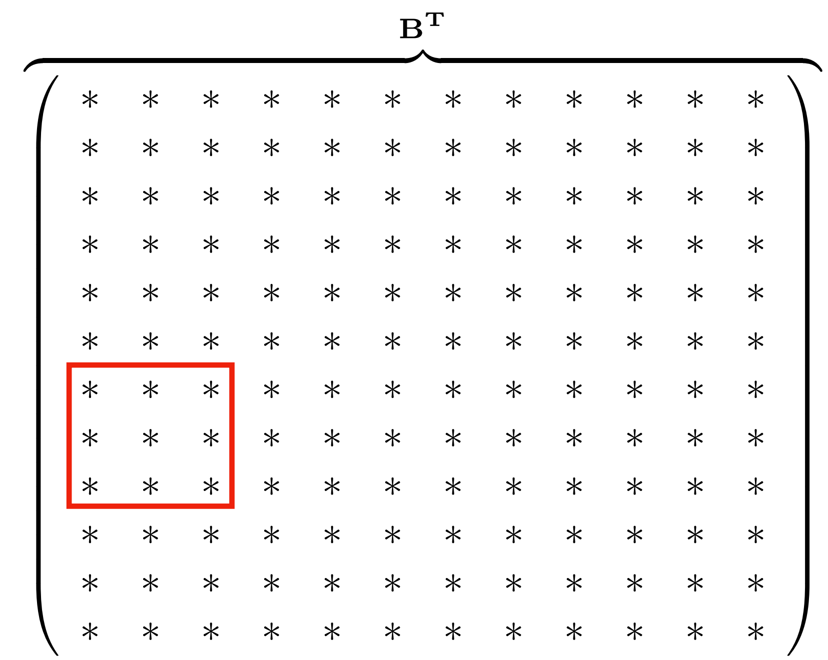


for $i = 1 \ldots N$
    for $j = 1 \ldots N$
        $C_{ij} \leftarrow 0$
        for $k = 1 \ldots N$
            $C_{ij} \leftarrow C_{ij} + A_{ik}[B^T]_{jk}$

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

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

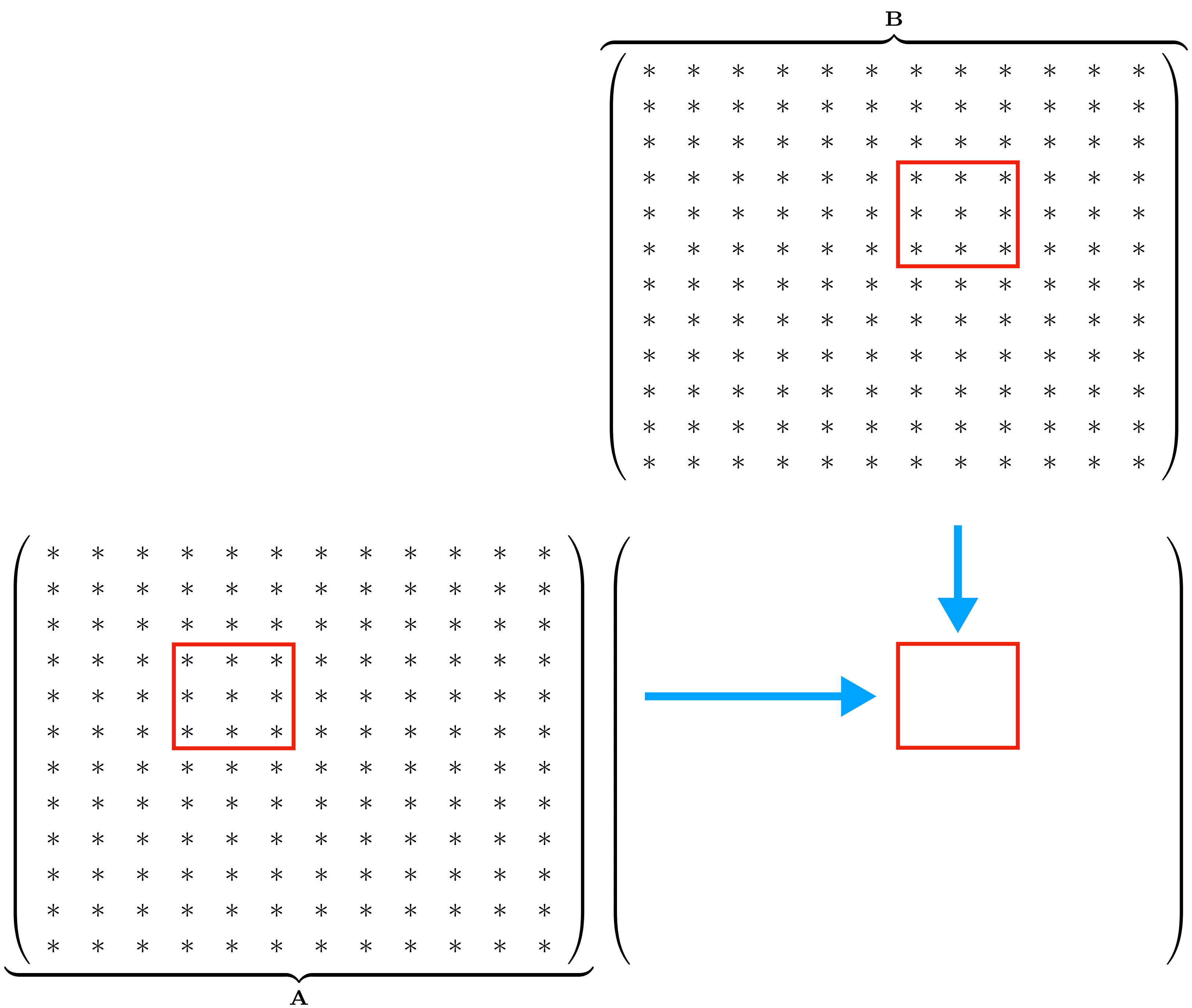




$$
\begin{aligned}
&\texttt{for}\ \ i = 1 \ldots N \\
&\quad \texttt{for}\ \ j = 1 \ldots N \\
&\quad\quad C_{ij} \leftarrow 0 \\
&\quad\quad \texttt{for}\ \ k = 1 \ldots N \\
&\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} [B^T]_{jk}
\end{aligned}
$$

***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 $j = 1 \ldots N$

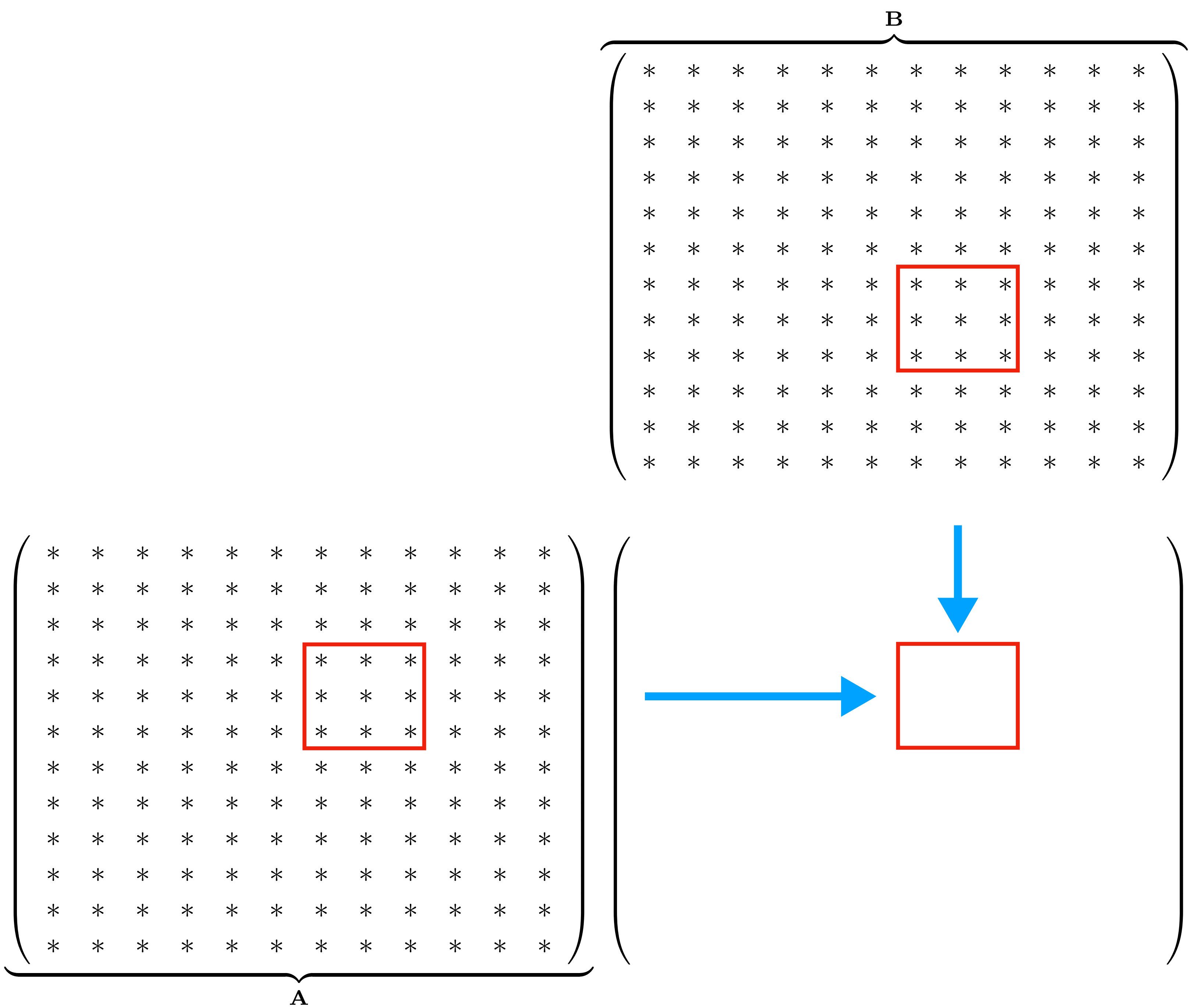
    $C_{ij} \leftarrow 0$

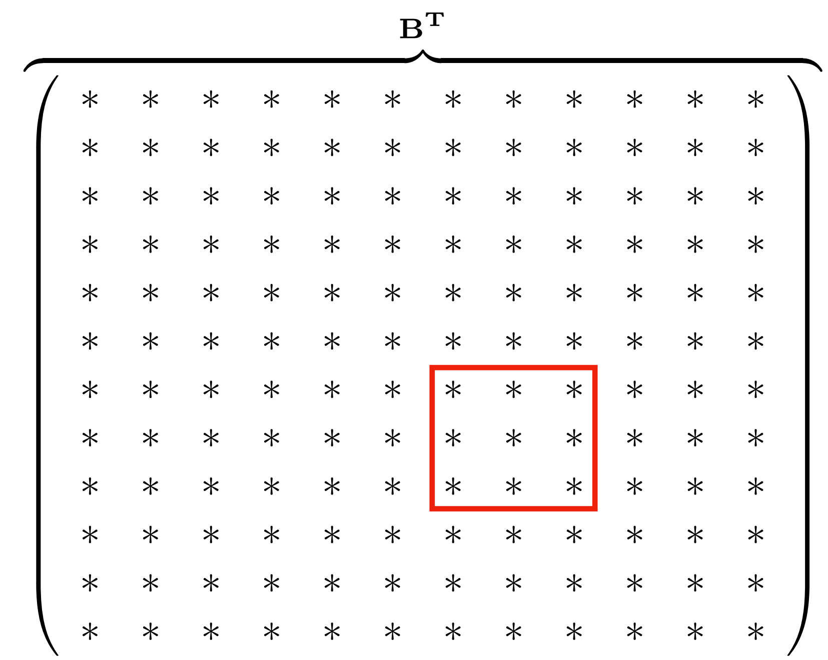
    for $k = 1 \ldots N$

      $C_{ij} \leftarrow C_{ij} + A_{ik} [B^T]_{jk}$

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

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

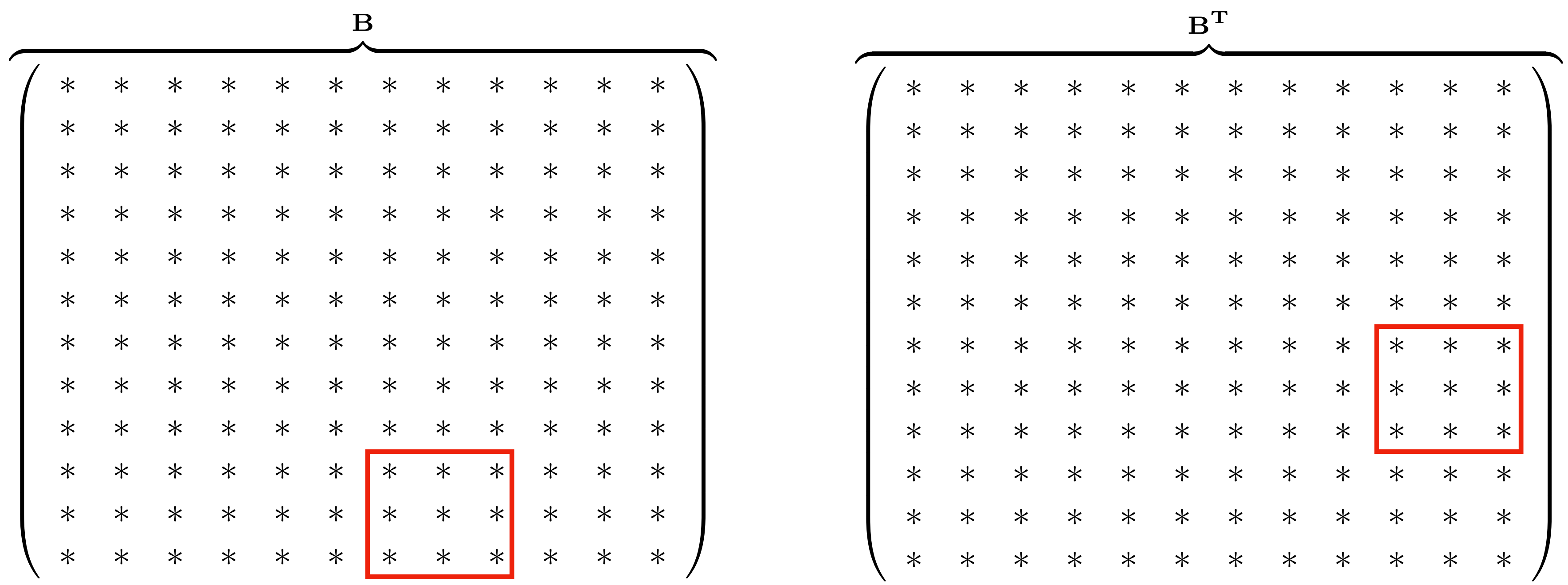


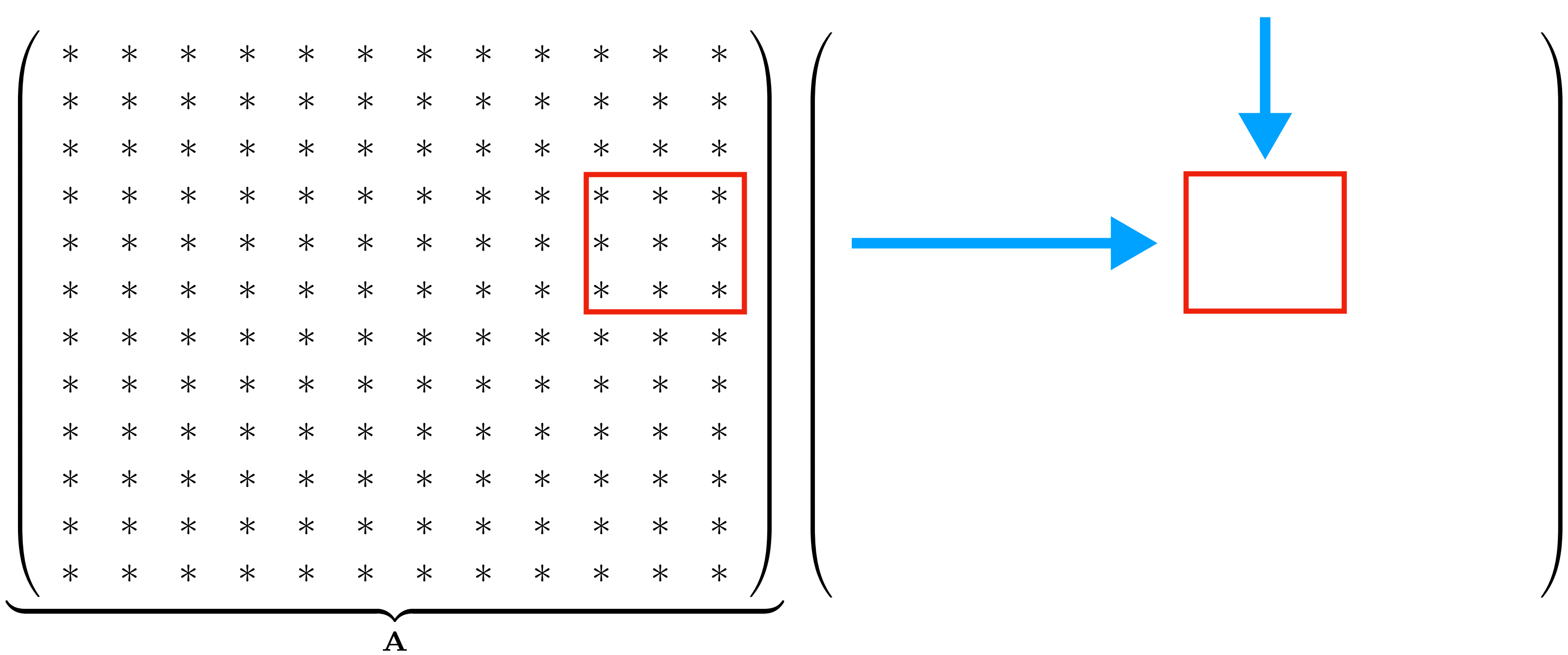


$$
\begin{array}{l}
\texttt{for } i = 1 \ldots N \\
\quad \texttt{for } j = 1 \ldots N \\
\quad\quad C_{ij} \leftarrow 0 \\
\quad\quad \texttt{for } k = 1 \ldots N \\
\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} [B^T]_{jk}
\end{array}
$$

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

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

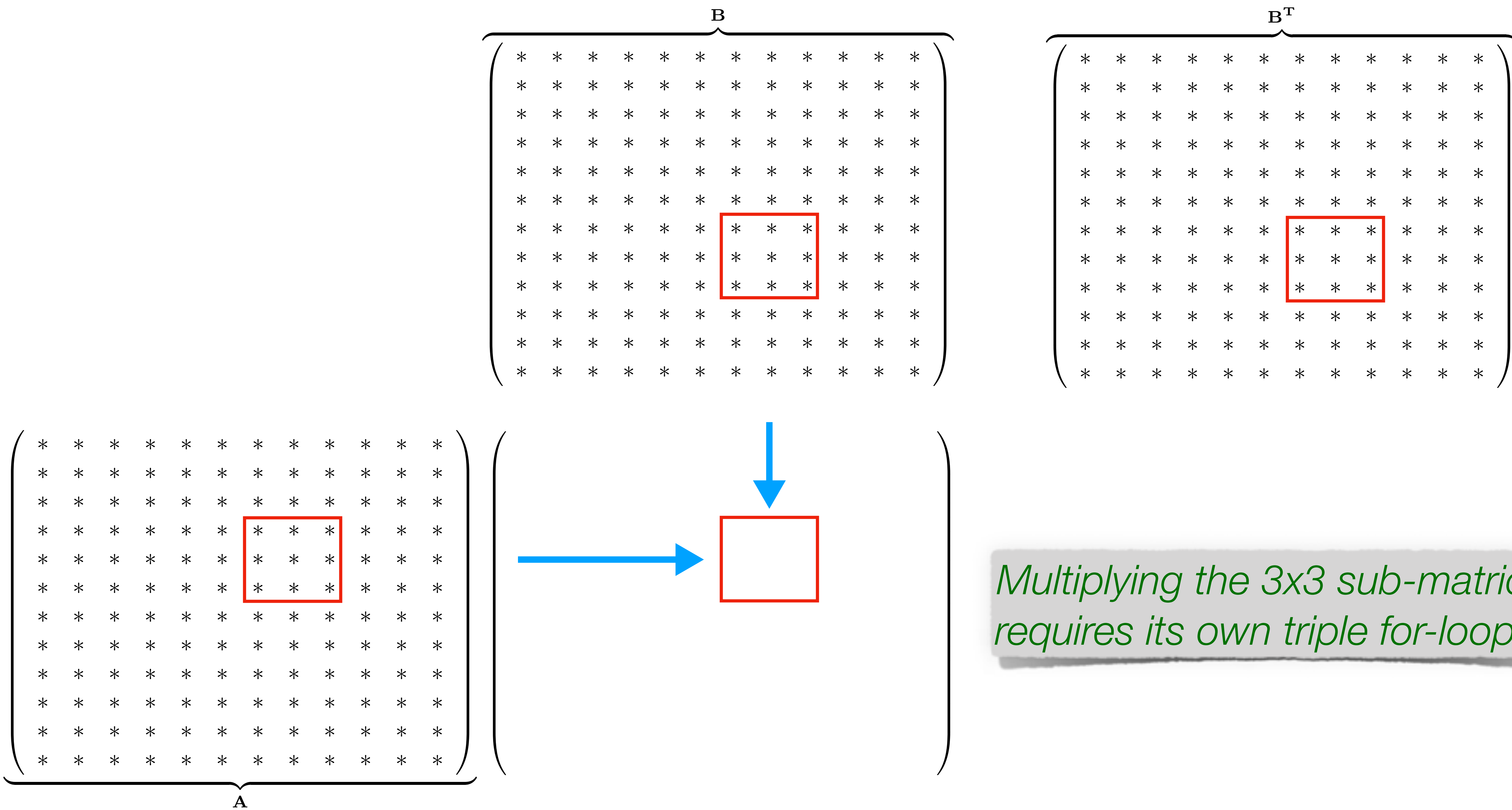




for $i = 1 \ldots N$
    for $j = 1 \ldots N$
        $C_{ij} \leftarrow 0$
        for $k = 1 \ldots N$
            $C_{ij} \leftarrow C_{ij} + A_{ik} [B^T]_{jk}$

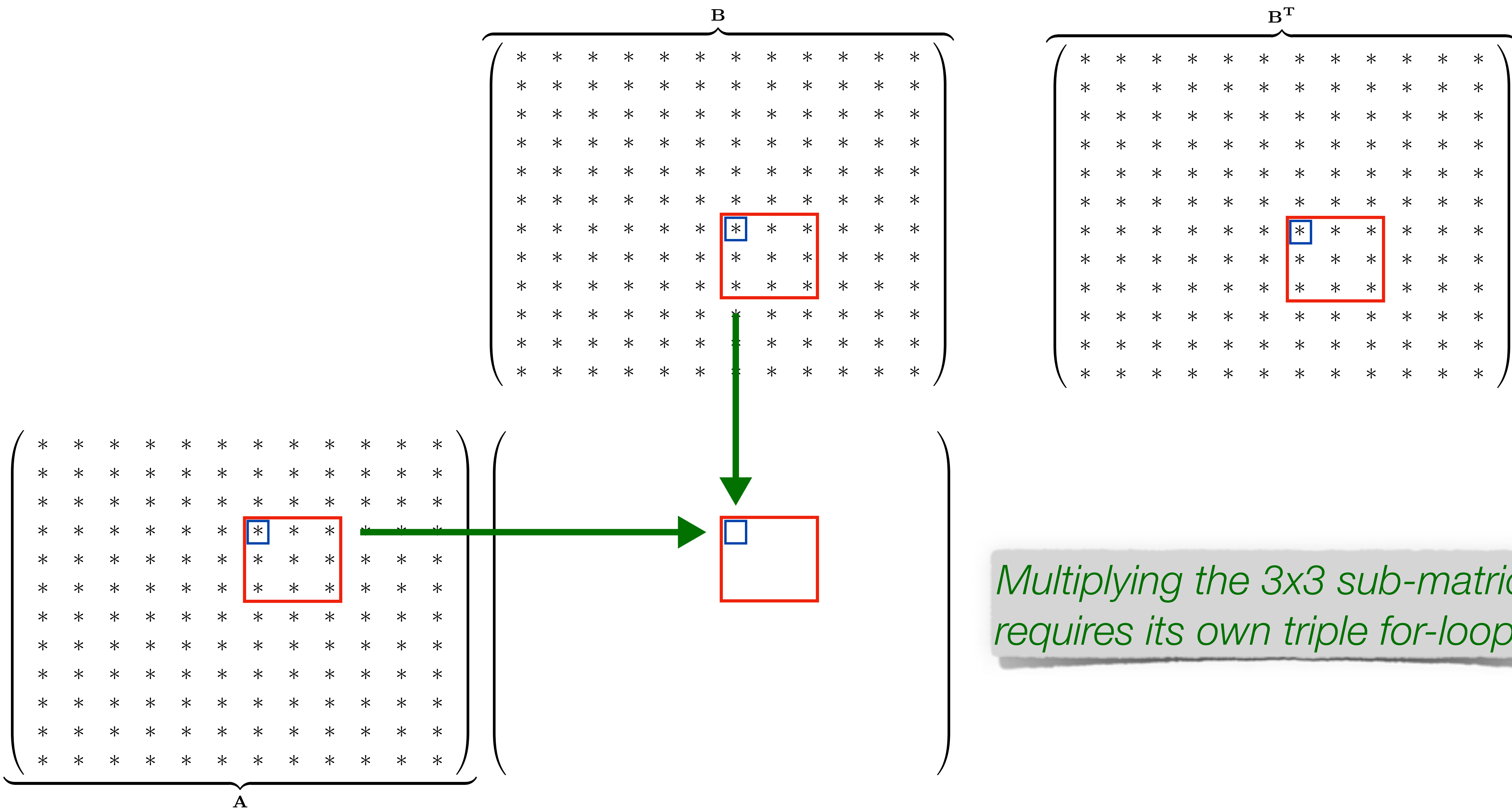
***C_{ij}*, *A_{ik}* and *$B^T{}_{jk}$* represent block 3x3 sub-matrices**

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



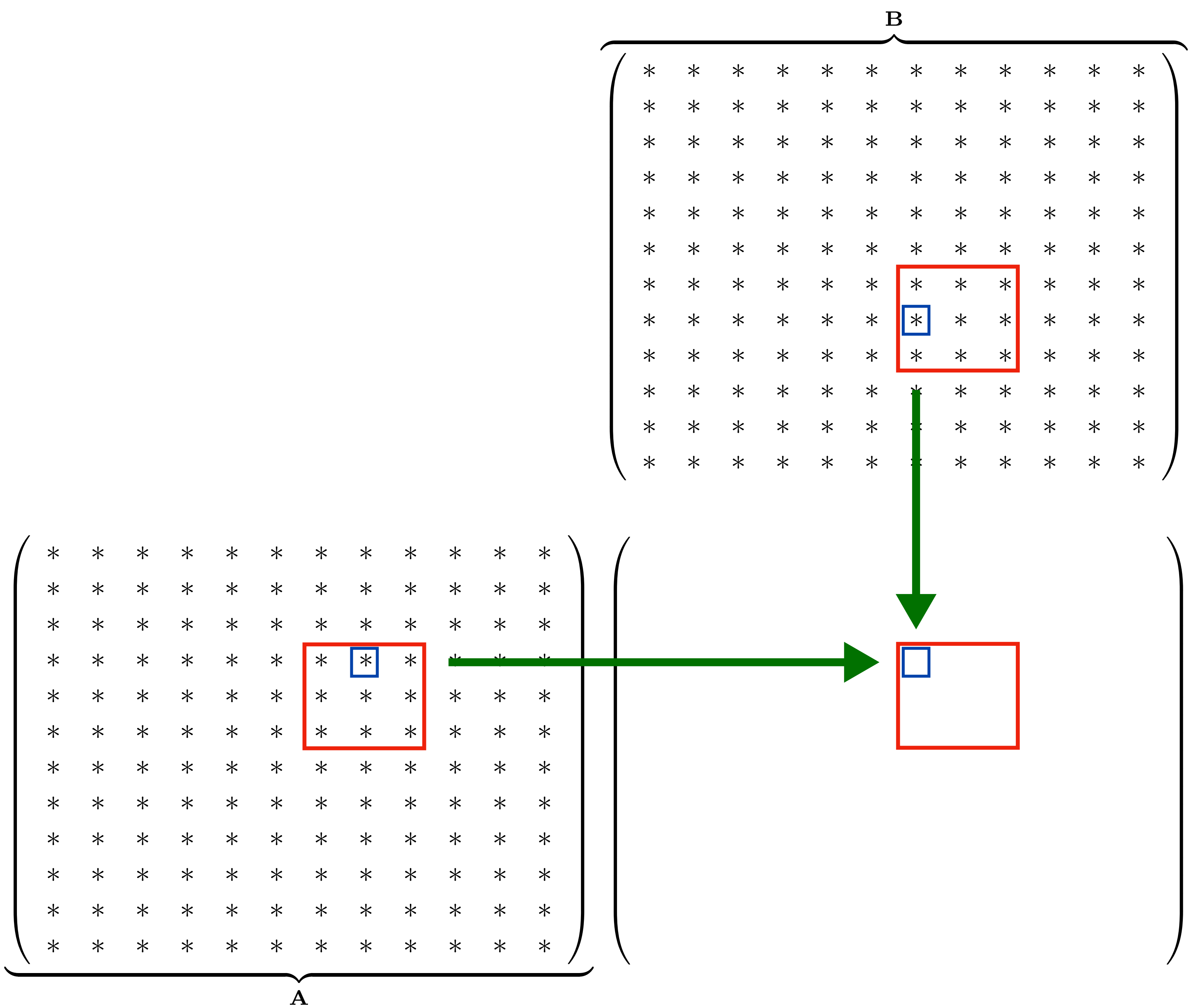
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

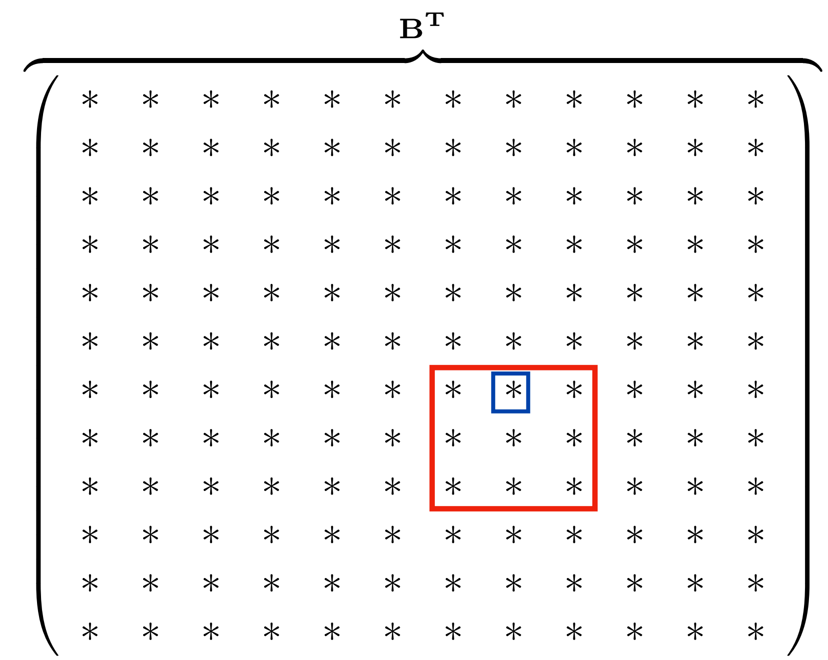
Combining blocking & pre-transposed B (or col-major B)



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

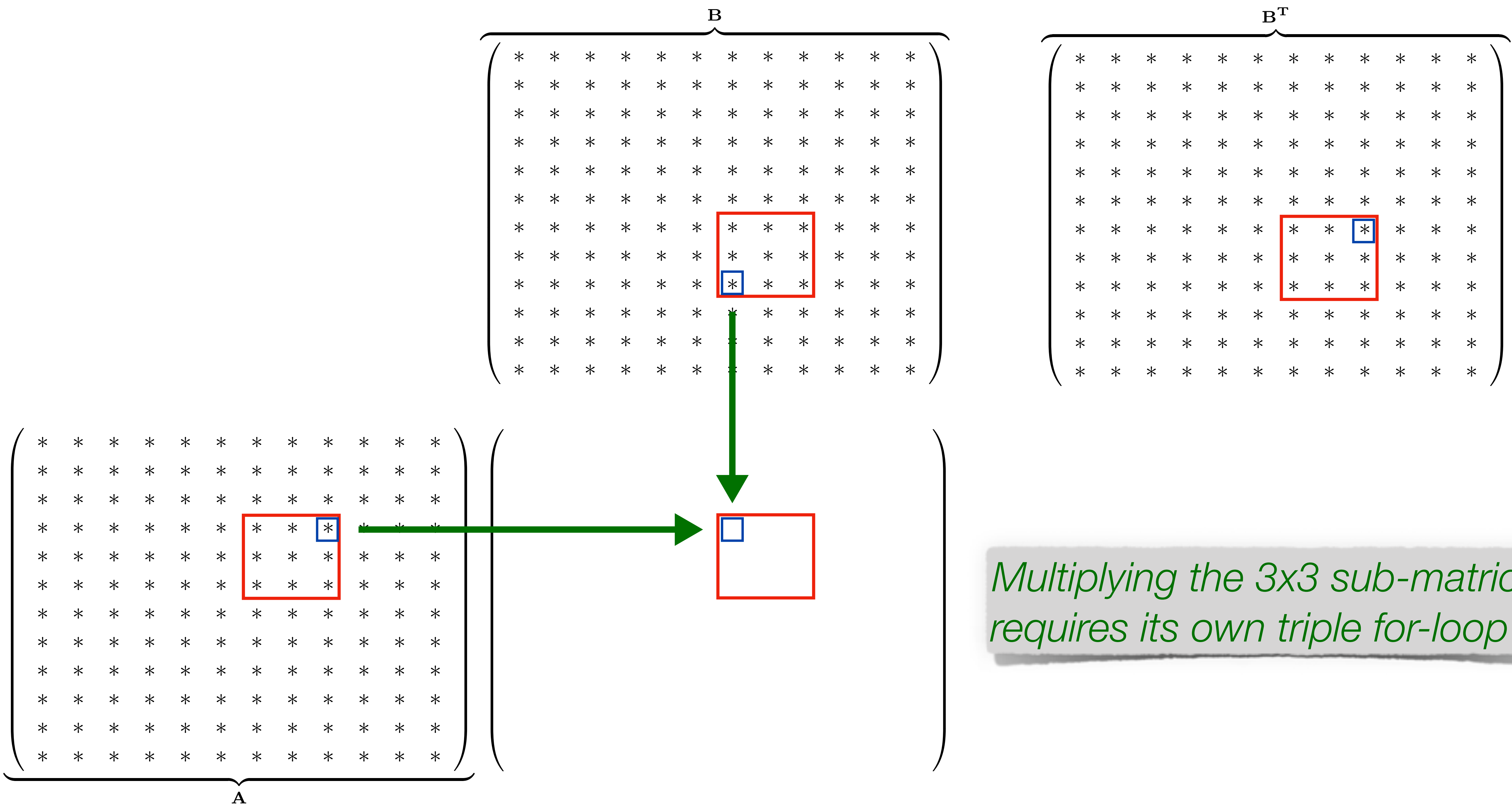
Combining blocking & pre-transposed B (or col-major B)





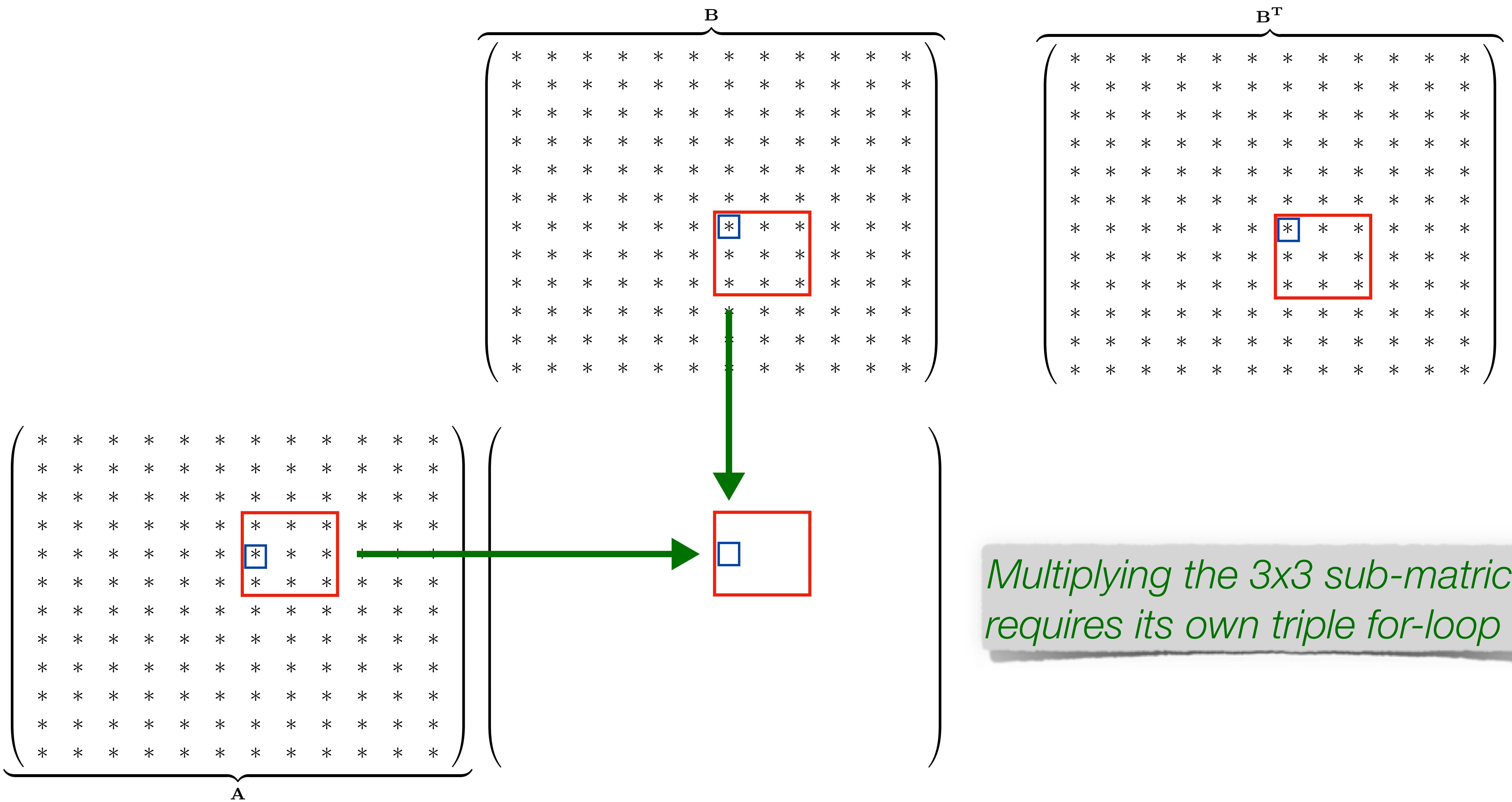
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

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



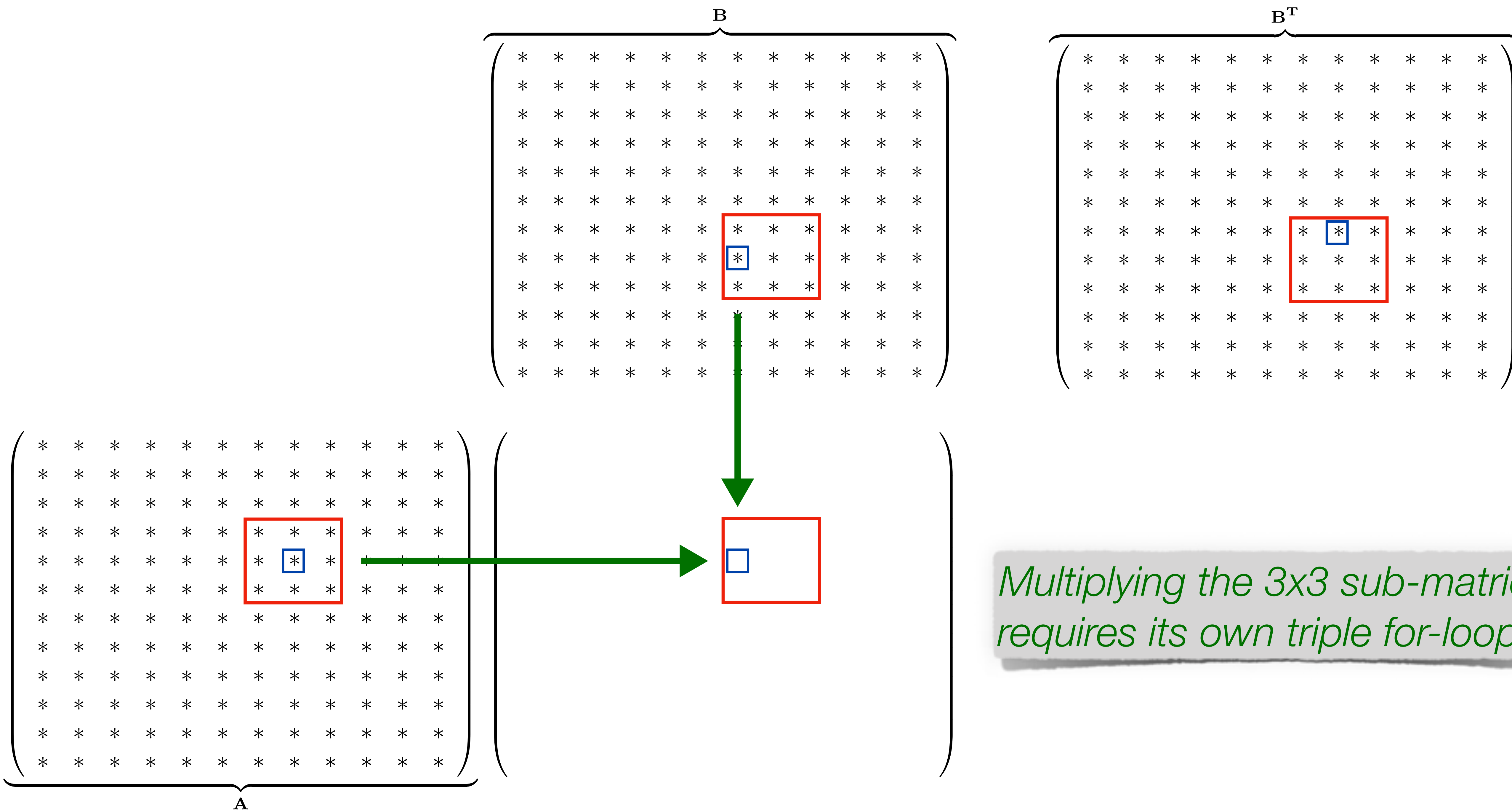
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

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



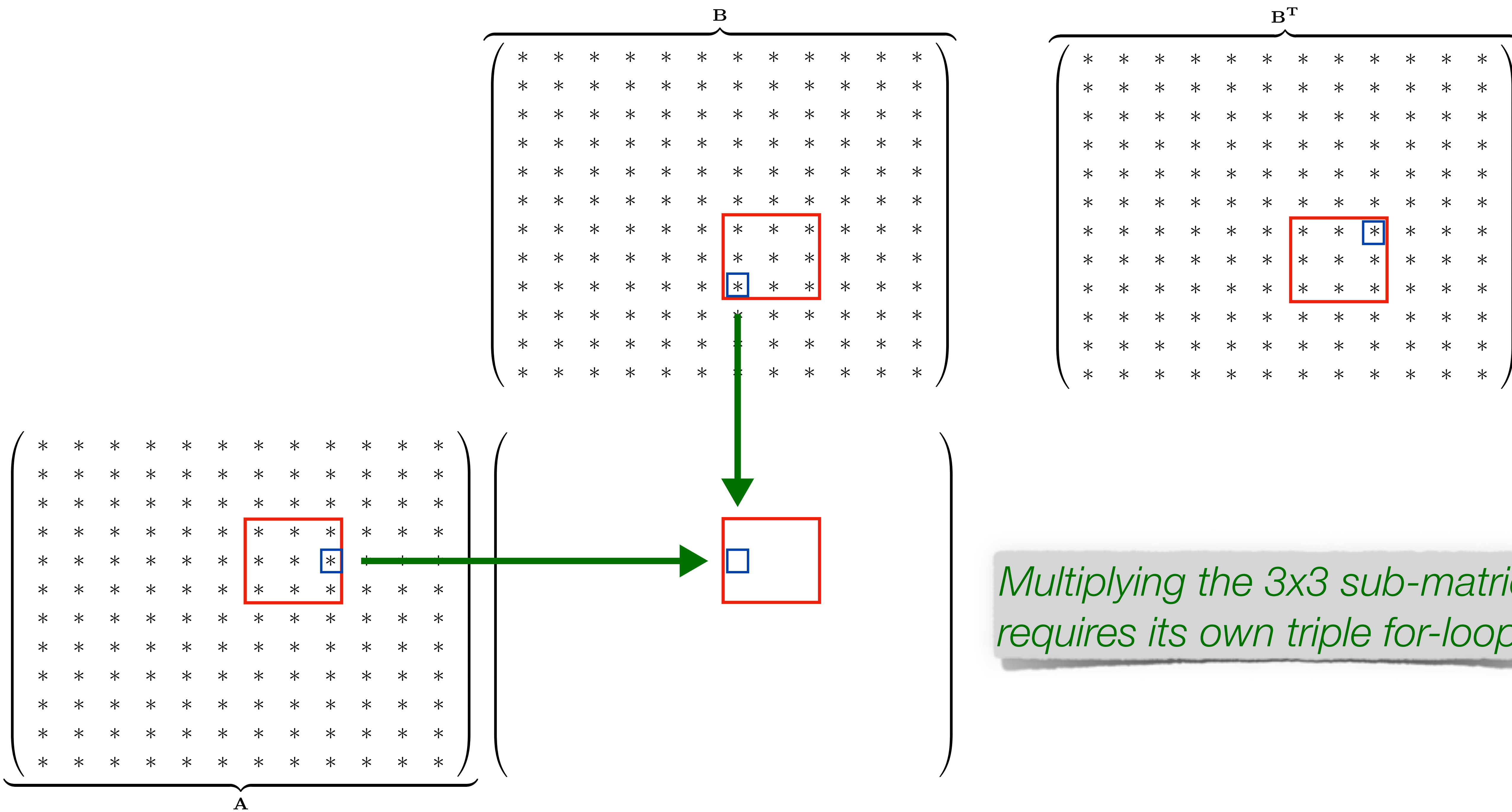
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

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



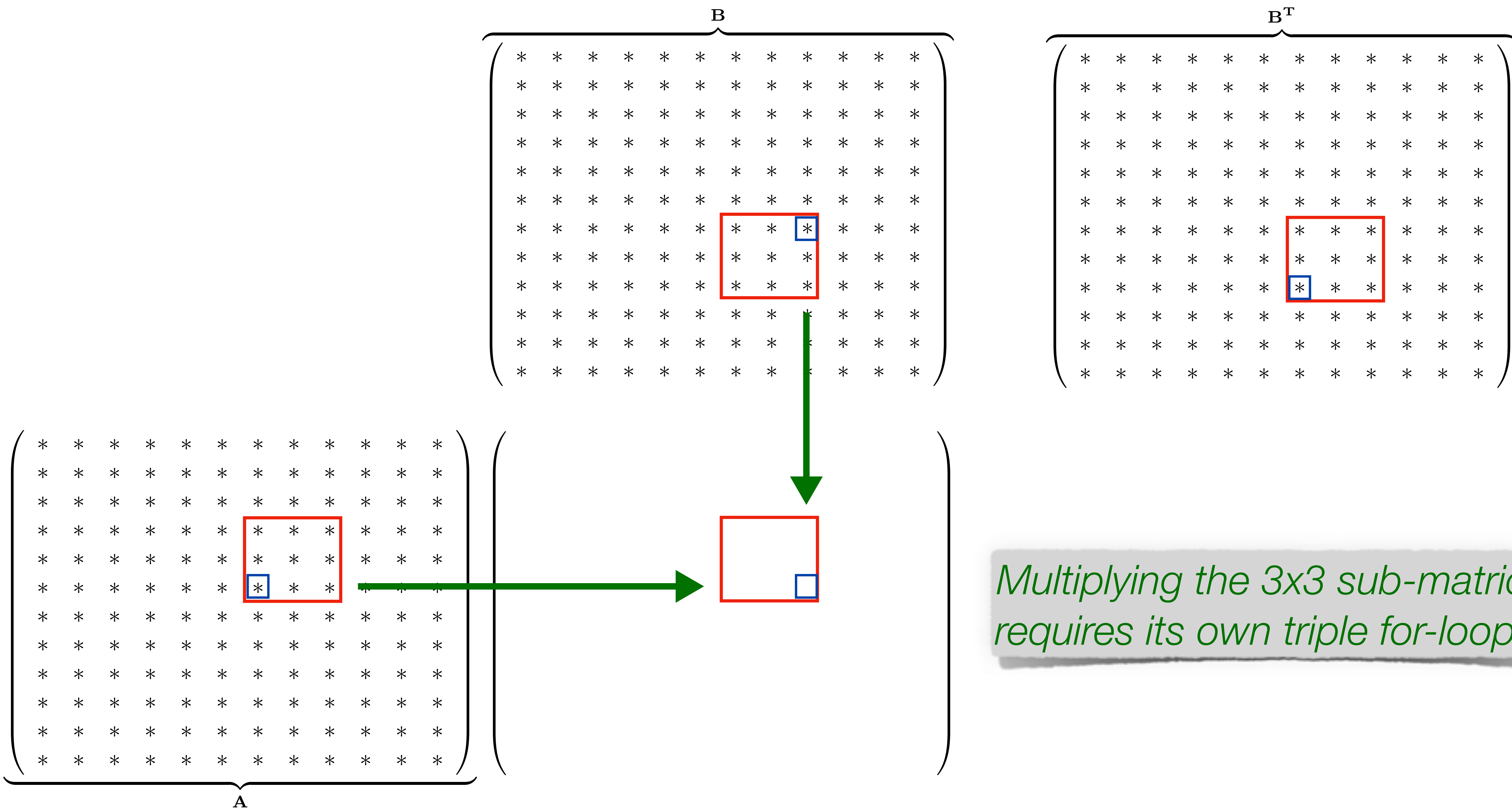
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

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



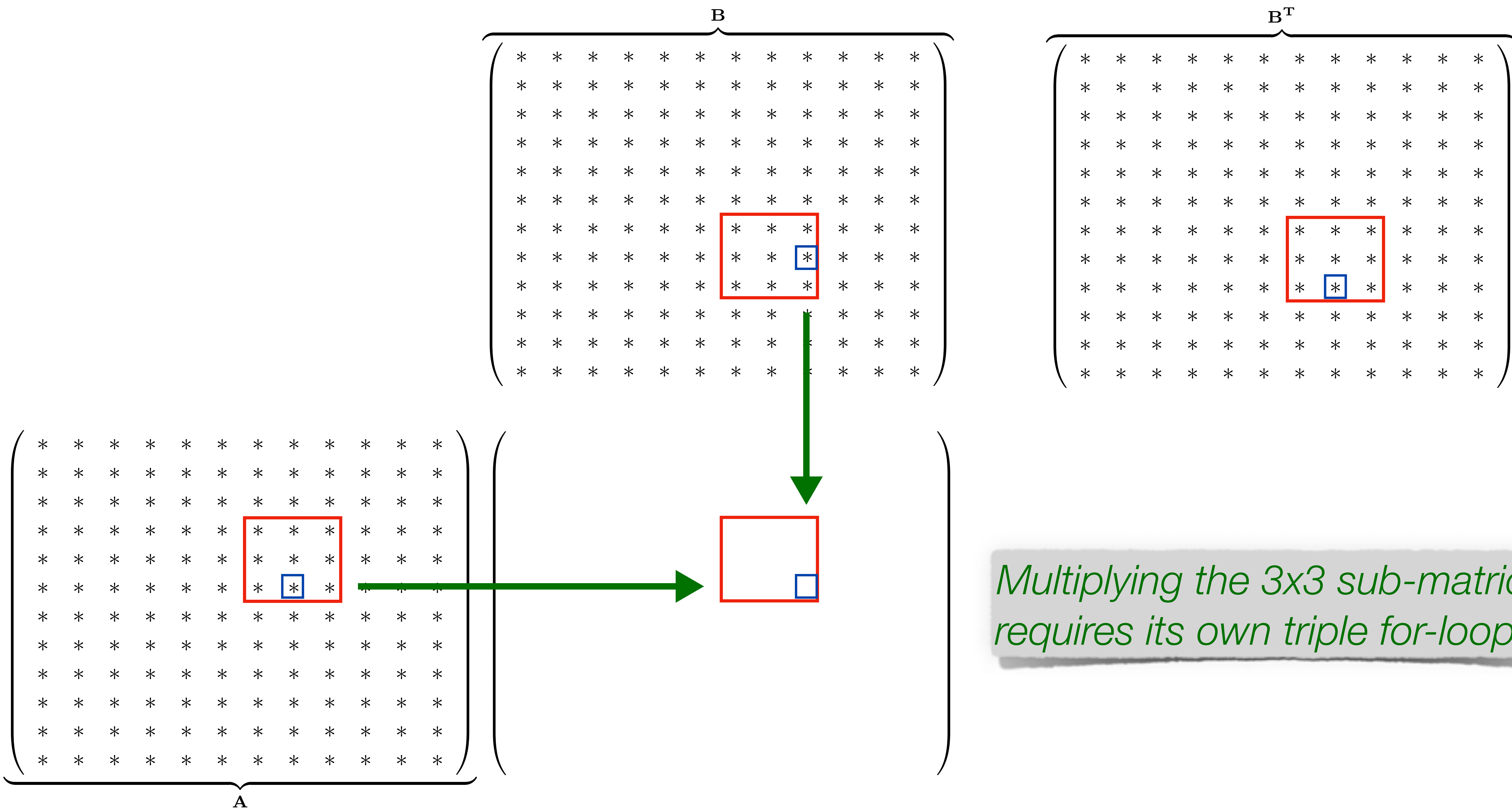
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

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



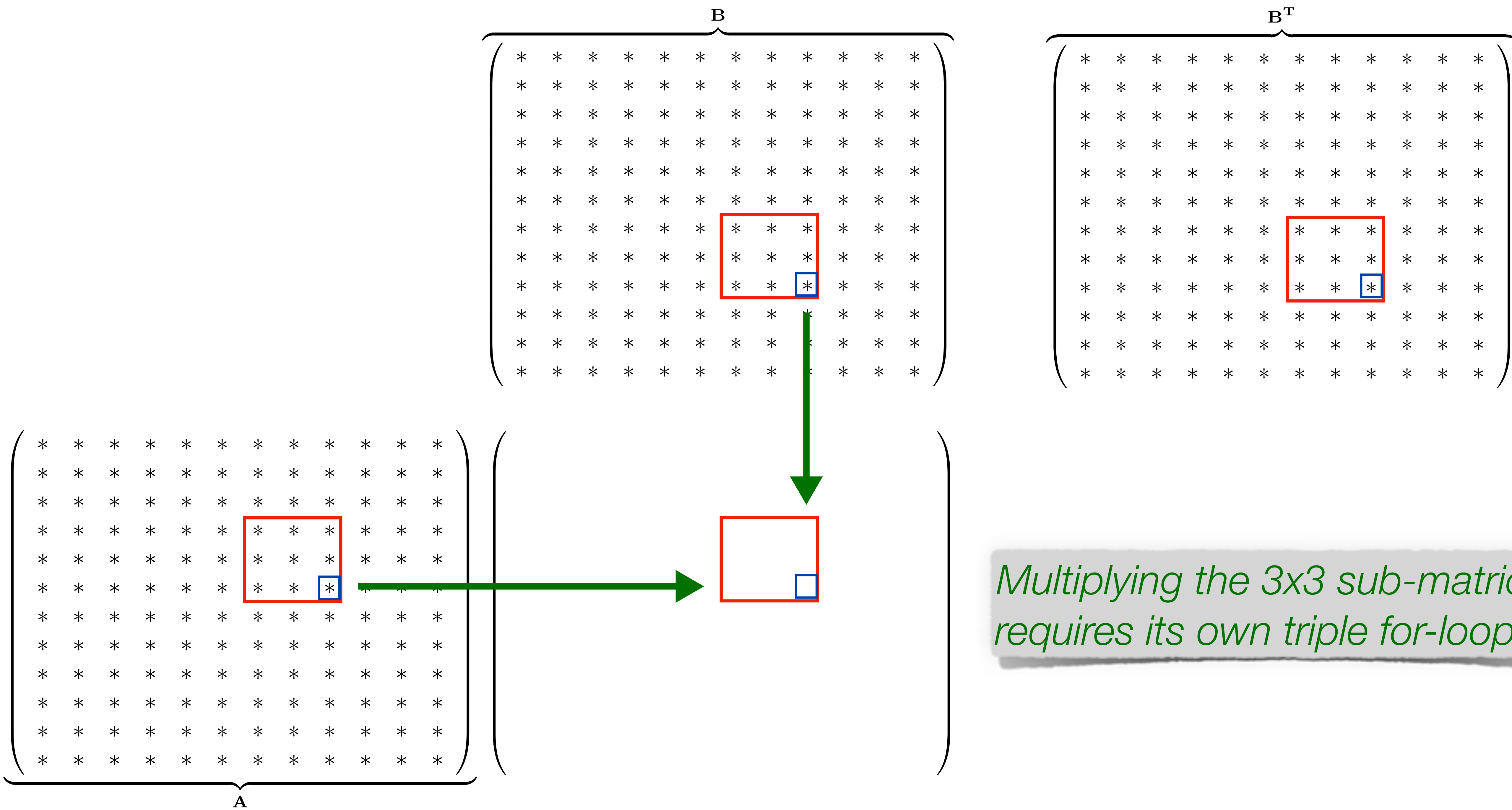
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

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)

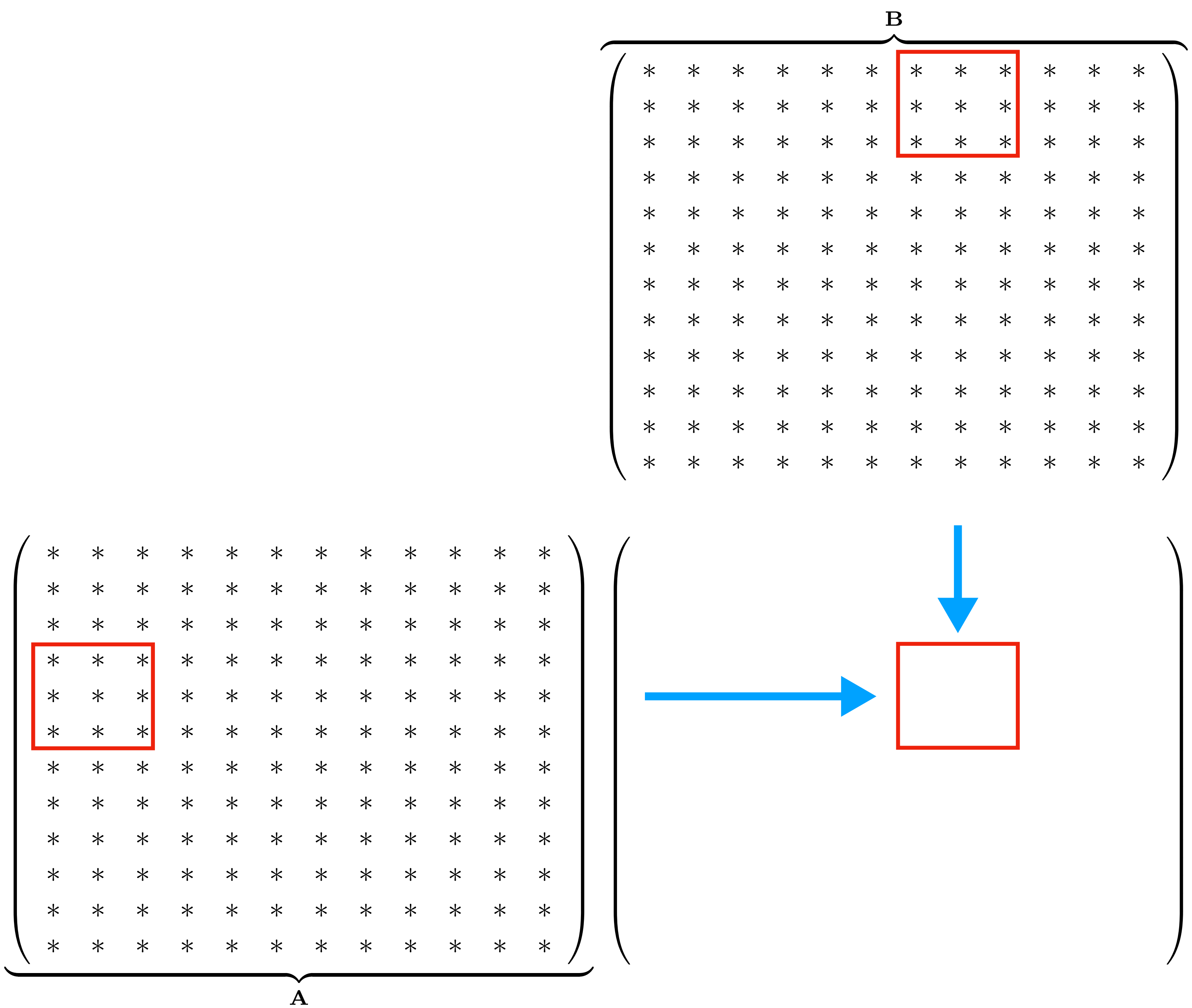
GEMM routine (MatMatMultiply.cpp)

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** …*

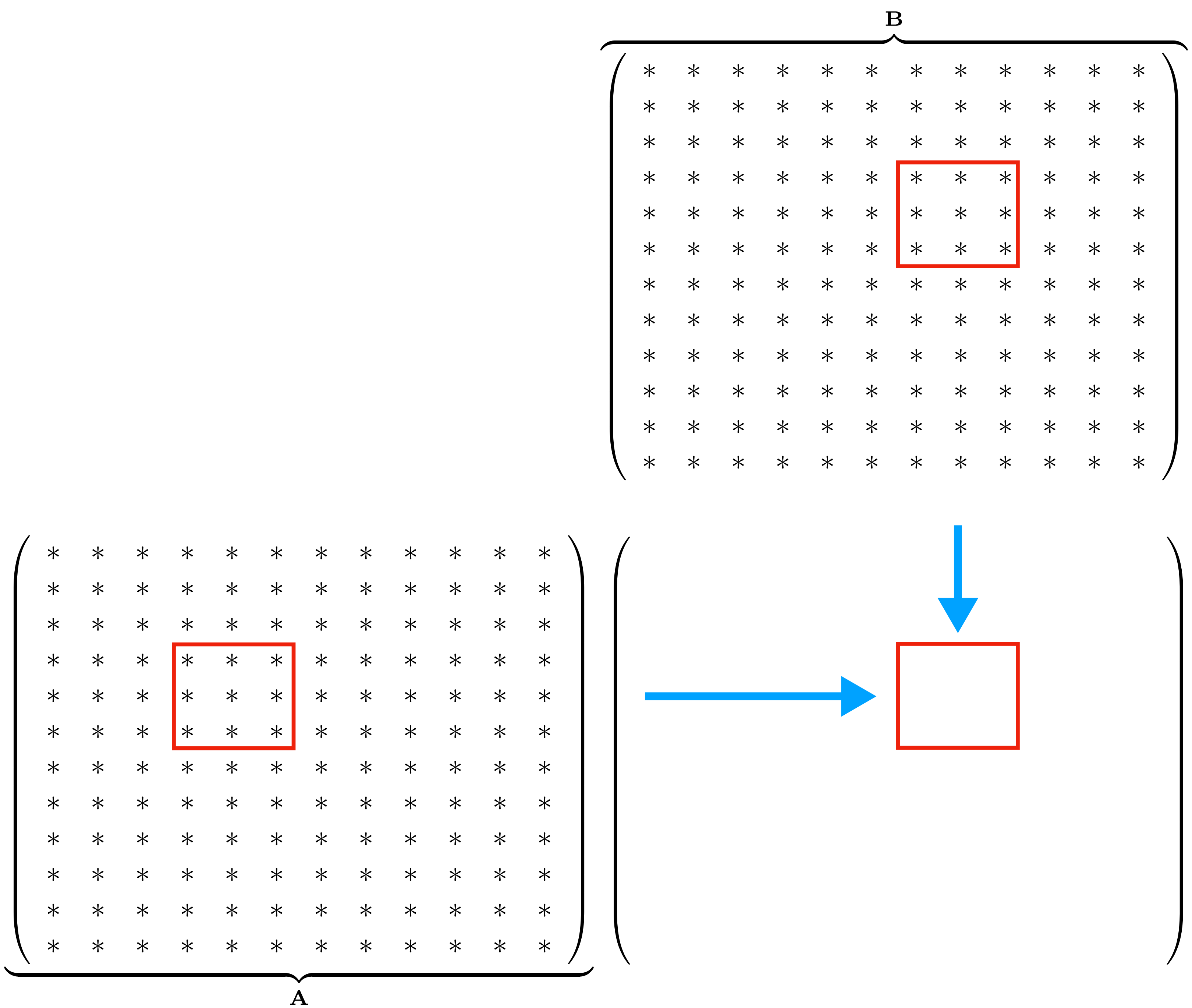
Combining blocking & pre-transposed B (or col-major B)

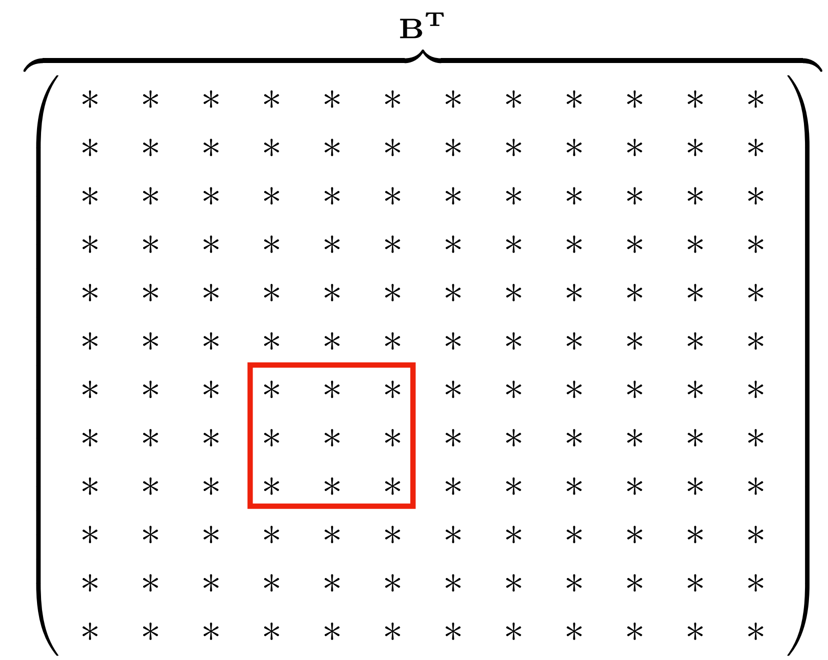


$$
\begin{array}{l}
\texttt{for } i = 1 \ldots N \\
\quad \texttt{for } j = 1 \ldots N \\
\quad\quad C_{ij} \leftarrow 0 \\
\quad\quad \texttt{for } k = 1 \ldots N \\
\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} [B^T]_{jk}
\end{array}
$$

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

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

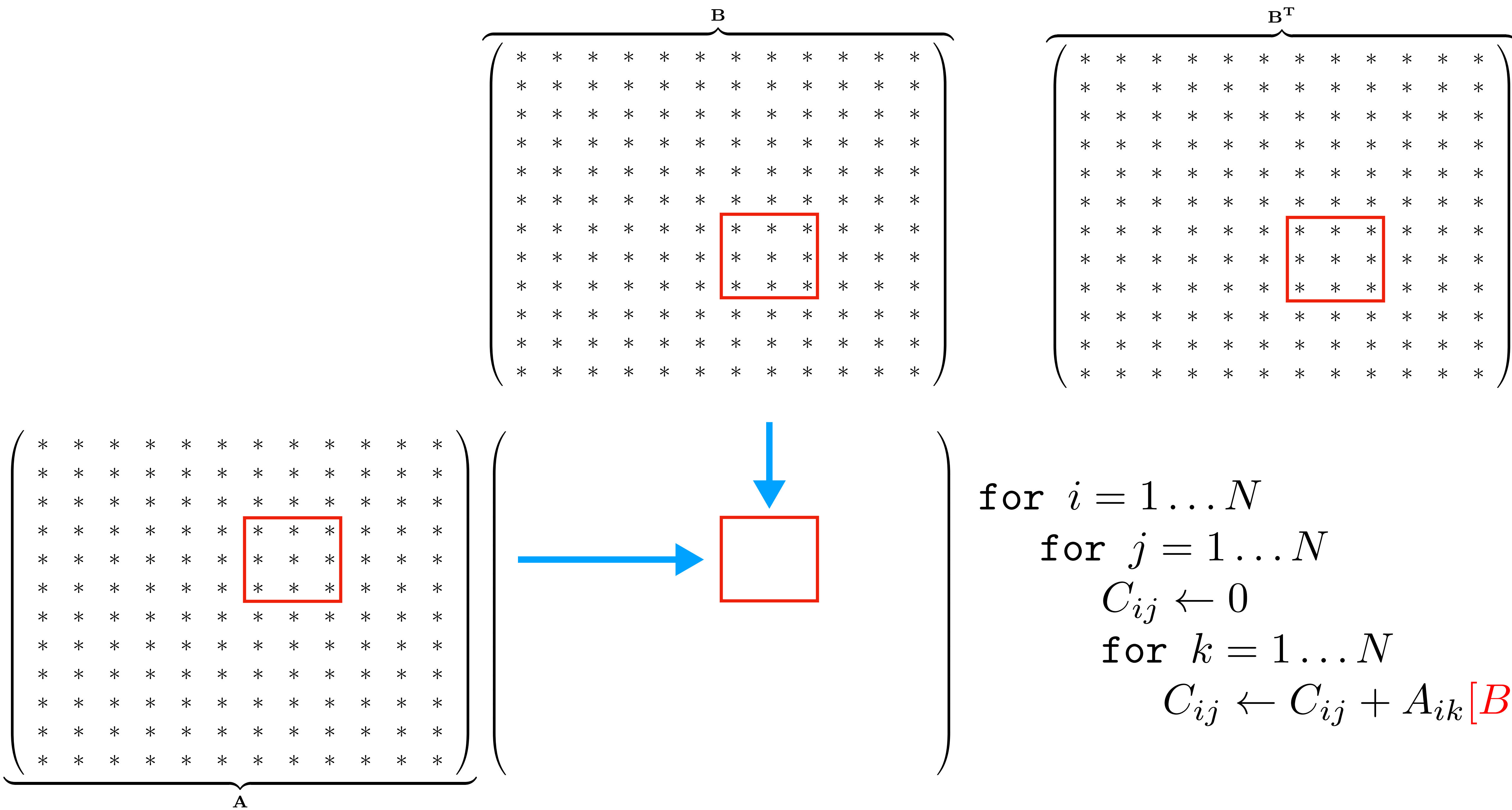




$$
\begin{aligned}
&\texttt{for}\ \ i = 1 \ldots N \\
&\quad \texttt{for}\ \ j = 1 \ldots N \\
&\quad\quad C_{ij} \leftarrow 0 \\
&\quad\quad \texttt{for}\ \ k = 1 \ldots N \\
&\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} [B^T]_{jk}
\end{aligned}
$$

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

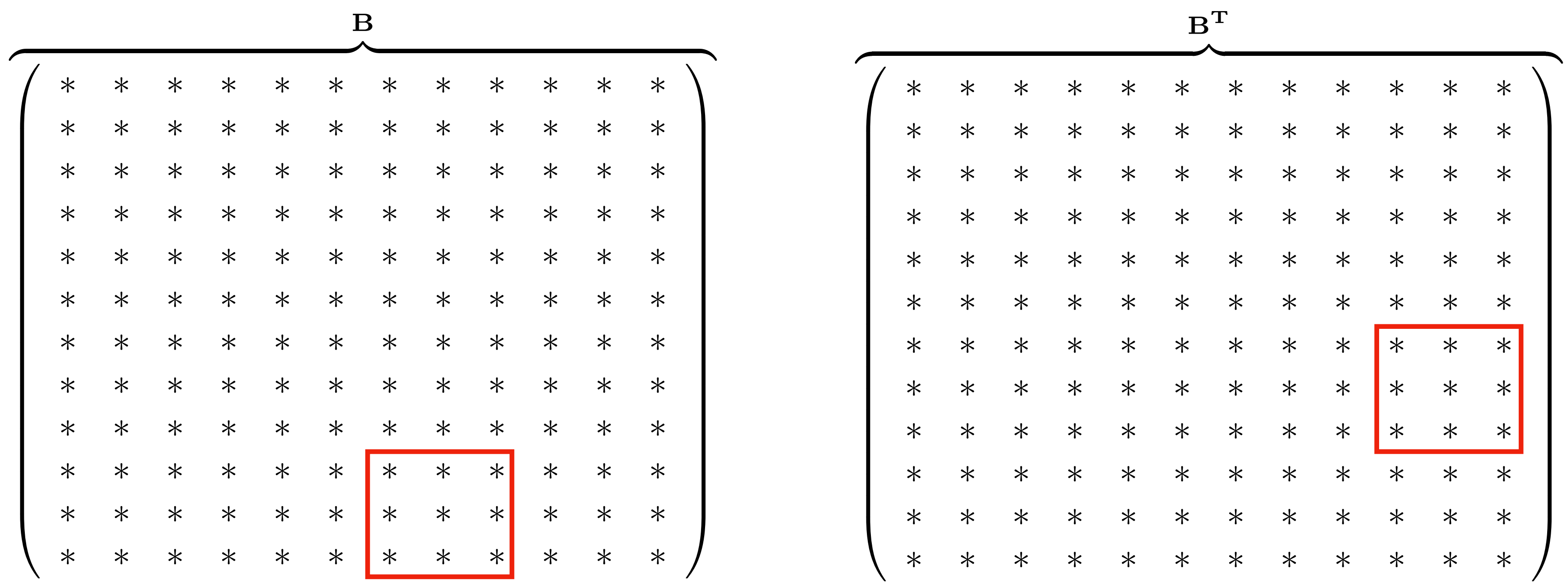
Combining blocking & pre-transposed B (or col-major B)

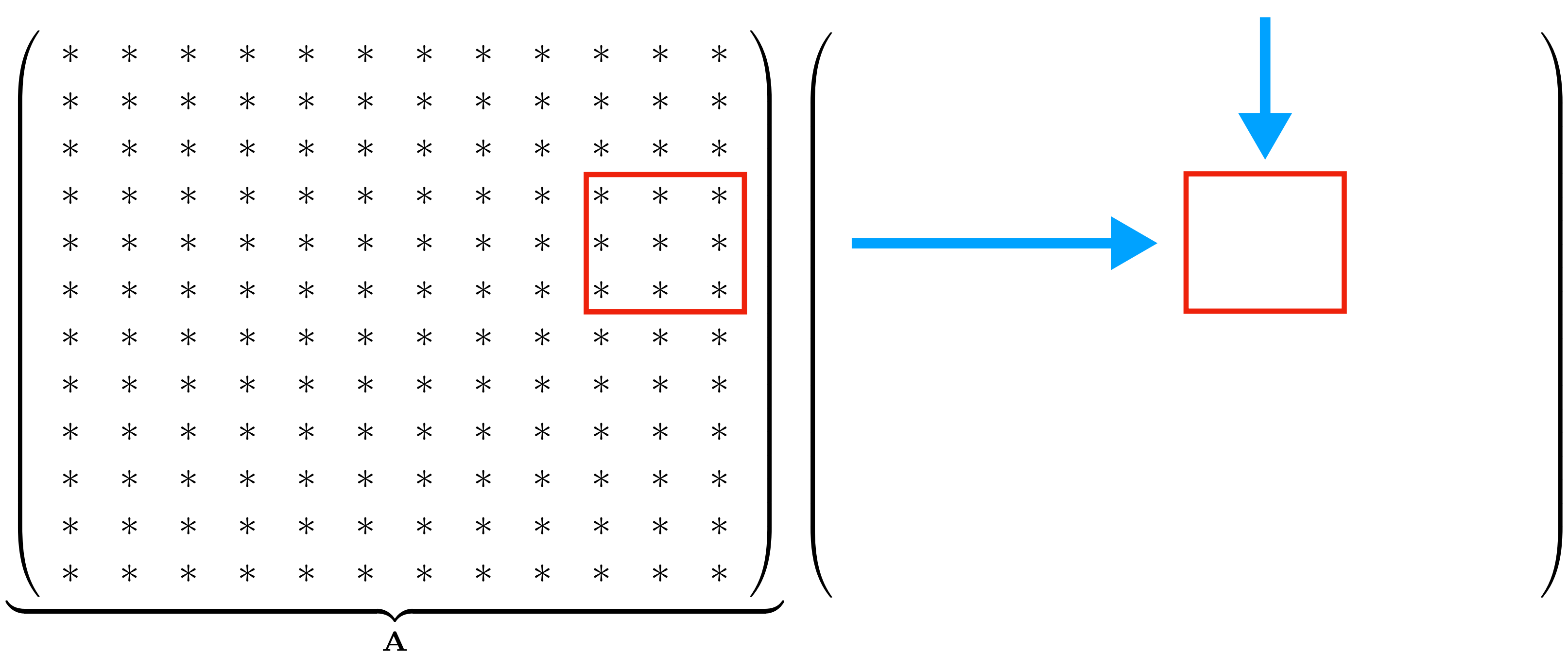


$$
\begin{aligned}
&\texttt{for}\ \ i = 1 \ldots N \\
&\quad \texttt{for}\ \ j = 1 \ldots N \\
&\quad\quad C_{ij} \leftarrow 0 \\
&\quad\quad \texttt{for}\ \ k = 1 \ldots N \\
&\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} [B^T]_{jk}
\end{aligned}
$$

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

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* *block 3x3 sub-matrices*

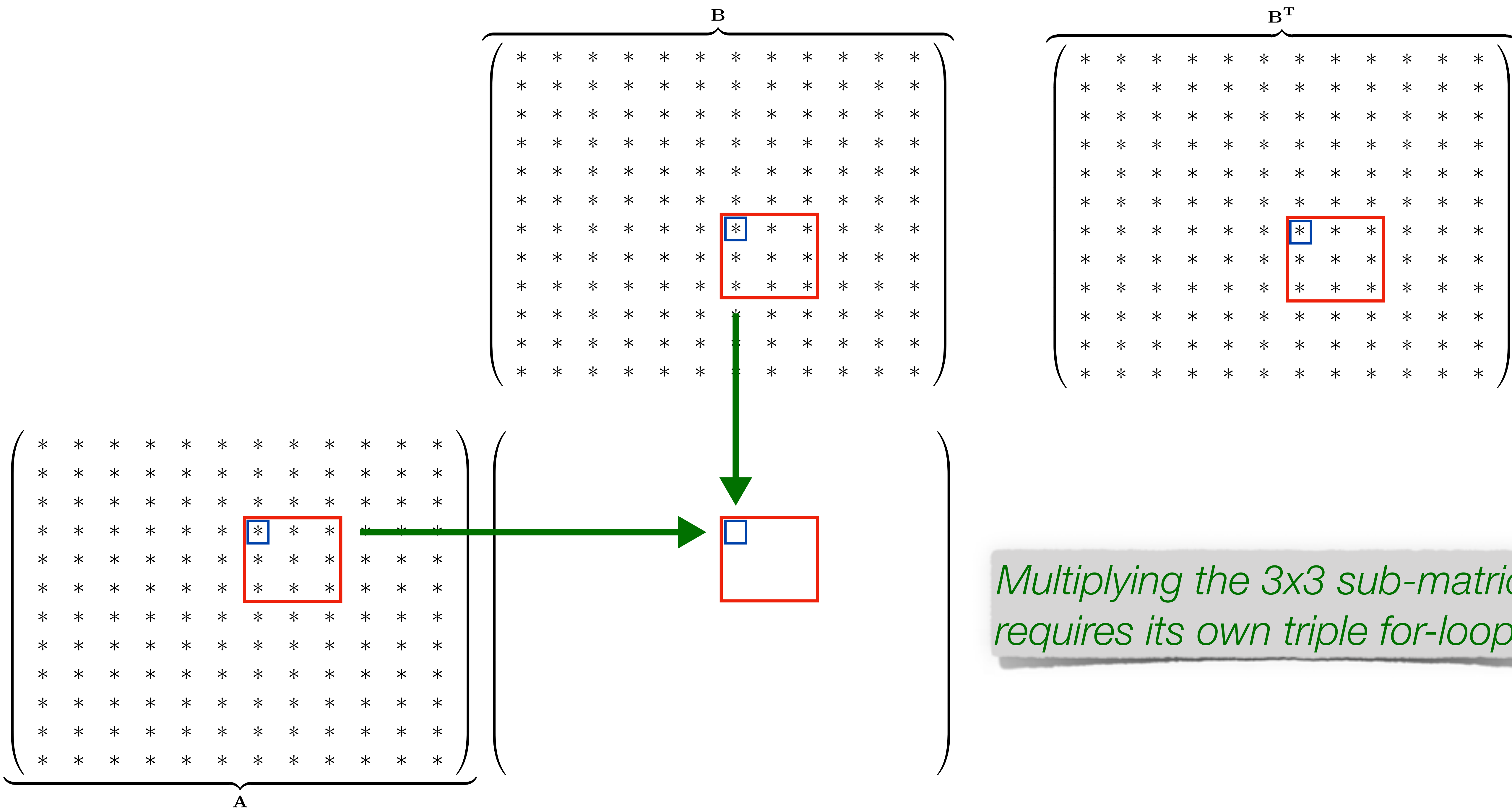
GEMM routine (MatMatMultiply.cpp)

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];
}
```

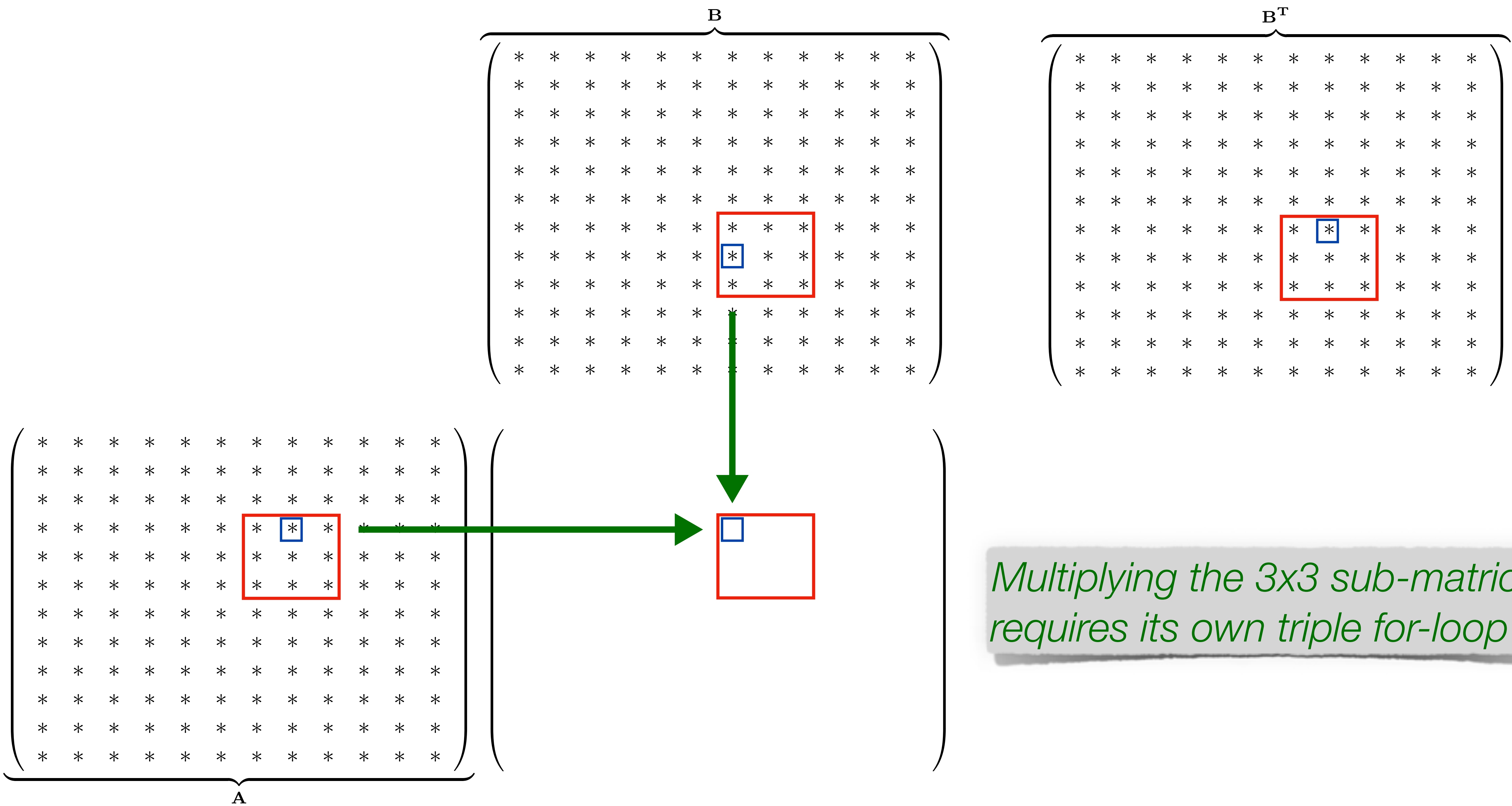
*… and iterate again **within** the blocks, to multiply them together*

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



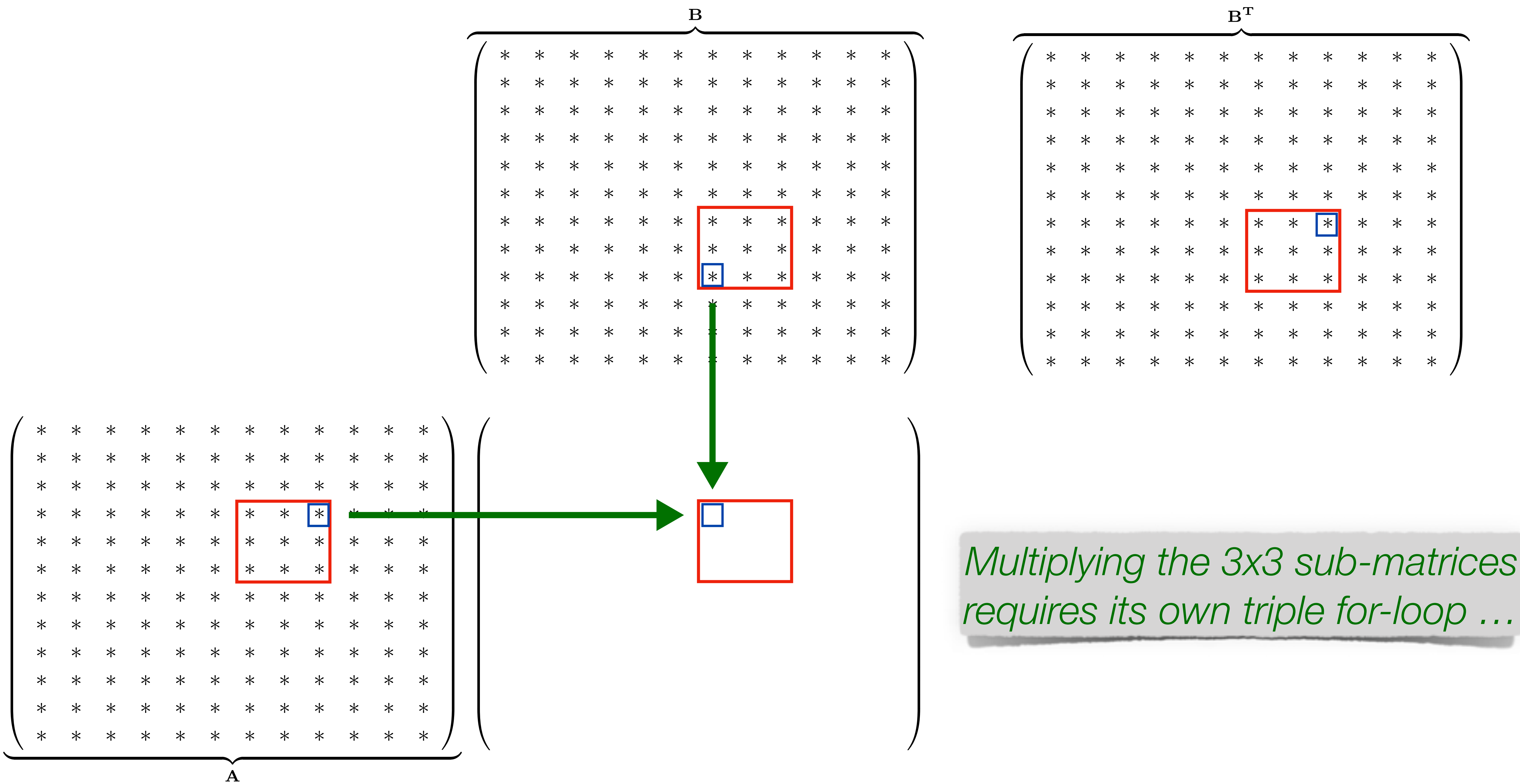
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

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



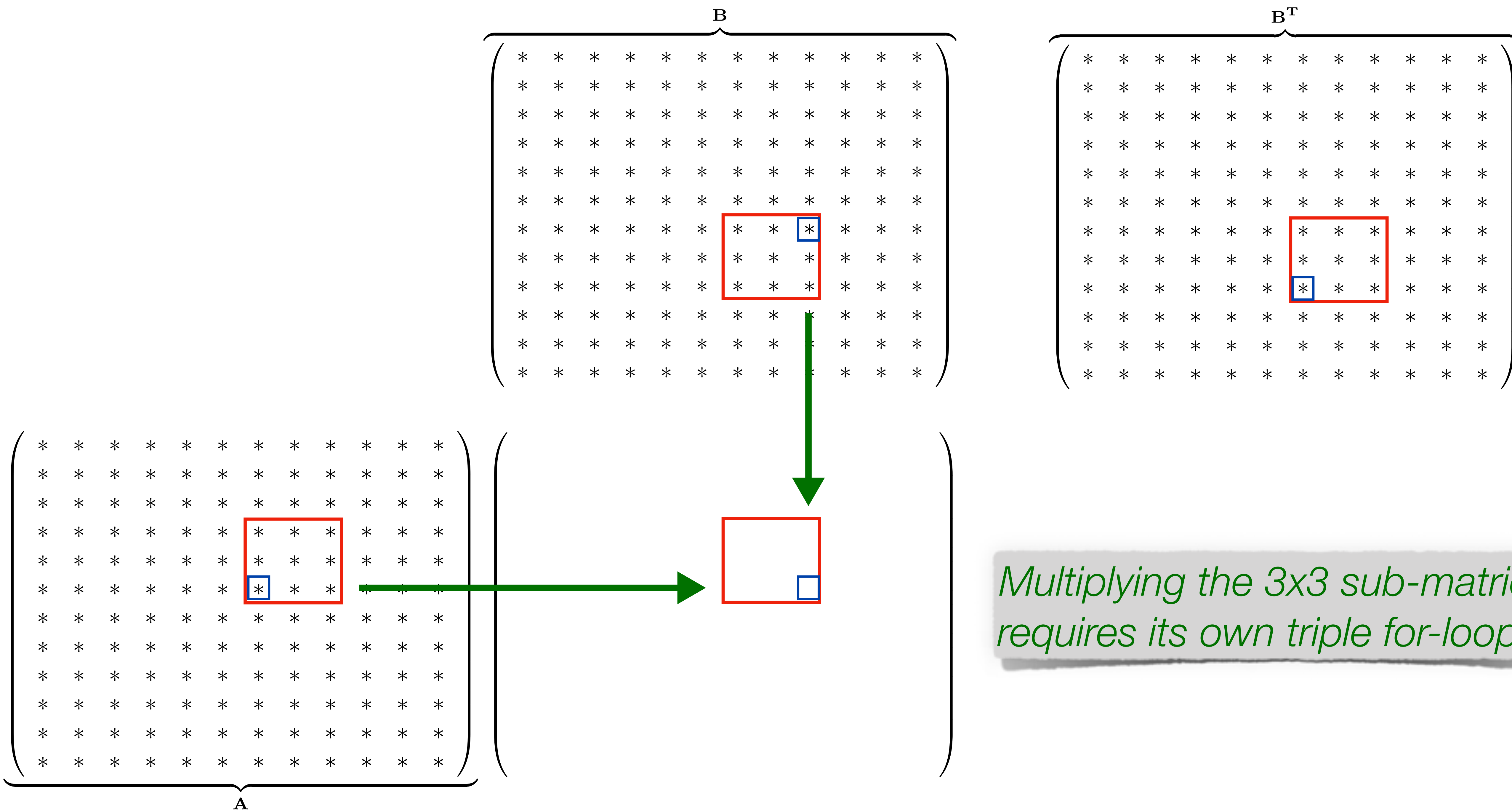
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

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



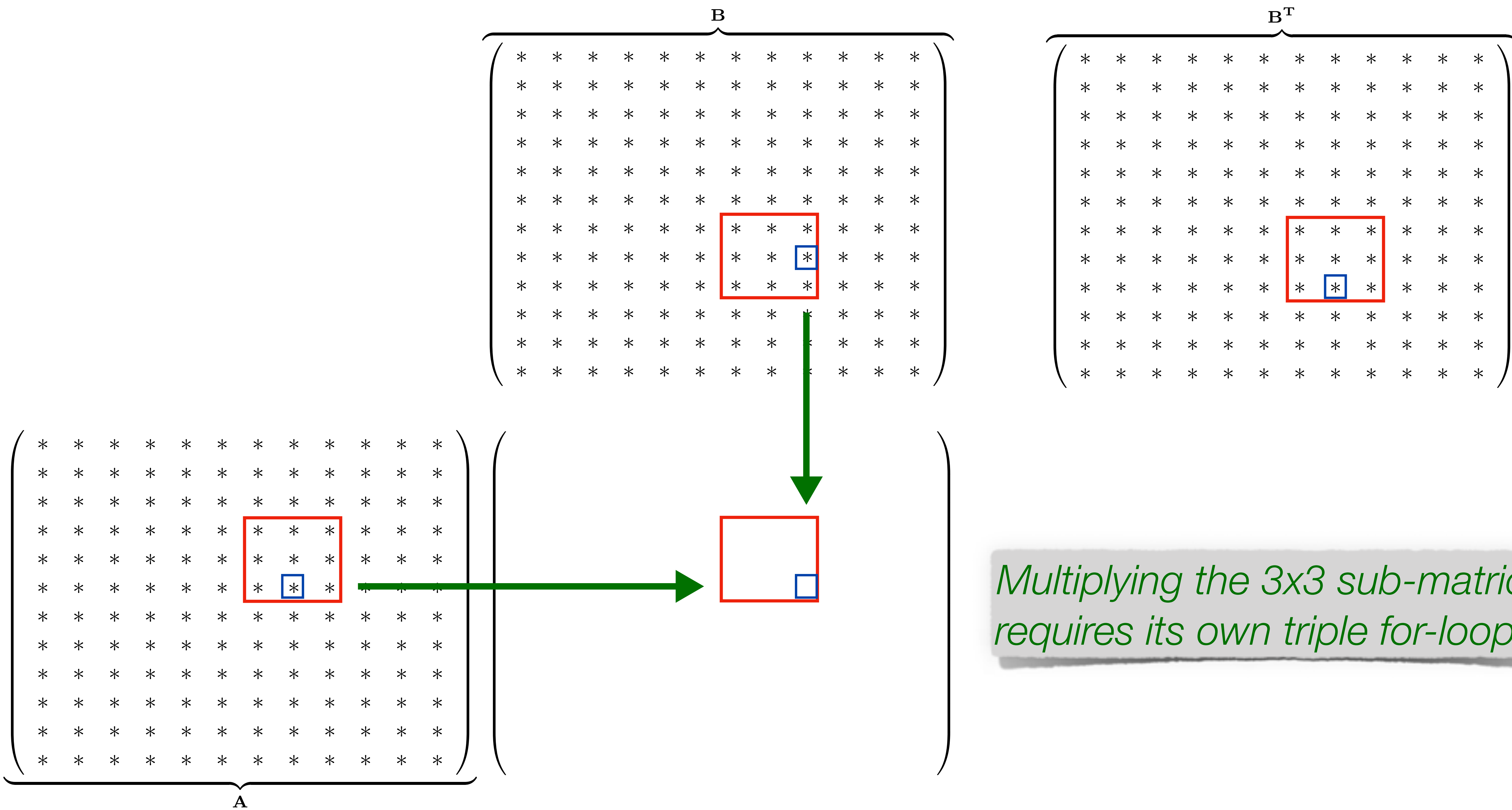
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

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



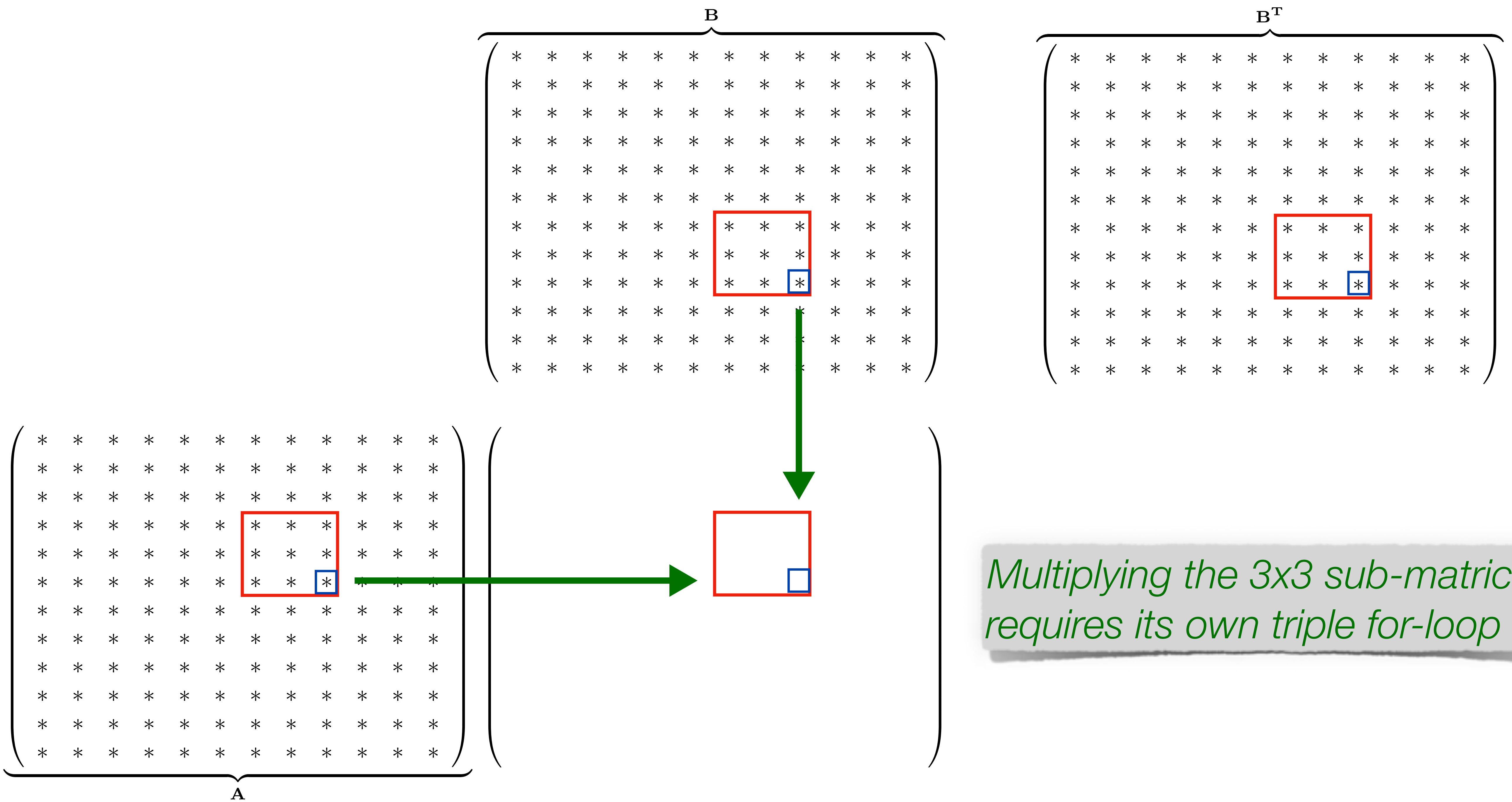
Multiplying the 3x3 sub-matrices requires its own triple for-loop …

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]
```

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_7

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

            float localA[BLOCK_SIZE][BLOCK_SIZE];
            float localB[BLOCK_SIZE][BLOCK_SIZE];
            float localC[BLOCK_SIZE][BLOCK_SIZE];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++) {
                localA[ii][jj] = blockA[bi][ii][bk][jj];
                localB[ii][jj] = blockB[bj][ii][bk][jj];
                localC[ii][jj] = 0.;}

            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++)
                    localC[ii][jj] += localA[ii][kk] * localB[jj][kk];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                blockC[bi][ii][bj][jj] += localC[ii][jj];
        }
}
```

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_7

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

            float localA[BLOCK_SIZE][BLOCK_SIZE];
            float localB[BLOCK_SIZE][BLOCK_SIZE];
            float localC[BLOCK_SIZE][BLOCK_SIZE];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++) {
                localA[ii][jj] = blockA[bi][ii][bk][jj];
                localB[ii][jj] = blockB[bj][ii][bk][jj];
                localC[ii][jj] = 0.;}

            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++)
                    localC[ii][jj] += localA[ii][kk] * localB[jj][kk];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                blockC[bi][ii][bj][jj] += localC[ii][jj];
        }
}
```

Further optimization:
Make local copies of block matrices while multiplying them (increased chances of them being cached)

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_7

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

            float localA[BLOCK_SIZE][BLOCK_SIZE];
            float localB[BLOCK_SIZE][BLOCK_SIZE];
            float localC[BLOCK_SIZE][BLOCK_SIZE];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++) {
                localA[ii][jj] = blockA[bi][ii][bk][jj];
                localB[ii][jj] = blockB[bj][ii][bk][jj];
                localC[ii][jj] = 0.;}

            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++)
                    localC[ii][jj] += localA[ii][kk] * localB[jj][kk];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                blockC[bi][ii][bj][jj] += localC[ii][jj];
        }
}
```

Populate the entries of these local matrices …

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_7

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

            float localA[BLOCK_SIZE][BLOCK_SIZE];
            float localB[BLOCK_SIZE][BLOCK_SIZE];
            float localC[BLOCK_SIZE][BLOCK_SIZE];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++) {
                localA[ii][jj] = blockA[bi][ii][bk][jj];
                localB[ii][jj] = blockB[bj][ii][bk][jj];
                localC[ii][jj] = 0.;}

            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++)
                    localC[ii][jj] += localA[ii][kk] * localB[jj][kk];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                blockC[bi][ii][bj][jj] += localC[ii][jj];
        }
}
```

Do the multiplication with local (hopefully cached) matrices …

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_7

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

            float localA[BLOCK_SIZE][BLOCK_SIZE];
            float localB[BLOCK_SIZE][BLOCK_SIZE];
            float localC[BLOCK_SIZE][BLOCK_SIZE];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++) {
                localA[ii][jj] = blockA[bi][ii][bk][jj];
                localB[ii][jj] = blockB[bj][ii][bk][jj];
                localC[ii][jj] = 0.;}

            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++)
                    localC[ii][jj] += localA[ii][kk] * localB[jj][kk];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                blockC[bi][ii][bj][jj] += localC[ii][jj];
        }
}
```

(#pragma omp simd does the right thing here, across compilers)

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_7

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

            float localA[BLOCK_SIZE][BLOCK_SIZE];
            float localB[BLOCK_SIZE][BLOCK_SIZE];
            float localC[BLOCK_SIZE][BLOCK_SIZE];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++) {
                localA[ii][jj] = blockA[bi][ii][bk][jj];
                localB[ii][jj] = blockB[bj][ii][bk][jj];
                localC[ii][jj] = 0.;}

            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++)
                    localC[ii][jj] += localA[ii][kk] * localB[jj][kk];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                blockC[bi][ii][bj][jj] += localC[ii][jj];
        }
}
```

… and add the block product back to the output matrix

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_7

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

            float localA[BLOCK_SIZE][BLOCK_SIZE];
            float localB[BLOCK_SIZE][BLOCK_SIZE];
            float localC[BLOCK_SIZE][BLOCK_SIZE];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++) {
                lo
                lo
                lo

            for (i
            for (i
#pragma omp simd
                fo


            for (i
            for (i
                bl
        }
}
```

Matrix size 2048 x 2048

Execution:

```
Transposing second matrix factor ... [Elapsed time : 40.4736ms]
Running candidate kernel for correctness test ... [Elapsed time : 83.171ms]
Running reference kernel for correctness test ... [Elapsed time : 15.4552ms]
Discrepancy between two methods : 4.95911e-05
Running kernel for performance run # 1 ... [Elapsed time : 77.7191ms]
Running kernel for performance run # 2 ... [Elapsed time : 77.0233ms]
Running kernel for performance run # 3 ... [Elapsed time : 76.8495ms]
Running kernel for performance run # 4 ... [Elapsed time : 78.1119ms]
Running kernel for performance run # 5 ... [Elapsed time : 77.1877ms]
Running kernel for performance run # 6 ... [Elapsed time : 77.3701ms]
Running kernel for performance run # 7 ... [Elapsed time : 77.3569ms]
Running kernel for performance run # 8 ... [Elapsed time : 77.822ms]
[...]
```

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_7

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

            float localA[BLOCK_SIZE][BLOCK_SIZE];
            float localB[BLOCK_SIZE][BLOCK_SIZE];
            float localC[BLOCK_SIZE][BLOCK_SIZE];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++) {
                localA[ii][jj] = blockA[bi][ii][bk][jj];
                localB[ii][jj] = blockB[bj][ii][bk][jj];
                localC[ii][jj] = 0.;}

            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++)
                    localC[ii][jj] += localA[ii][kk] * localB[jj][kk];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                blockC[bi][ii][bj][jj] += localC[ii][jj];
        }
}
```

Can we do better than creating these local copies every time? (they're stack allocated, but do they always stay in the same place in cache?)

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_8

```
alignas(64) float localA[BLOCK_SIZE][BLOCK_SIZE];
alignas(64) float localB[BLOCK_SIZE][BLOCK_SIZE];
alignas(64) float localC[BLOCK_SIZE][BLOCK_SIZE];
#pragma omp threadprivate(localA, localB, localC)

 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 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++) {
                localA[ii][jj] = blockA[bi][ii][bk][jj];
                localB[ii][jj] = blockB[bj][ii][bk][jj];
                localC[ii][jj] = 0.;}

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
#pragma omp simd aligned(localA: 64, localB: 64, localC: 64)
                for (int kk = 0; kk < BLOCK_SIZE; kk++)
                    localC[ii][jj] += localA[ii][kk] * localB[jj][kk];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                blockC[bi][ii][bj][jj] += localC[ii][jj];
        }
}
```

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_8

```
alignas(64) float localA[BLOCK_SIZE][BLOCK_SIZE];
alignas(64) float localB[BLOCK_SIZE][BLOCK_SIZE];
alignas(64) float localC[BLOCK_SIZE][BLOCK_SIZE];
#pragma omp threadprivate(localA, localB, localC)

 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 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++) {
                localA[ii][jj] = blockA[bi][ii][bk][jj];
                localB[ii][jj] = blockB[bj][ii][bk][jj];
                localC[ii][jj] = 0.;}

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
#pragma omp simd aligned(localA: 64, localB: 64, localC: 64)
                for (int kk = 0; kk < BLOCK_SIZE; kk++)
                    localC[ii][jj] += localA[ii][kk] * localB[jj][kk];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                blockC[bi][ii][bj][jj] += localC[ii][jj];
        }
}
```

*The **threadprivate** clause creates a local copy of each array, but only once per thread (rather than once per iteration of the for-loop)*

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_8

```
alignas(64) float localA[BLOCK_SIZE][BLOCK_SIZE];
alignas(64) float localB[BLOCK_SIZE][BLOCK_SIZE];
alignas(64) float localC[BLOCK_SIZE][BLOCK_SIZE];
#pragma omp threadprivate(localA, localB, localC)

 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 bi = 0; bi < NBLOCKS; bi++)
    for (int bj = 0; bj < NBLOCKS; bj++)
        for (int bk = 0; bk < NBLOCKS; bk++) {
            for (int ii = 0; ii < BLOC
            for (int jj = 0; jj < BLOC
                localA[ii][jj] = block
                localB[ii][jj] = blockB[bj][ii][bk][jj];
                localC[ii][jj] = 0.;}

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
#pragma omp simd aligned(localA: 64, localB: 64, localC: 64)
                for (int kk = 0; kk < BLOCK_SIZE; kk++)
                    localC[ii][jj] += localA[ii][kk] * localB[jj][kk];

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                blockC[bi][ii][bj][jj] += localC[ii][jj];
        }
}
```

If we know that our local matrices are aligned, we can provide that extra hint to #pragma omp simd

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_8

```
alignas(64) float localA[BLOCK_SIZE][BLOCK_SIZE];
alignas(64) float localB[BLOCK_SIZE][BLOCK_SIZE];
alignas(64) float localC[BLOCK_SIZE][BLOCK_SIZE];
#pragma omp threadprivate(localA, localB, localC)

 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 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++) {
                localA[ii][jj] = blockA[bi][ii][bk][jj];
                lo
                lo

            for (i
            for (i
#pragma omp simd a
                fo


            for (i
            for (i
                bl
        }
}
```

Matrix size 2048 x 2048

Execution:

```
Transposing second matrix factor ... [Elapsed time : 40.2025ms]
Running candidate kernel for correctness test ... [Elapsed time : 81.5569ms]
Running reference kernel for correctness test ... [Elapsed time : 15.0727ms]
Discrepancy between two methods : 4.3869e-05
Running kernel for performance run # 1 ... [Elapsed time : 71.6641ms]
Running kernel for performance run # 2 ... [Elapsed time : 70.7464ms]
Running kernel for performance run # 3 ... [Elapsed time : 71.8588ms]
Running kernel for performance run # 4 ... [Elapsed time : 72.4279ms]
Running kernel for performance run # 5 ... [Elapsed time : 70.8966ms]
Running kernel for performance run # 6 ... [Elapsed time : 70.7259ms]
Running kernel for performance run # 7 ... [Elapsed time : 71.4455ms]
Running kernel for performance run # 8 ... [Elapsed time : 69.7041ms]
[...]
```

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_8

```
alignas(64) float localA[BLOCK_SIZE][BLOCK_SIZE];
alignas(64) float localB[BLOCK_SIZE][BLOCK_SIZE];
alignas(64) float localC[BLOCK_SIZE][BLOCK_SIZE];
#pragma omp threadprivate(localA, localB, localC)

 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 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++) {
                lo
                lo
                lo

            for (i
            for (i
#pragma omp simd a
                fo



            for (i
            for (i
                bl
        }
}
```

Matrix size 1024 x 1024

Execution:

```
Transposing second matrix factor ... [Elapsed time : 17.5709ms]
Running candidate kernel for correctness test ... [Elapsed time : 20.5706ms]
Running reference kernel for correctness test ... [Elapsed time : 2.88713ms]
Discrepancy between two methods : 3.48091e-05
Running kernel for performance run # 1 ... [Elapsed time : 8.86364ms]
Running kernel for performance run # 2 ... [Elapsed time : 8.75748ms]
Running kernel for performance run # 3 ... [Elapsed time : 8.72064ms]
Running kernel for performance run # 4 ... [Elapsed time : 8.73407ms]
Running kernel for performance run # 5 ... [Elapsed time : 8.70559ms]
Running kernel for performance run # 6 ... [Elapsed time : 8.69288ms]
Running kernel for performance run # 7 ... [Elapsed time : 8.66264ms]
Running kernel for performance run # 8 ... [Elapsed time : 8.74384ms]
[...]
```