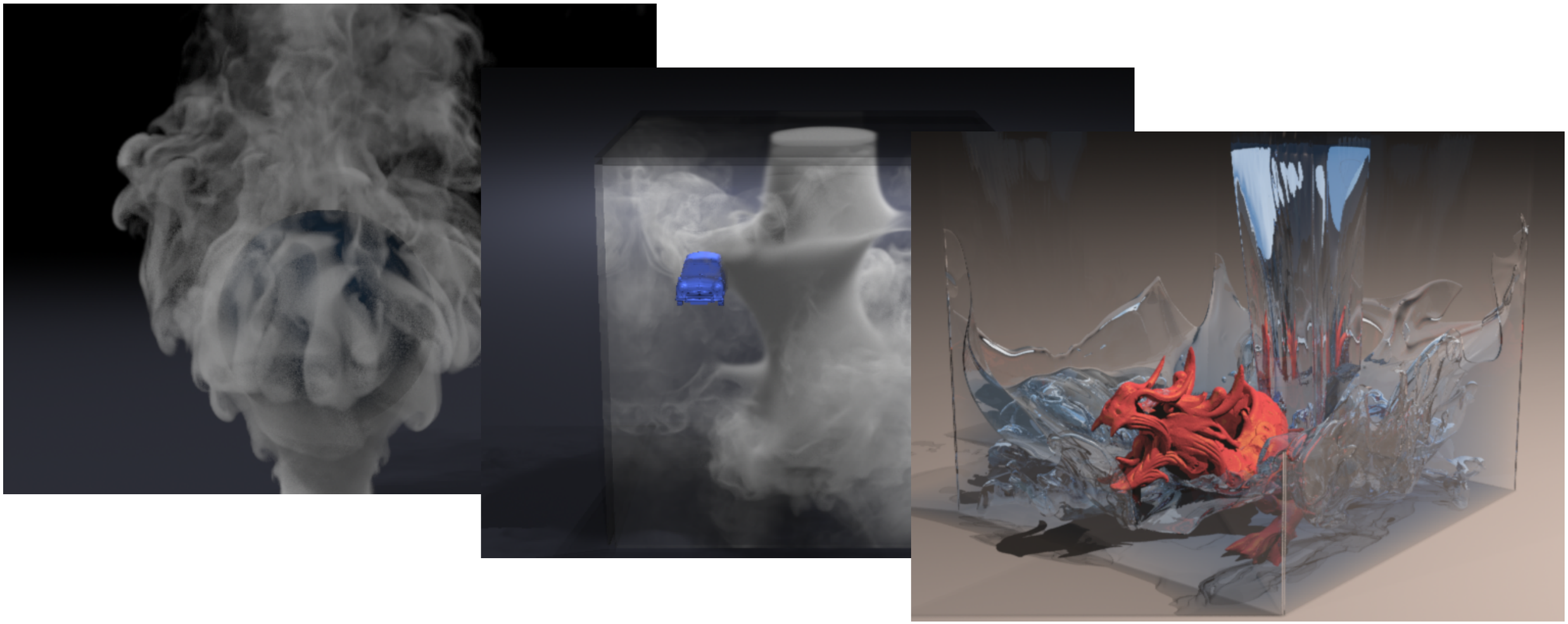
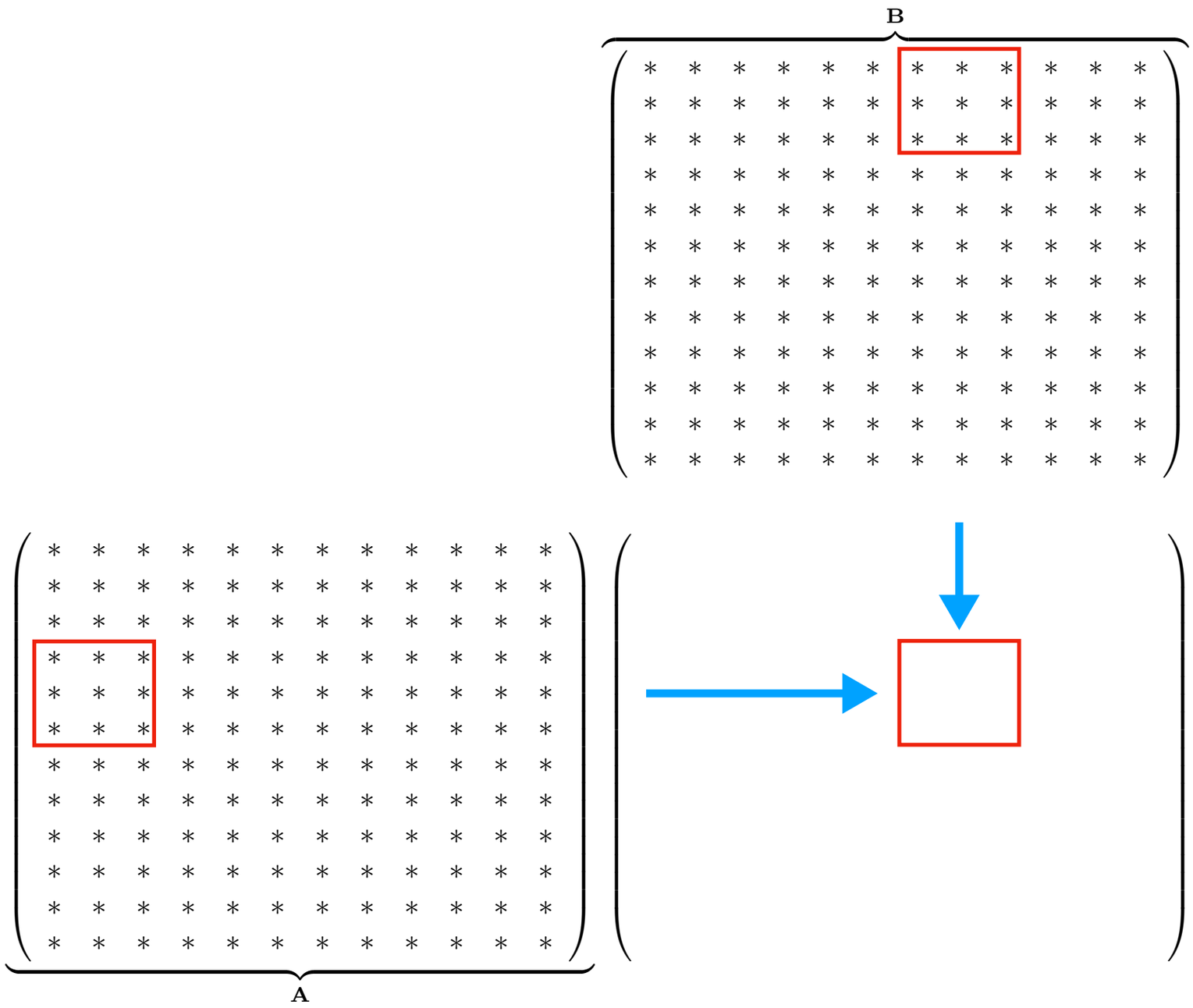


Dense Matrix Computations
Optimizing GEMM Operations in OpenMP
(Part#3 - Advanced Optimizations)

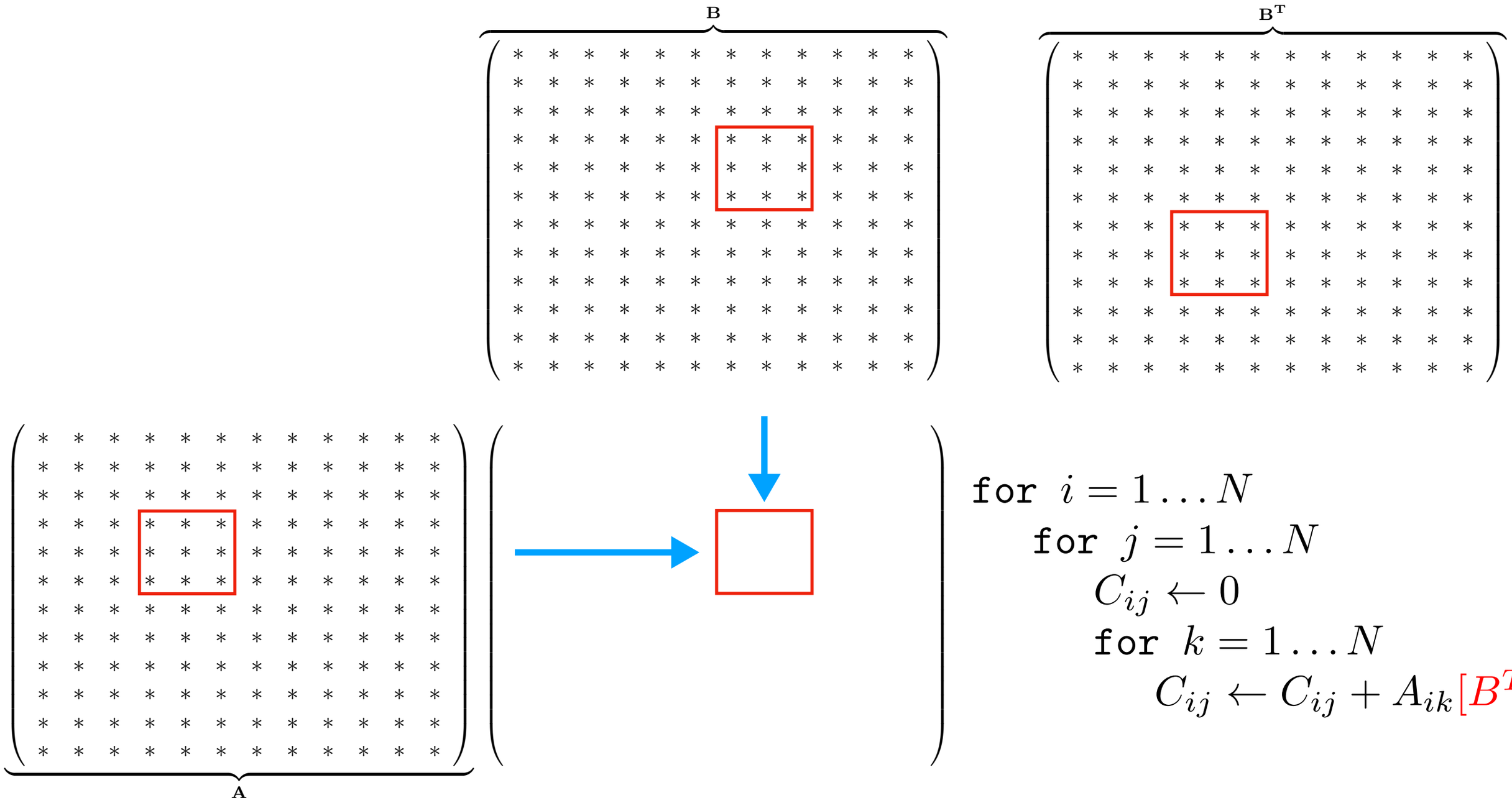
Combining blocking & pre-transposed B (or col-major B)




```
for i = 1...N
    for j = 1...N
        C_ij ← 0
        for k = 1...N
            C_ij ← C_ij + A_ik [B^T]_jk
```

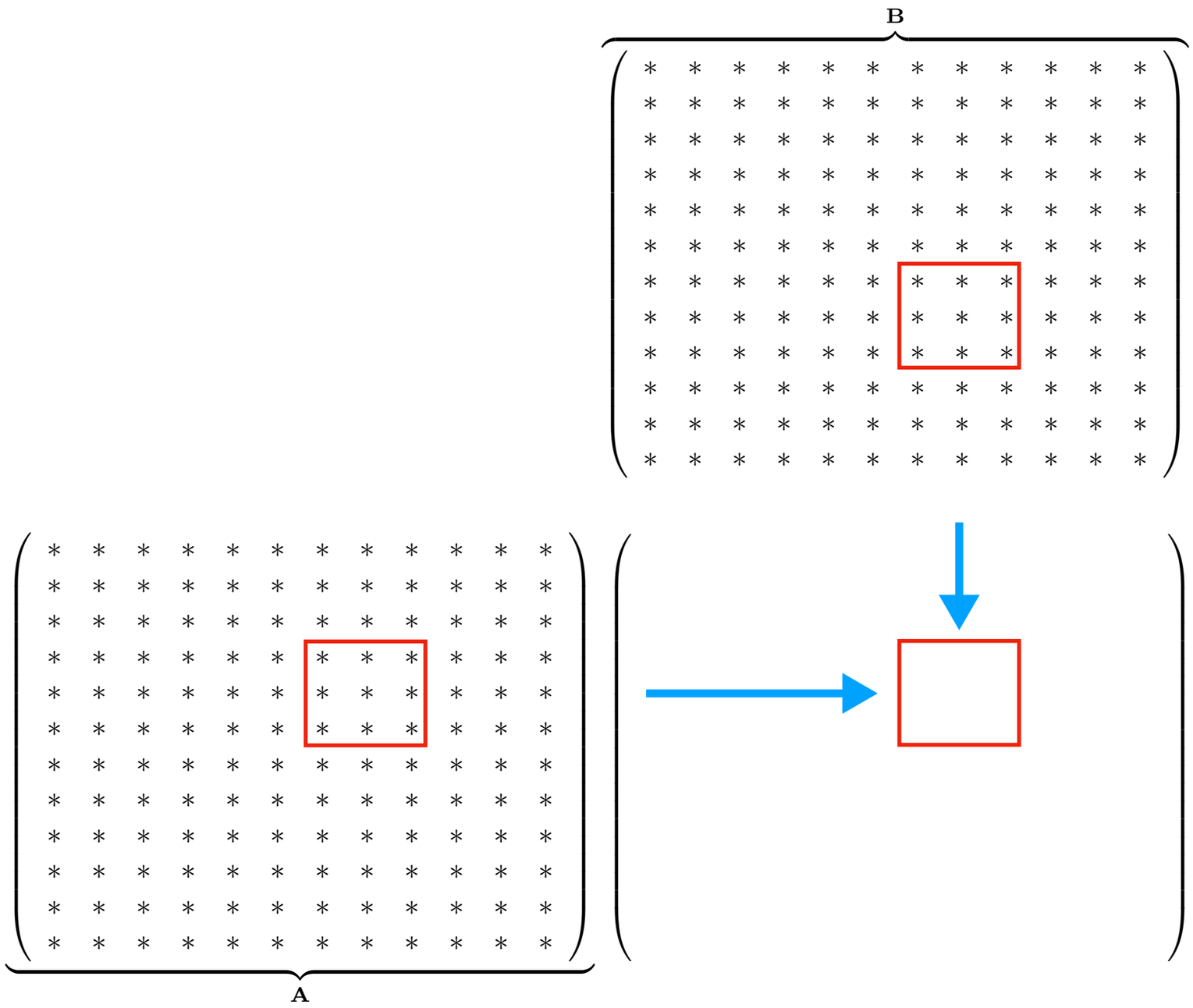
C_{ij}, A_{ik} *and* ***B^T_{jk}*** *represent block 3x3 sub-matrices*

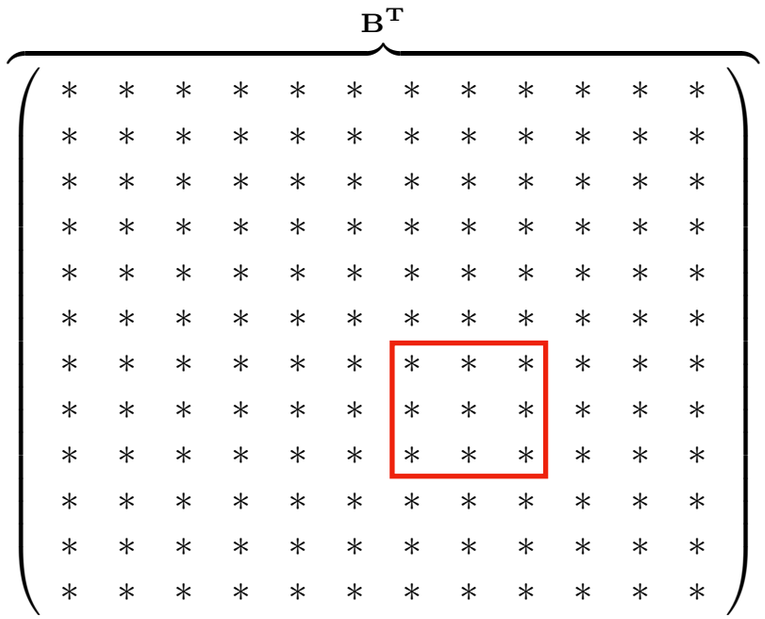
Combining blocking & pre-transposed B (or col-major B)



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

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

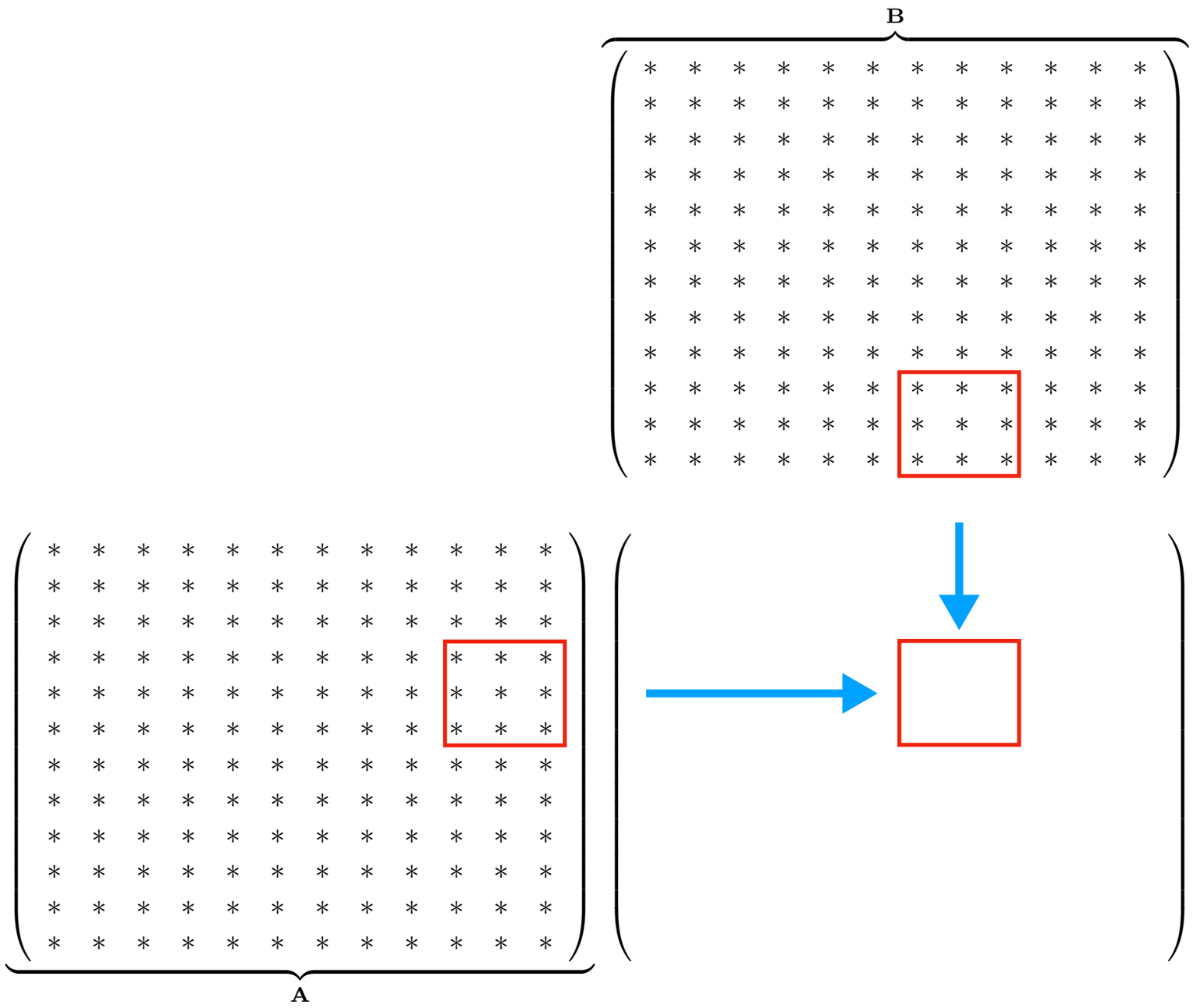




$$
\begin{aligned}
&\texttt{for } i = 1 \ldots N \\
&\quad \texttt{for } j = 1 \ldots N \\
&\qquad C_{ij} \leftarrow 0 \\
&\qquad \texttt{for } k = 1 \ldots N \\
&\qquad\quad C_{ij} \leftarrow C_{ij} + A_{ik}[B^T]_{jk}
\end{aligned}
$$

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

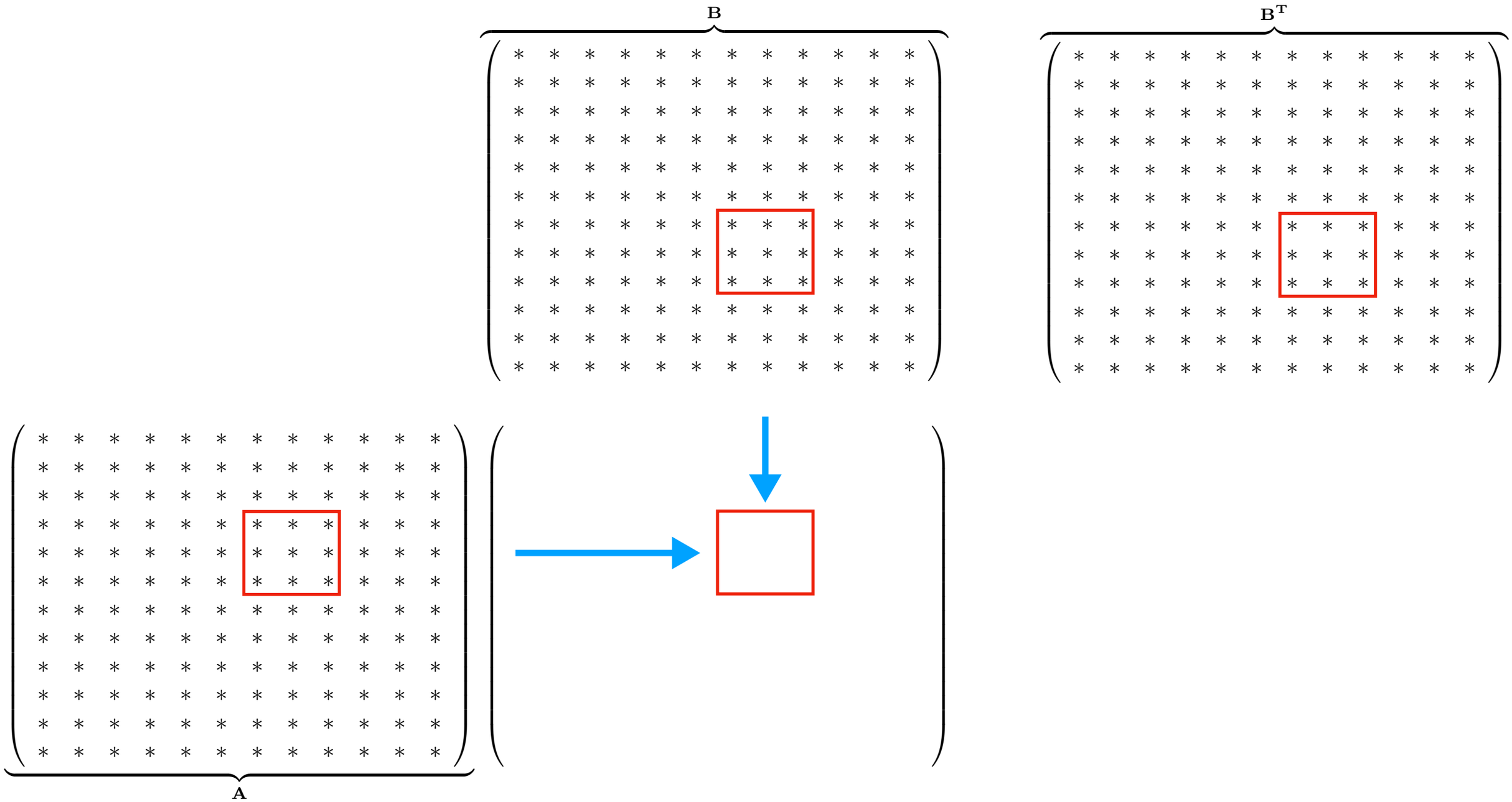
Combining blocking & pre-transposed B (or col-major B)



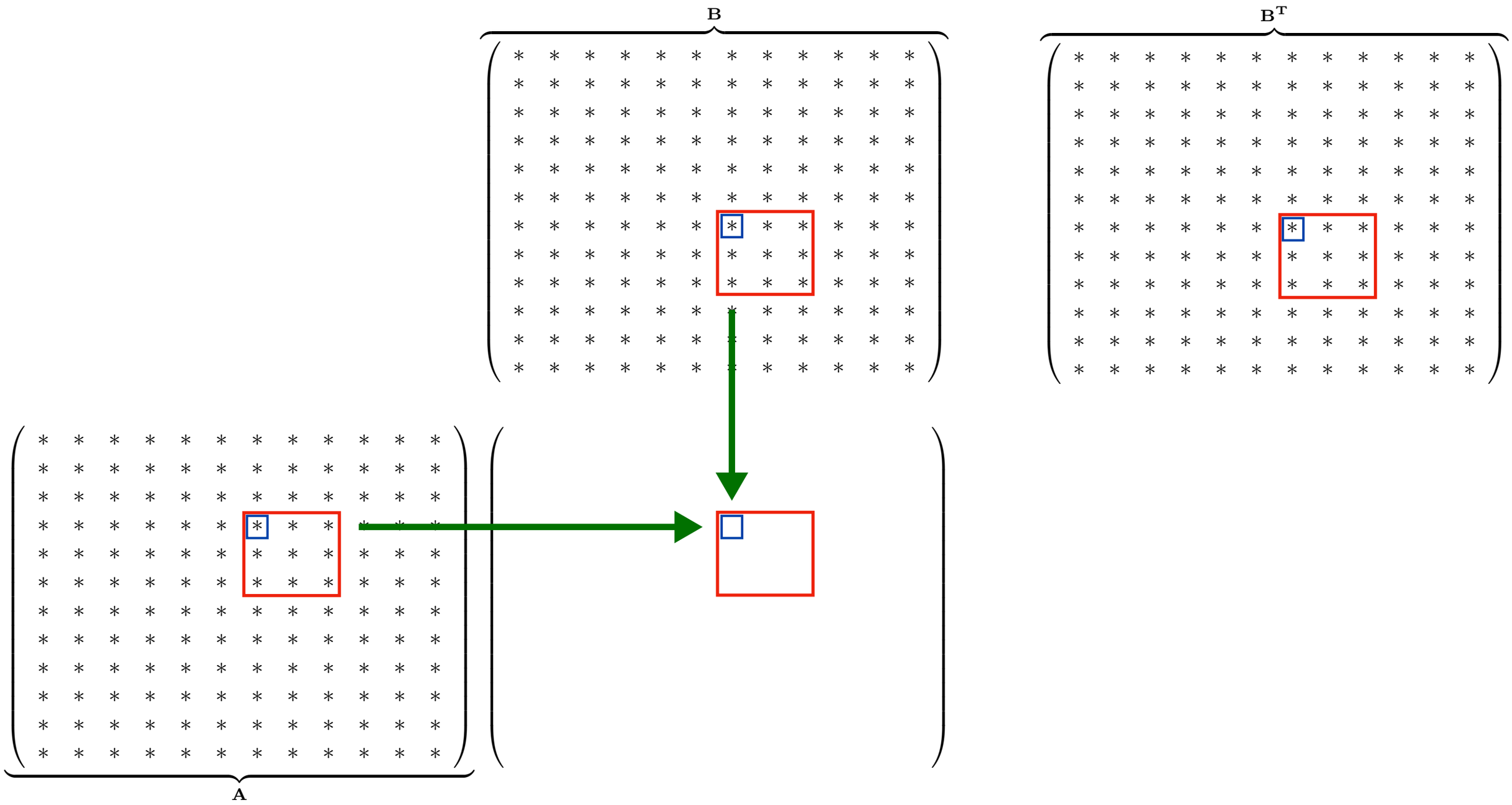

```
for i = 1...N
    for j = 1...N
        C_ij ← 0
        for k = 1...N
            C_ij ← C_ij + A_ik [B^T]_jk
```

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

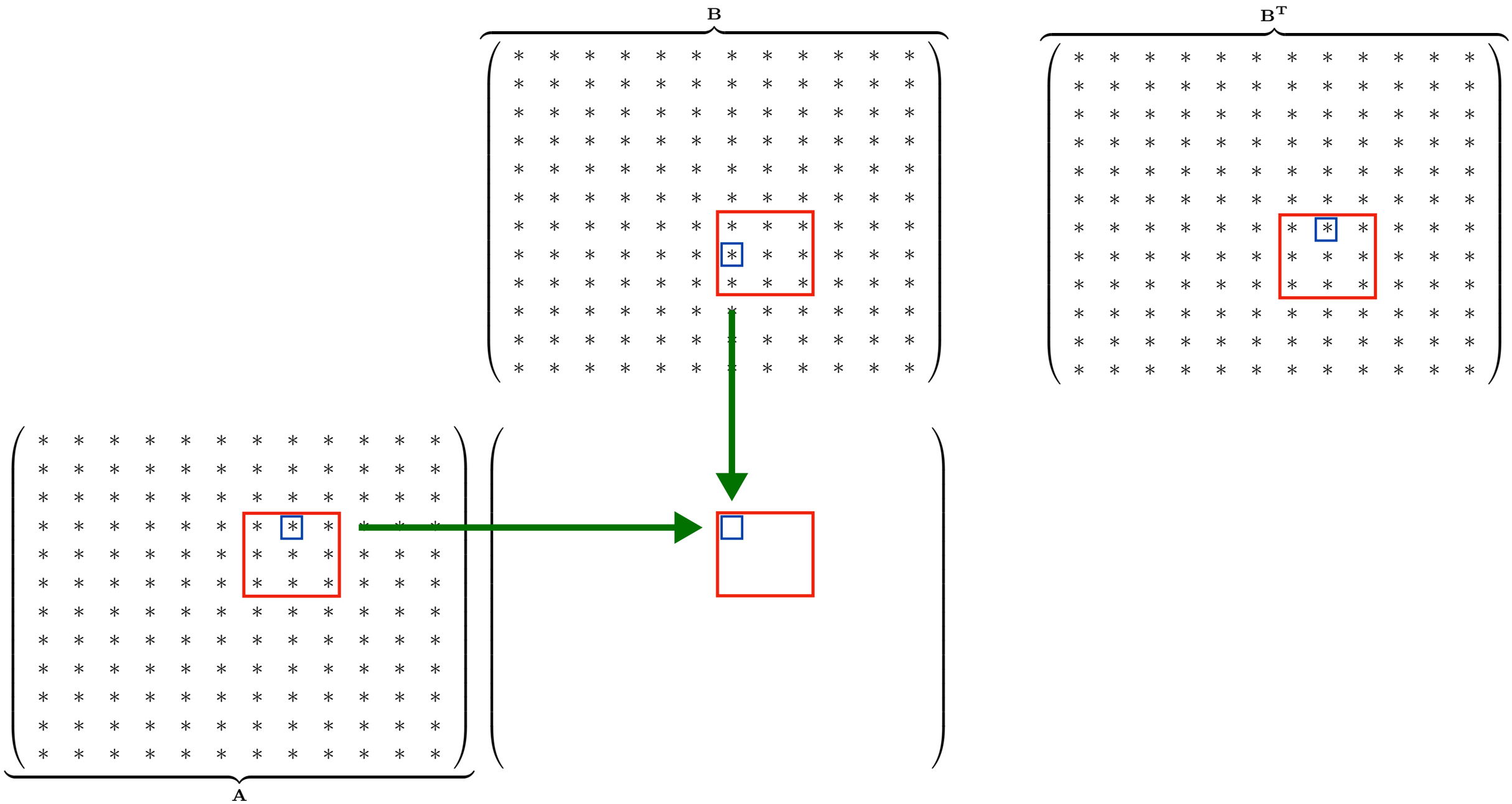
Combining blocking & pre-transposed B (or col-major B)



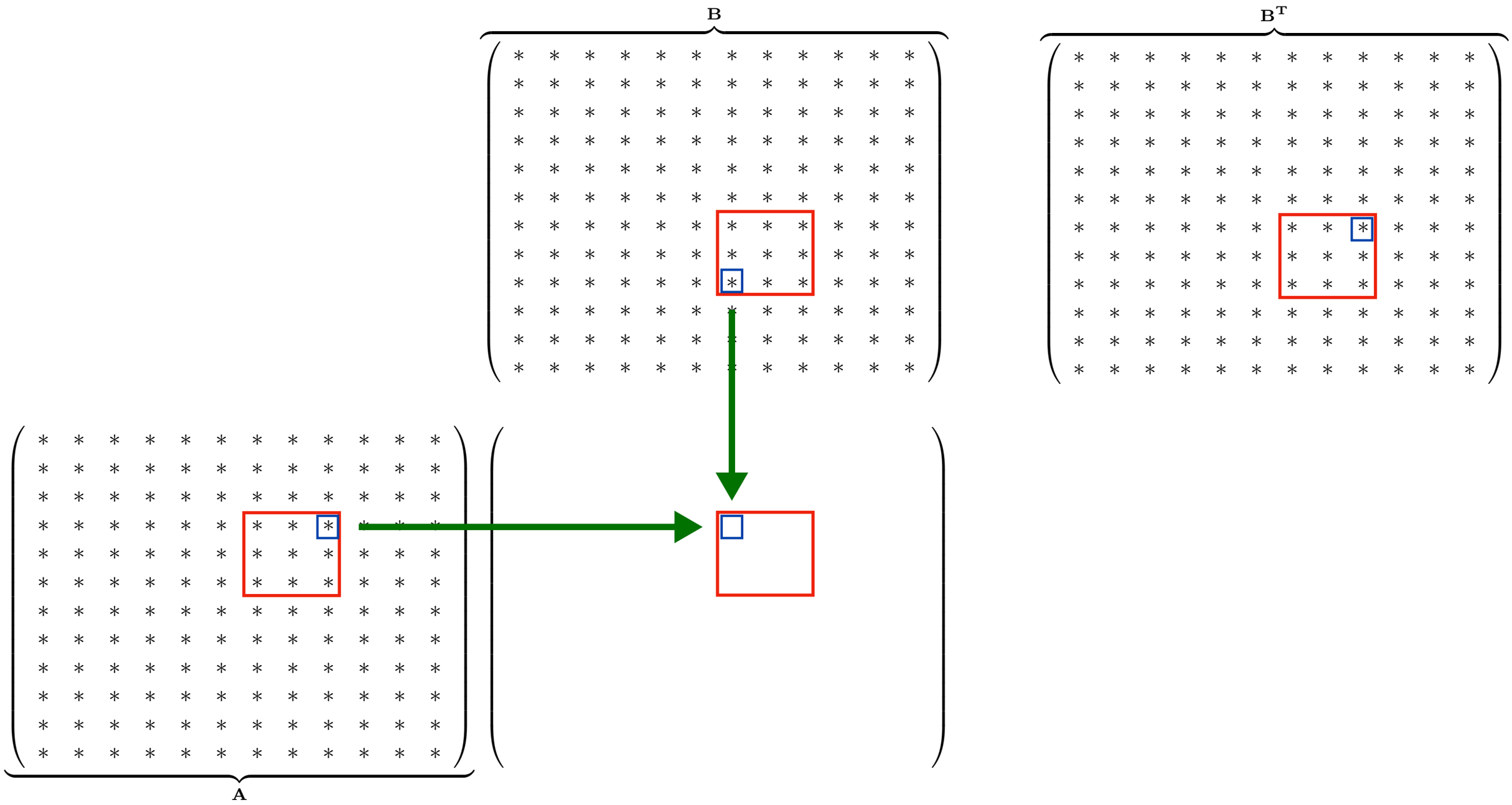
Combining blocking & pre-transposed B (or col-major B)



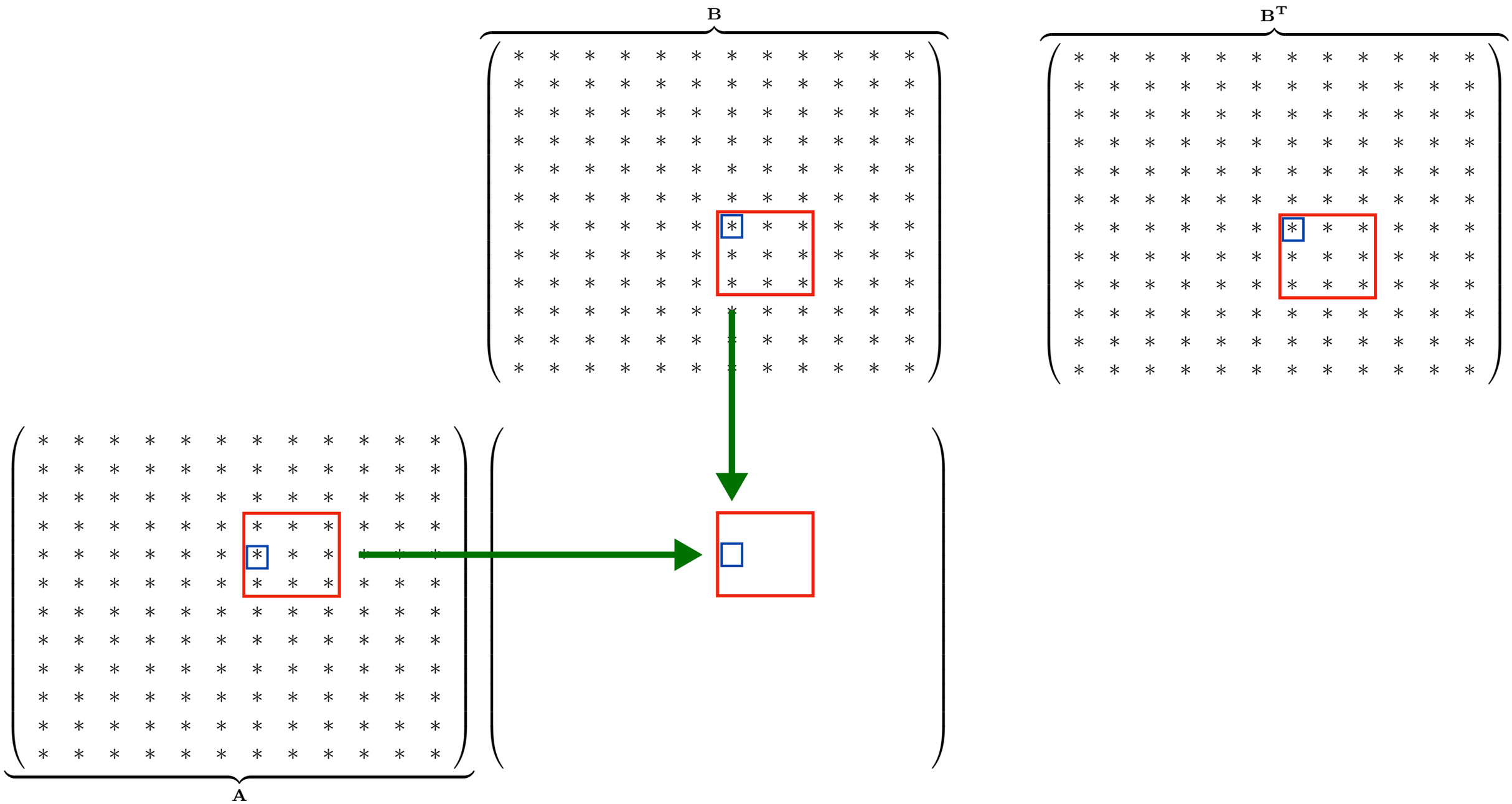
Combining blocking & pre-transposed B (or col-major B)



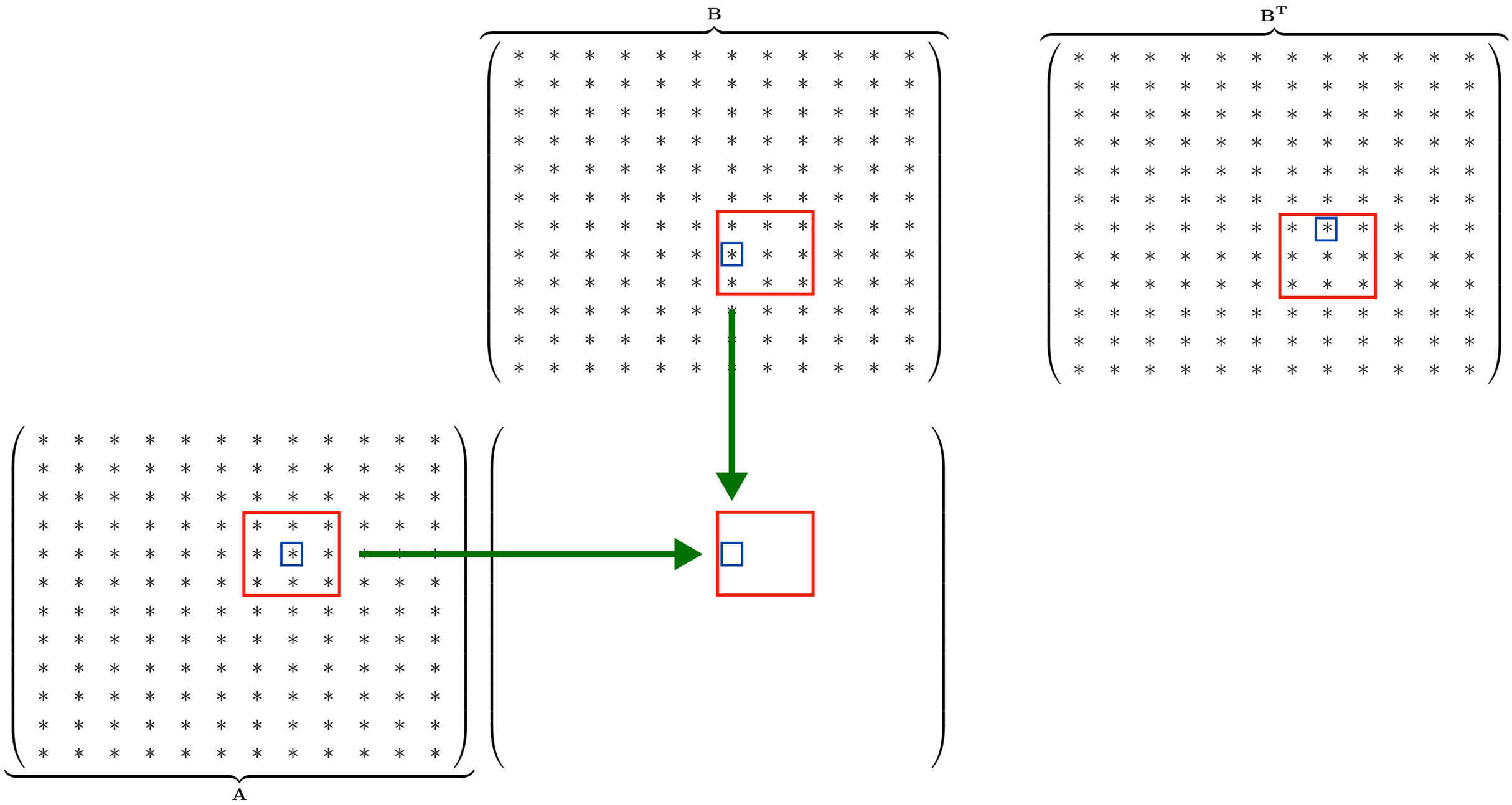
Combining blocking & pre-transposed B (or col-major B)



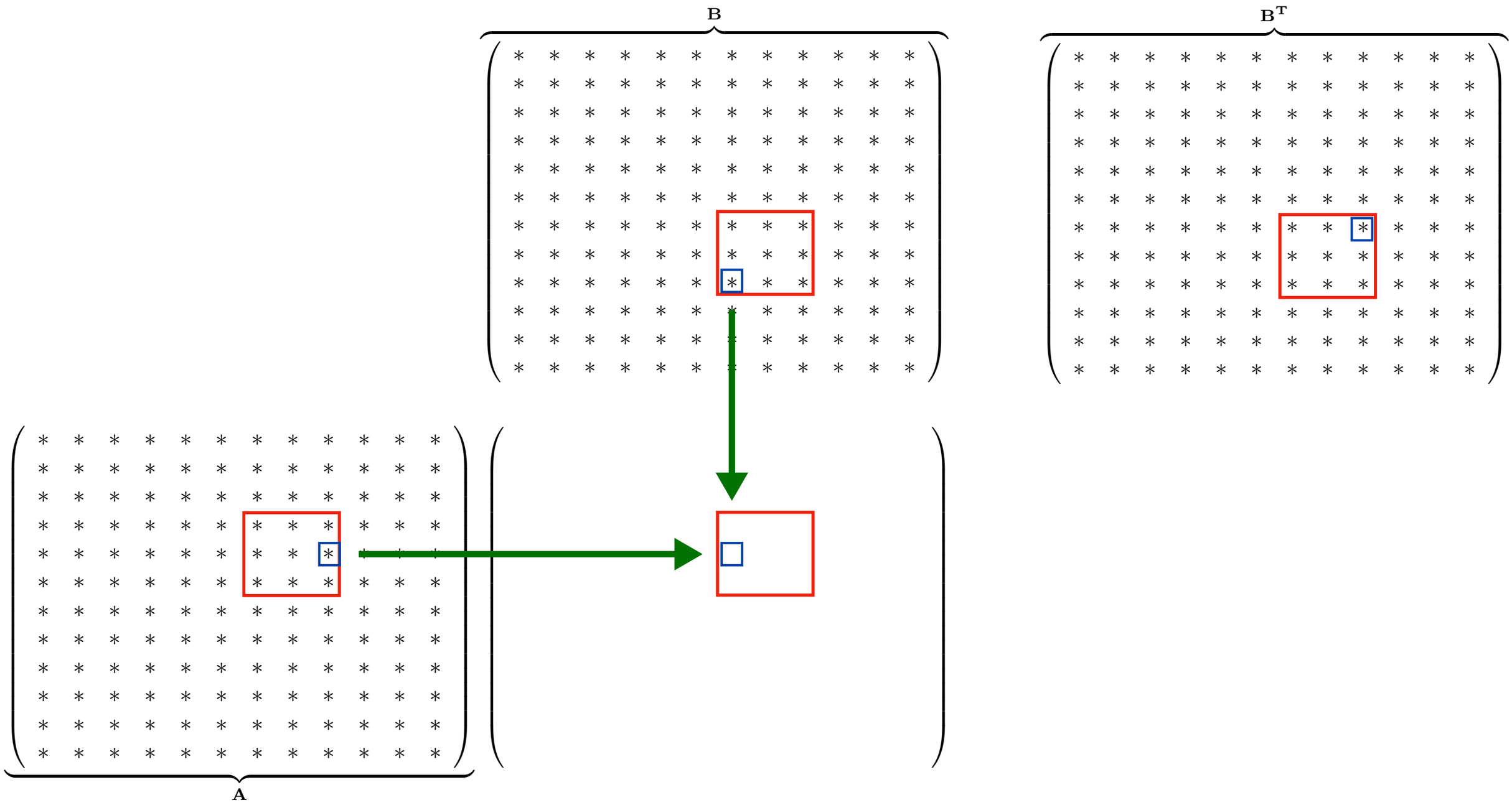
Combining blocking & pre-transposed B (or col-major B)



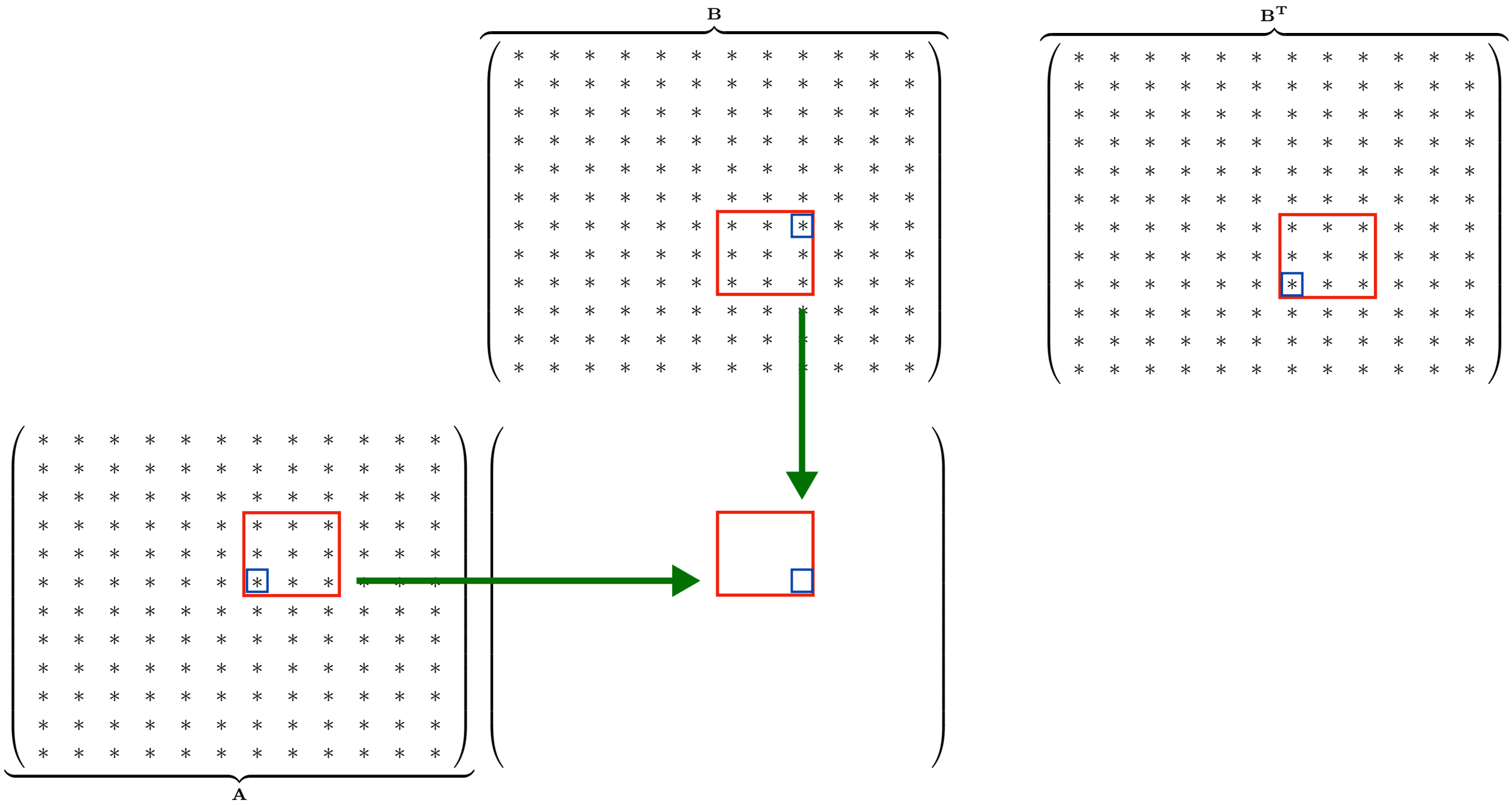
Combining blocking & pre-transposed B (or col-major B)



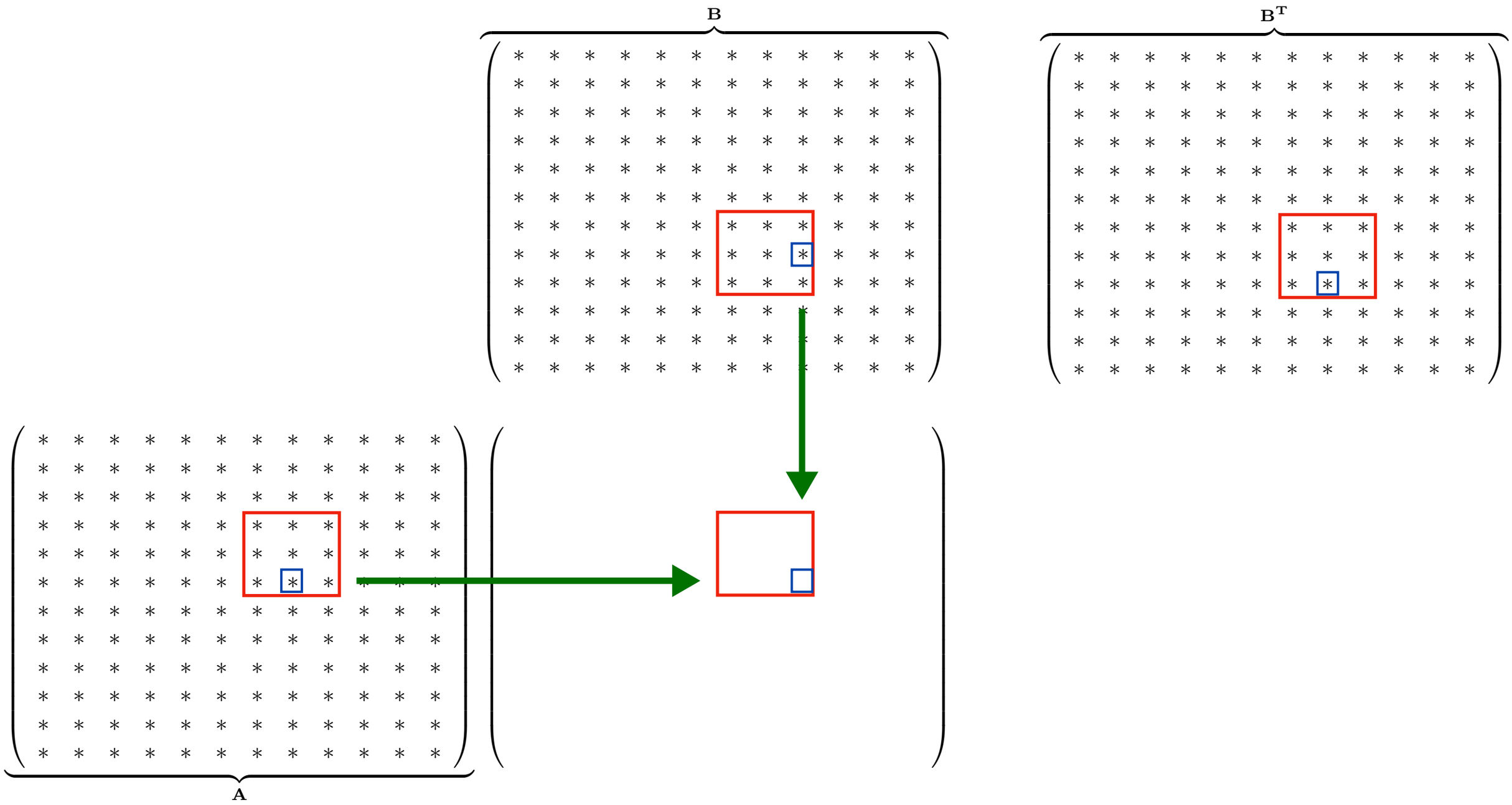
Combining blocking & pre-transposed B (or col-major B)



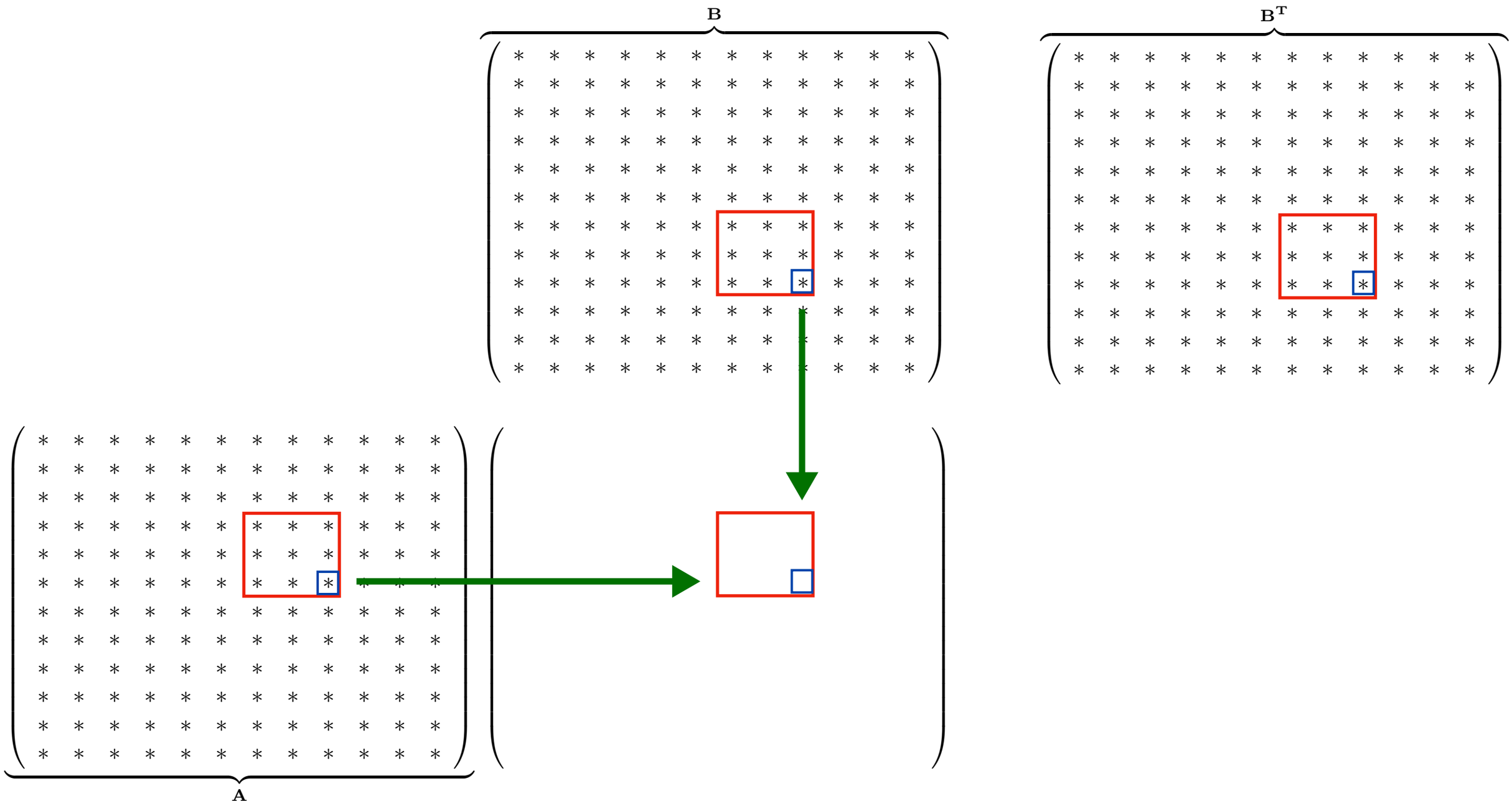
Combining blocking & pre-transposed B (or col-major B)



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



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



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],
    [...]
#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];
        }
}
```

We use local matrices, to cache the data for the “inner multiplication”, one such buffer for each thread

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];
        }
}
```

Data synchronized to "master" copies of matrices before & after the block multiplication operation

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 focus of the operation shifts on how the inner (block) multiplication is performed

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++) {
                loc
                lo
            for (i
            for (i
#pragma omp simd a
                fo
            for (i
            for (i
                bl
        }
}
```

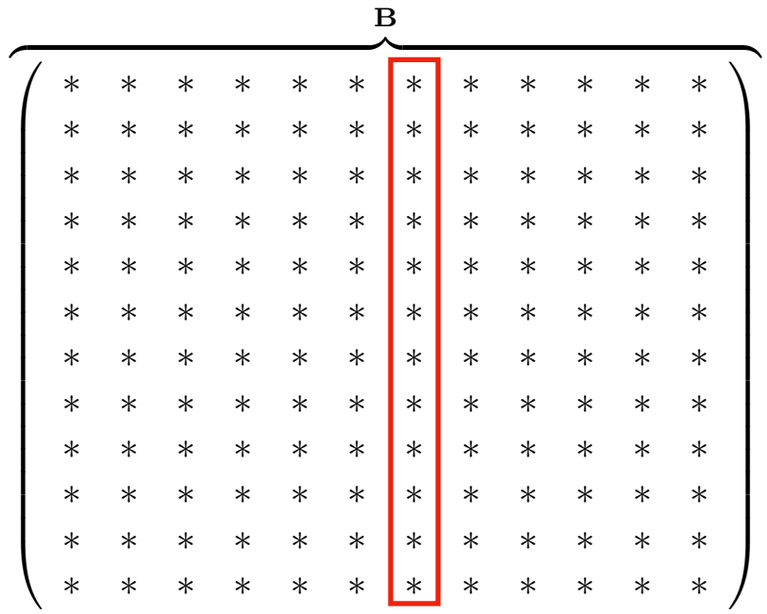
Matrix size 2048 x 2048

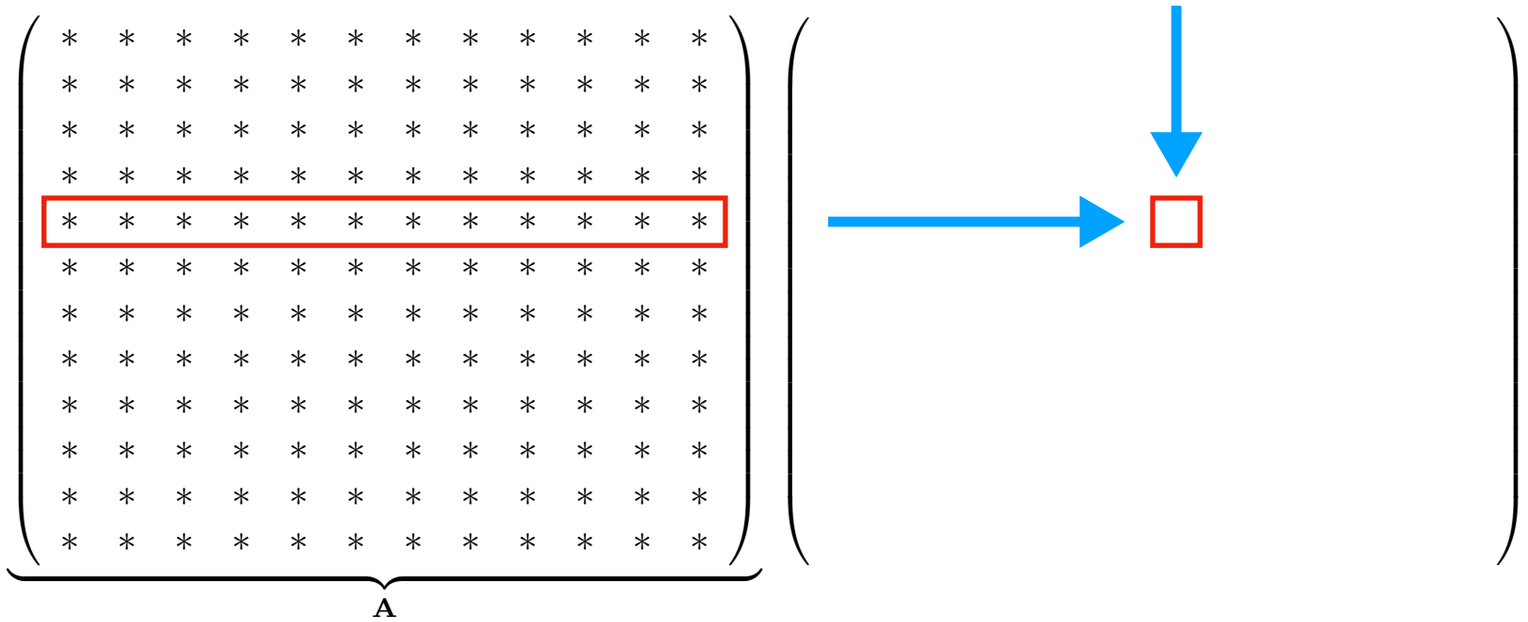
Execution:

```
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]
  [...]
```

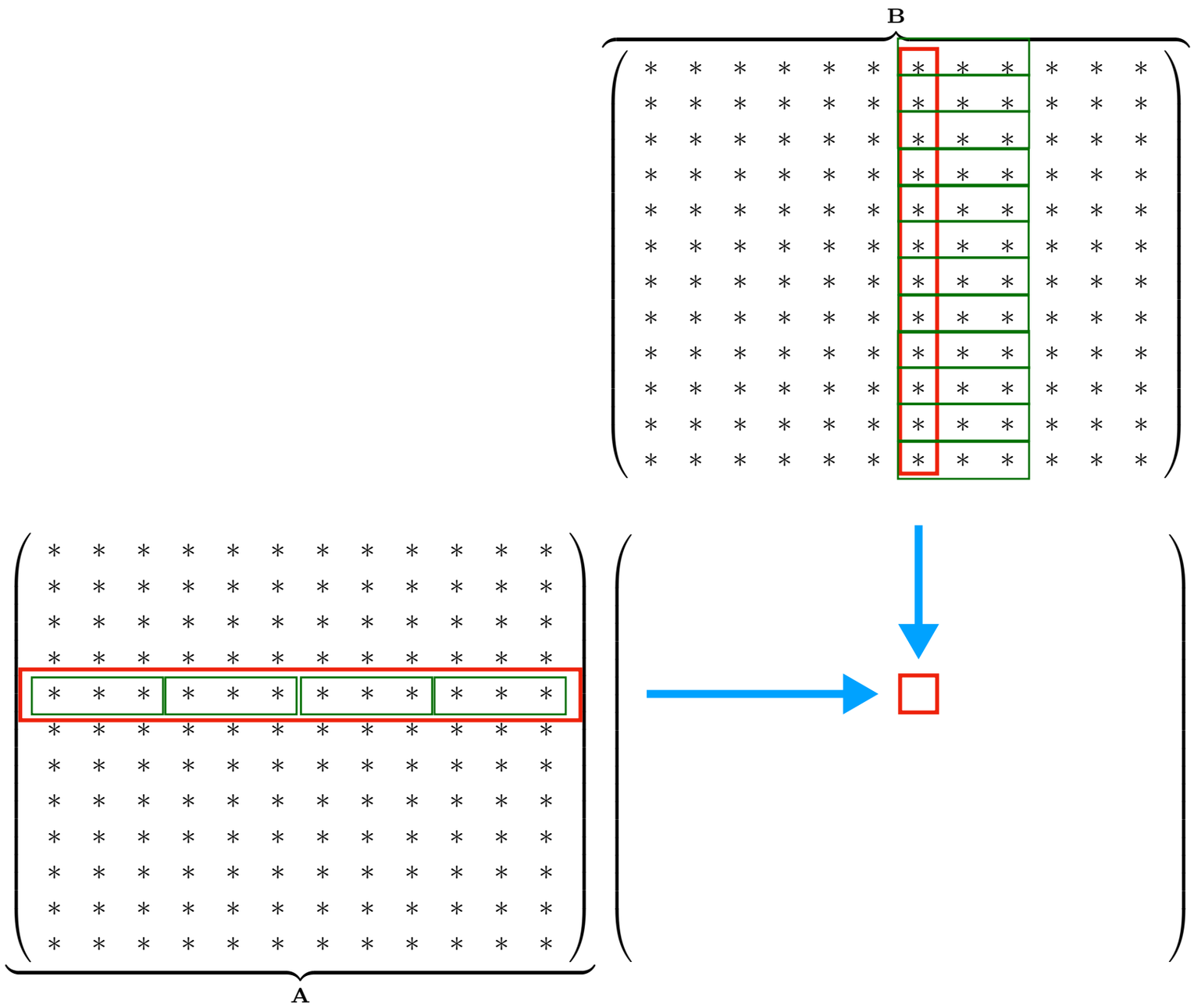
Causes of slowdown

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





Causes of slowdown



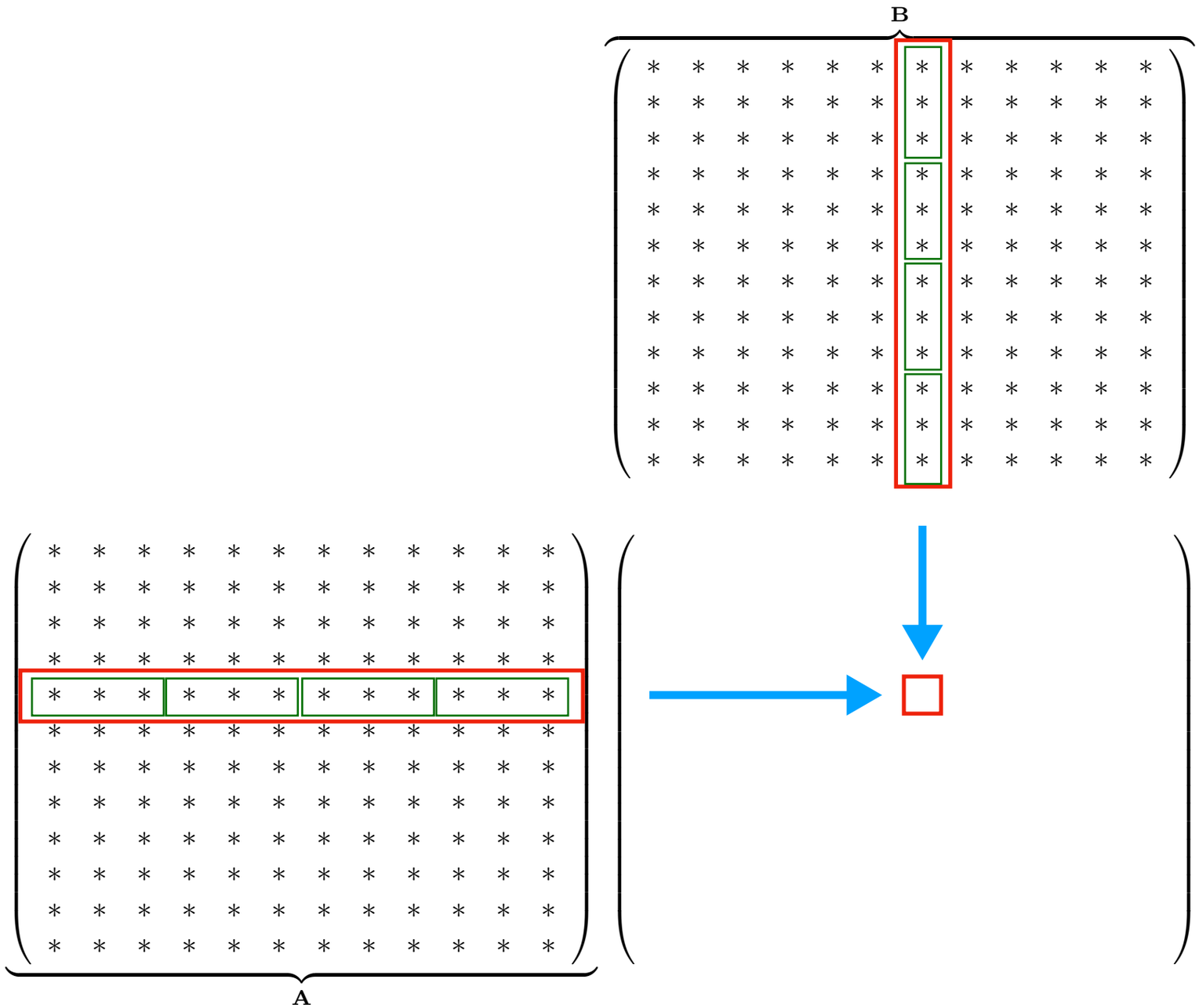
$$
\begin{array}{l}
\texttt{for } i = 1 \ldots N \\
\quad \texttt{for } j = 1 \ldots N \\
\qquad C_{ij} \leftarrow 0 \\
\qquad \texttt{for } k = 1 \ldots N \\
\qquad\quad C_{ij} \leftarrow C_{ij} + A_{ik} B_{kj}
\end{array}
$$

Shapes of cache lines (for row-major matrices)

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

Causes of slowdown

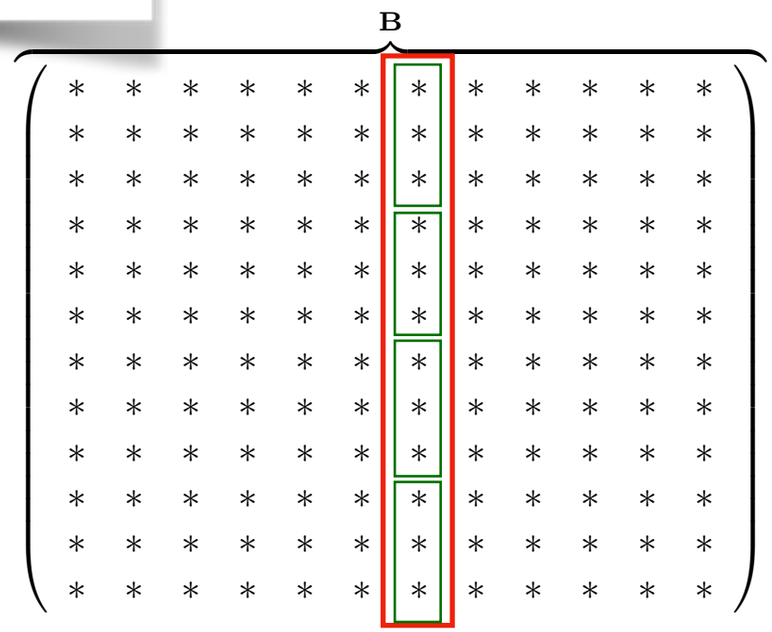
$$
\begin{array}{l}
\texttt{for } i = 1 \ldots N \\
\quad \texttt{for } j = 1 \ldots N \\
\qquad C_{ij} \leftarrow 0 \\
\qquad \texttt{for } k = 1 \ldots N \\
\qquad\quad C_{ij} \leftarrow C_{ij} + A_{ik} B_{kj}
\end{array}
$$

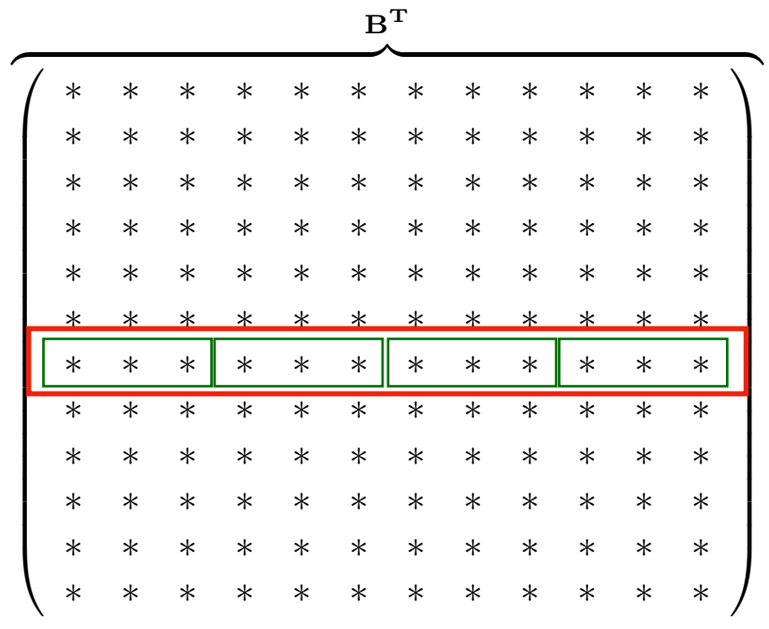


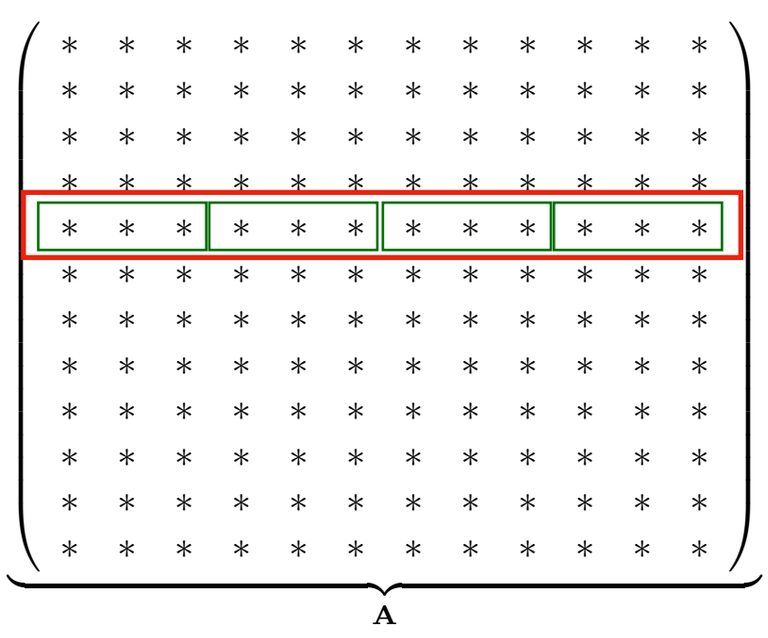
*If, instead, **B** was given as column-major …*

… cache lines are more effectively utilized

Using transpose …

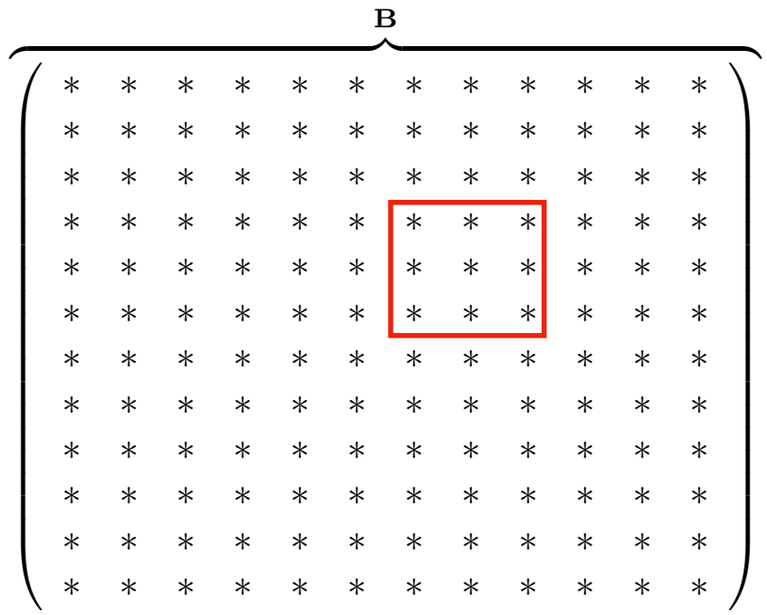


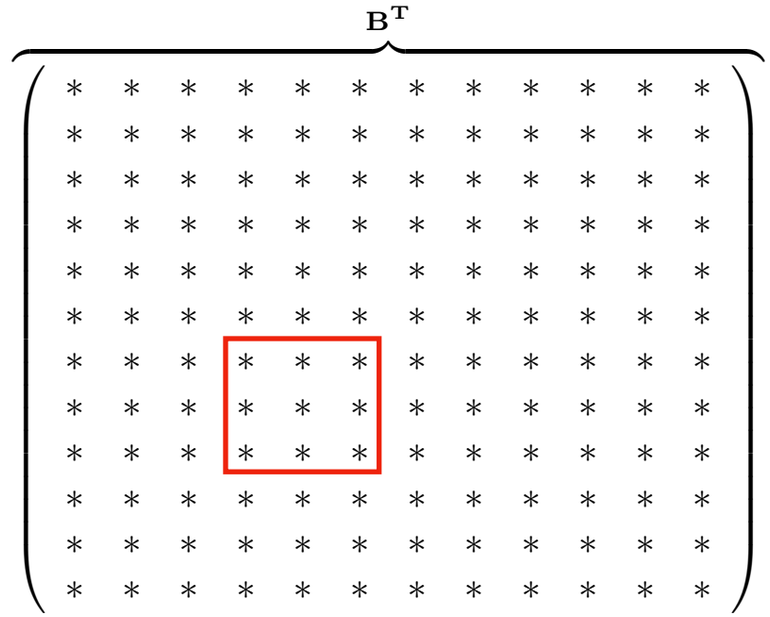


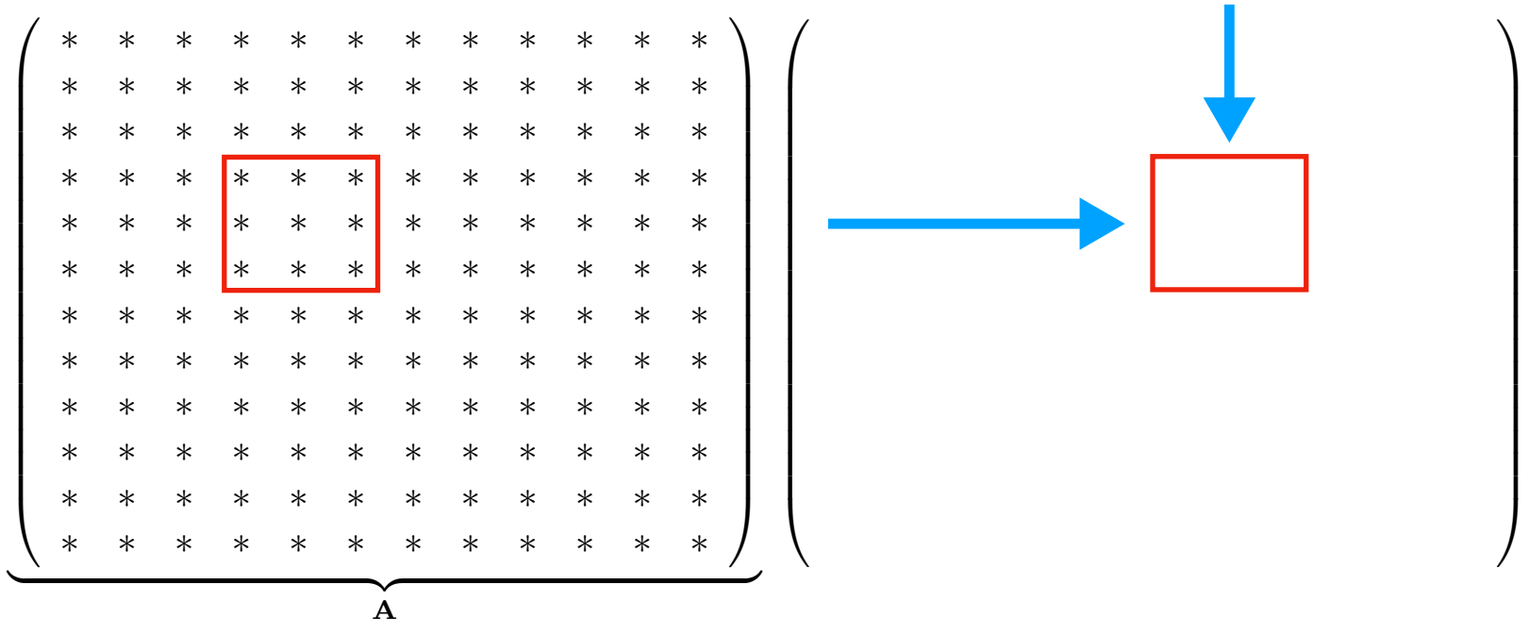



```
for i = 1...N
    for j = 1...N
        C_ij ← 0
        for k = 1...N
            C_ij ← C_ij + A_ik [B^T]_jk
```

Is that still a problem with blocking?



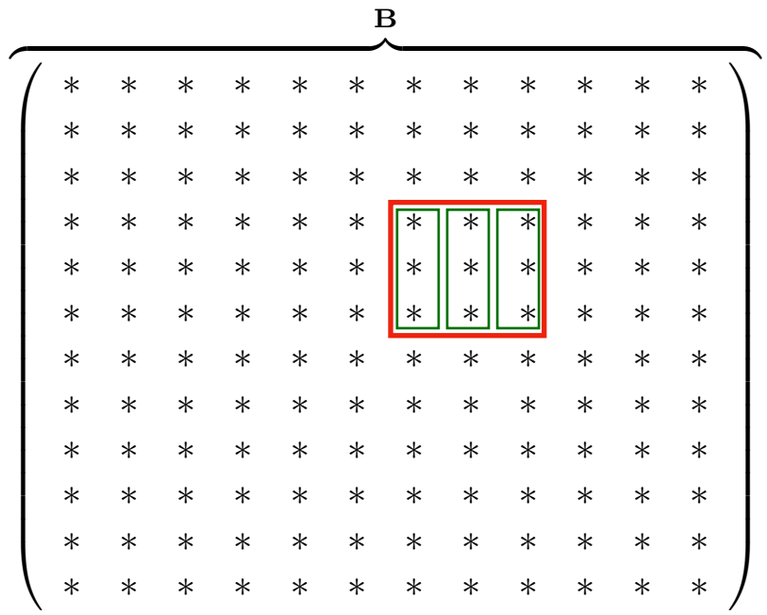


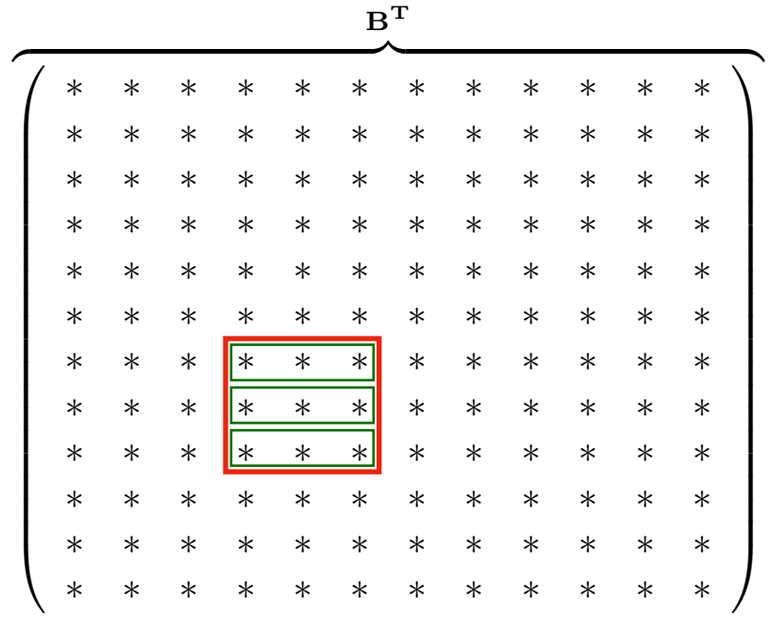


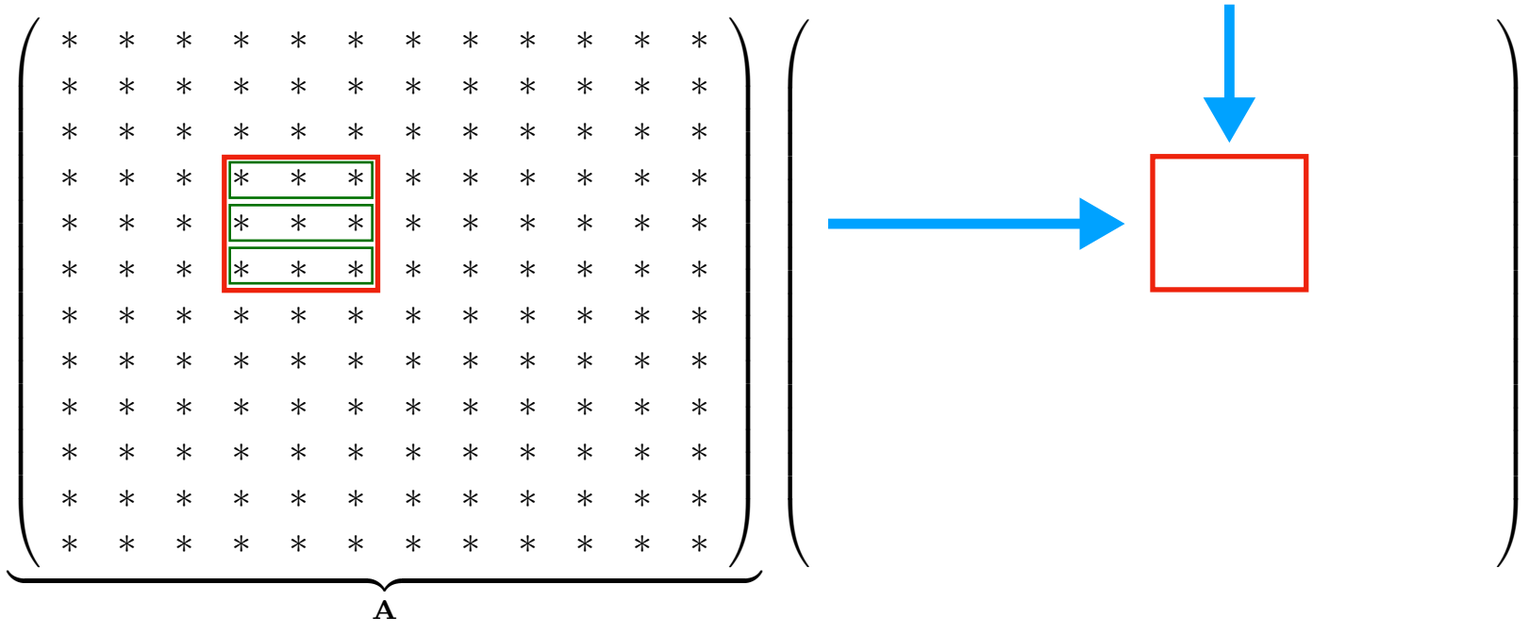
$$
\begin{array}{l}
\texttt{for } i = 1 \ldots N \\
\quad \texttt{for } j = 1 \ldots N \\
\quad\quad C_{ij} \leftarrow 0 \\
\quad\quad \texttt{for } k = 1 \ldots N \\
\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik}[B^T]_{jk}
\end{array}
$$

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

Is that still a problem with blocking?



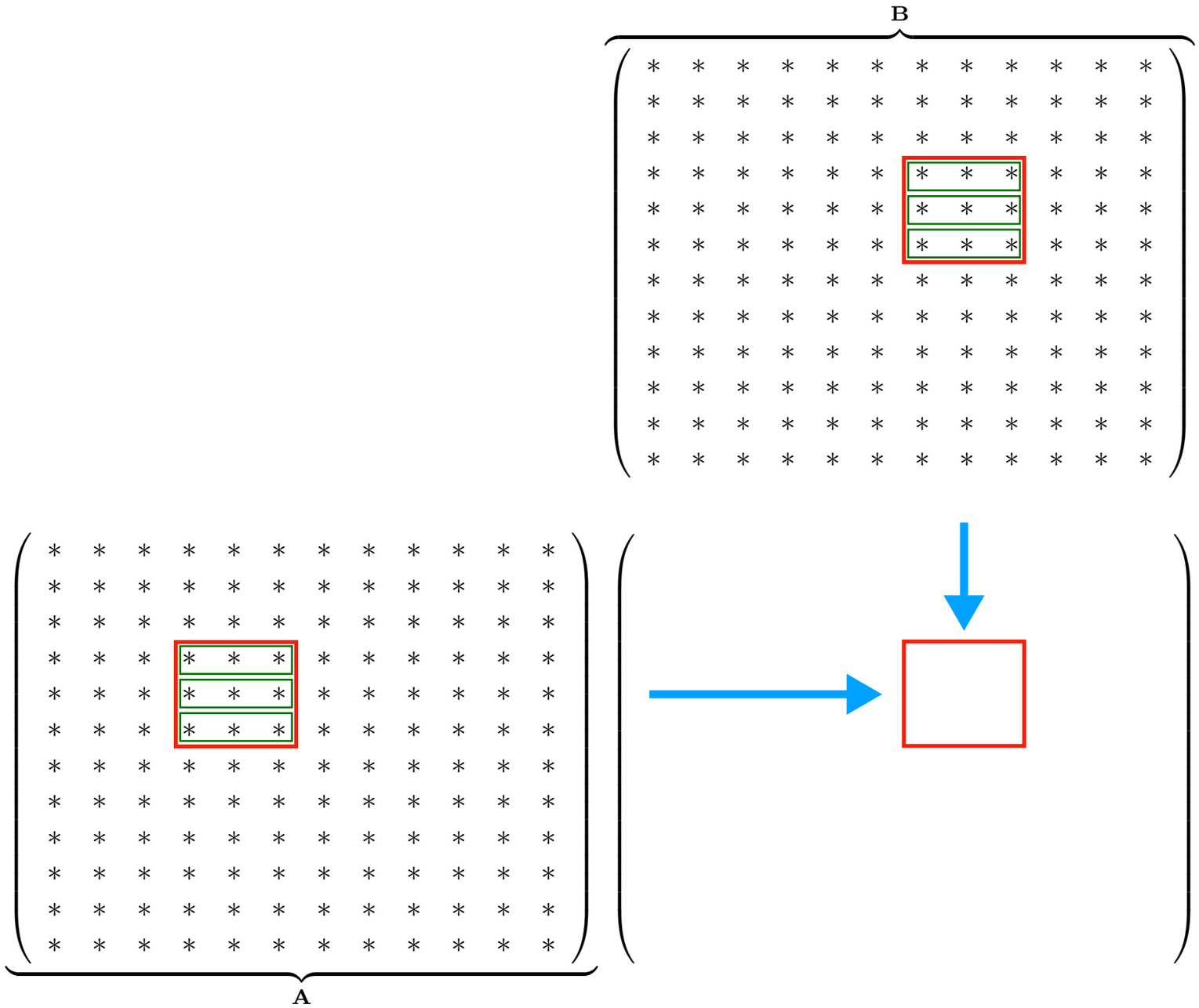





```
for i = 1...N
    for j = 1...N
        C_ij ← 0
        for k = 1...N
            C_ij ← C_ij + A_ik [B^T]_jk
```

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

Blocks larger than cache lines: no need to transpose




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

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

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_4

```
[...]

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

    InitializeMatrices(A, B);
    Timer timer;

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

    [...]
}
```

Build the matrix B^T in advance …

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_9

```
[...]

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));
    [...]
    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;

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

... no longer needed

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_9

```
[...]

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));
    [...]
    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;

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

*Multiply with (non transposed) **B** here …*

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_9

```
[...]

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 : ");
```

... and here ...

```
    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;
}
```

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;
}
```

... as opposed to what we did before

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
    );
}

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)

void MatMatTransposeMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
[...]
}
```

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
    );
}

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)

void MatMatTransposeMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
[...]
}
```

No longer needed …

DenseAlgebra/GEMM_Test_0_9

Multiply w/Transpose (MatMatMultiply.cpp)

```
[...]

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)

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
[...]
}
```

Multiply w/Transpose (MatMatMultiply.cpp)

```
void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]) {
   [...] // Code that recasts A,B & C into blocked blockA, blockB & block C
#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 ii = 0; ii < BLOCK_SIZE; ii++) for (int jj = 0; jj < BLOCK_SIZE; jj++)
            localC[ii][jj] = 0.;

        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[bk][ii][bj][jj];}

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                for (int kk = 0; kk < BLOCK_SIZE; kk++)
                    localC[ii][jj] += localA[ii][kk] * localB[kk][jj];
        }

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

Multiply w/Transpose (MatMatMultiply.cpp)

```
void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]) {
   [...] // Code that recasts A,B & C into blocked blockA, blockB & block C
#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 ii = 0; ii < BLOCK_SIZE; ii++) for (int jj = 0; jj < BLOCK_SIZE; jj++)
            localC[ii][jj] = 0.;

        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[bk][ii][bj][jj];}

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                for (int kk = 0; kk < BLOCK_SIZE; kk++)
                    localC[ii][jj] += localA[ii][kk] * localB[kk][jj];
        }

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

Reverted to (non-transposed) multiply

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];
        }
}
```

… as compared to this

Multiply w/Transpose (MatMatMultiply.cpp)

```
void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]) {
   [...] // Code that recasts A,B & C into blocked blockA, blockB & block C
#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 ii = 0; ii < BLOCK_SIZE; ii++) for (int jj = 0; jj < BLOCK_SIZE; jj++)
            localC[ii][jj] = 0.;

        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[bk][ii][bj][jj];}

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                for (int kk = 0; kk < BLOCK_SIZE; kk++)
                    localC[ii][jj] += localA[ii][kk] * localB[kk][jj];
        }

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

Reading block from (non-transposed B) …

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];
        }
}
```

... as compared to this

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];
        }
}
```

Final modification:
*We were zeroing out **localC** at every iteration of **bk**,*
*and adding back to **blockC** at every such iteration*

Multiply w/Transpose (MatMatMultiply.cpp)

```
void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]) {
   [...] // Code that recasts A,B & C into blocked blockA, blockB & block C
#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 ii = 0; ii < BLOCK_SIZE; ii++) for (int jj = 0; jj < BLOCK_SIZE; jj++)
            localC[ii][jj] = 0.;

        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[bk][ii][bj][jj];}

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                for (int kk = 0; kk < BLOCK_SIZE; kk++)
                    localC[ii][jj] += localA[ii][kk] * localB[kk][jj];
        }

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

Instead do this:
*Zero out **localC** before iterating over **bk**, add back to **blockC** at the end of the loop*

Multiply w/Transpose (MatMatMultiply.cpp)

```
void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]) {
   [...] // Code that recasts A,B & C into blocked blockA, blockB & block C
#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 ii = 0; ii < BLOCK_SIZE; ii++) for (int jj = 0; jj < BLOCK_SIZE; jj++)
            localC[ii][jj] = 0.;

        for (int bk = 0; bk < NBLOCKS; bk++) {
            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (i
                lo
                lo
            for (i
            for (i
                fo
        }
        for (int i
            blockC
    }
}
```

Size = 2048 (no transposition)

Execution:

```
Running candidate kernel for correctness test ... [Elapsed time : 51.8363ms]
Running reference kernel for correctness test ... [Elapsed time : 39.8164ms]
Discrepancy between two methods : 0.000164032
Running kernel for performance run # 1 ... [Elapsed time : 48.7371ms]
Running kernel for performance run # 2 ... [Elapsed time : 45.8482ms]
Running kernel for performance run # 3 ... [Elapsed time : 45.6323ms]
Running kernel for performance run # 4 ... [Elapsed time : 45.5578ms]
Running kernel for performance run # 5 ... [Elapsed time : 45.5312ms]
Running kernel for performance run # 6 ... [Elapsed time : 45.5392ms]
Running kernel for performance run # 7 ... [Elapsed time : 45.5347ms]
Running kernel for performance run # 8 ... [Elapsed time : 45.5578ms]
[...]
```

Multiply w/Transpose (MatMatMultiply.cpp)

```
void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]) {
   [...] // Code that recasts A,B & C into blocked blockA, blockB & block C
#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 ii = 0; ii < BLOCK_SIZE; ii++) for (int jj = 0; jj < BLOCK_SIZE; jj++)
            localC[ii][jj] = 0.;

        for (int bk = 0; bk < NBLOCKS; bk++)
            for (int ii = 0; ii < BLOCK_SIZ
            for (int jj = 0; jj < BLOCK_SIZ
                localA[ii][jj] = blockA[bi]
                localB[ii][jj] = blockB[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++)
                    localC[ii][jj] += localA[ii][kk] * localB[kk][jj];
        }

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

Remaining issues:
*Do we need to **add** to "master" blockC?*
(and do we need to initialize the matrix to zero?)

Multiply w/Transpose (MatMatMultiply.cpp)

```
void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]) {
   [...] // Code that recasts A,B & C into blocked blockA, blockB & block C
#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 ii = 0; ii < BLOCK_SIZE; ii++) for (int jj = 0; jj < BLOCK_SIZE; jj++)
            localC[ii][jj] = 0.;

        for (int bk = 0; bk < NBLOCKS; bk++)
            for (int ii = 0; ii < BLOCK_SIZ
            for (int jj = 0; jj < BLOCK_SIZ
                localA[ii][jj] = blockA[bi]
                localB[ii][jj] = blockB[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++)
                    localC[ii][jj] += localA[ii][kk] * localB[kk][jj];
        }

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

Remaining issues:
*Do we need to **add** to "master" blockC?*
(and do we need to initialize the matrix to zero?)

Multiply w/Transpose (MatMatMultiply.cpp)

```
void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]) {
   [...] // Code that recasts A,B & C into blocked blockA, blockB & block C
#pragma omp parallel for
    for (int bi = 0; bi < NBLOCKS; bi++)
    for (int bj = 0; bj < NBLOCKS; bj++) {

        for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                localC[ii][jj] = 0.;

        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[bk][ii][bj][jj]; }

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

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

Multiply w/Transpose (MatMatMultiply.cpp)

```
void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]) {
   [...] // Code that recasts A,B & C into blocked blockA, blockB & block C
#pragma omp parallel for
    for (int bi = 0; bi < NBLOCKS; bi++)
    for (int bj = 0; bj < NBLOCKS; bj++) {

        for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                localC[ii][jj] = 0.;

        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[bk][ii][bj][jj]; }

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
                for (int kk = 0; kk < BLOCK_SIZE; kk
#pragma omp simd
                    for (int jj = 0; jj < BLOCK_SIZE
                    localC[ii][jj] += localA[ii][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];
    }
}
```

*Simply "set" the corresponding block in **blockC** to the right result at the end (no need to initialize to zero)*

Multiply w/Transpose (MatMatMultiply.cpp)

```
void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]) {
   [...] // Code that recasts A,B & C into blocked blockA, blockB & block C
#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 ii = 0; ii < BLOCK_SIZE; ii++) for (int jj = 0; jj < BLOCK_SIZE; jj++)
            localC[ii][jj] = 0.;

        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[bk][ii][bj][jj];}

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                for (int kk = 0; kk < BLOCK_SIZE; kk++)
                    localC[ii][jj] += localA[ii][kk] * localB[kk][jj];
        }

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

Are we generating opportunities for SIMD execution?

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];
        }
}
```

Previously, this was an opportunity for SIMD execution (data was laid out linearly in memory)

Multiply w/Transpose (MatMatMultiply.cpp)

```
void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]) {
   [...] // Code that recasts A,B & C into blocked blockA, blockB & block C
#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 ii = 0; ii < BLOCK_SIZE; ii++) for (int jj = 0; jj < BLOCK_SIZE; jj++)
            localC[ii][jj] = 0.;

        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][
                localB[ii][jj] = blockB[bk][ii][bj][

            for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                for (int kk = 0; kk < BLOCK_SIZE; kk++)
                    localC[ii][jj] += localA[ii][kk] * localB[kk][jj];
        }

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

*Only **localA** is laid out linearly in memory (**localB** is cached; so not too bad …)*

Multiply w/Transpose (MatMatMultiply.cpp)

```
void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]) {
   [...] // Code that recasts A,B & C into blocked blockA, blockB & block C
#pragma omp parallel for
    for (int bi = 0; bi < NBLOCKS; bi++)
    for (int bj = 0; bj < NBLOCKS; bj++) {

        for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                localC[ii][jj] = 0.;

        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][
                localB[ii][jj] = blockB[bk][ii][bj][jj]; }

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

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

*Swapped order of **jj** and **kk** loops (perfectly allowable; operation is the same)*

Multiply w/Transpose (MatMatMultiply.cpp)

```
void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]) {
   [...] // Code that recasts A,B & C into blocked blockA, blockB & block C
#pragma omp parallel for
    for (int bi = 0; bi < NBLOCKS; bi++)
    for (int bj = 0; bj < NBLOCKS; bj++) {

        for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                localC[ii][jj] = 0.;

        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[bk][ii][bj][jj];

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

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

Restored regularity of computation
Code is more SIMD-friendly

DenseAlgebra/GEMM_Test_0_10

Multiply w/Transpose (MatMatMultiply.cpp)

```
void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]) {
   [...] // Code that recasts A,B & C into blocked blockA, blockB & block C
#pragma omp parallel for
    for (int bi = 0; bi < NBLOCKS; bi++)
    for (int bj = 0; bj < NBLOCKS; bj++) {

        for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                localC[ii][jj] = 0.;

        for (int bk = 0; bk < NBLOCKS; bk++) {
            for (int ii = 0; ii < BLOCK_SIZ
            for (int jj = 0; jj < BLOCK_SIZE; jj++) {
                loc
                loc
            for (in
                for
#pragma omp simd

        }

        for (int ii
        for (int jj
            blockC[
    }
}
```

MATRIX_SIZE=2048 BLOCK_SIZE=32

Execution:

```
Running candidate kernel for correctness test ... [Elapsed time : 48.3319ms]
Running reference kernel for correctness test ... [Elapsed time : 38.1047ms]
Discrepancy between two methods : 0.000152588
Running kernel for performance run # 1 ... [Elapsed time : 36.4563ms]
Running kernel for performance run # 2 ... [Elapsed time : 34.6192ms]
Running kernel for performance run # 3 ... [Elapsed time : 34.3967ms]
Running kernel for performance run # 4 ... [Elapsed time : 34.4863ms]
Running kernel for performance run # 5 ... [Elapsed time : 34.5077ms]
Running kernel for performance run # 6 ... [Elapsed time : 34.6377ms]
Running kernel for performance run # 7 ... [Elapsed time : 34.5291ms]
Running kernel for performance run # 8 ... [Elapsed time : 34.4074ms]
[...]
```

DenseAlgebra/GEMM_Test_0_10

Multiply w/Transpose (MatMatMultiply.cpp)

```
void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]) {
   [...] // Code that recasts A,B & C into blocked blockA, blockB & block C
#pragma omp parallel for
    for (int bi = 0; bi < NBLOCKS; bi++)
    for (int bj = 0; bj < NBLOCKS; bj++) {

        for (int ii = 0; ii < BLOCK_SIZE; ii++)
            for (int jj = 0; jj < BLOCK_SIZE; jj++)
                localC[ii][jj] = 0.;

        for (int bk = 0; bk < NBLOCKS; bk++) {
            for (int ii = 0; ii < BLOCK_SIZ
            for (int jj = 0; jj < BLOCK_SIZE; jj++) {
                loc
                loc
            for (in
                for
#pragma omp simd

        }

        for (int ii
        for (int jj
            blockC[
    }
}
```

*MATRIX_SIZE=2048 BLOCK_SIZE=**64***

Execution:

```
Running candidate kernel for correctness test ... [Elapsed time : 43.3087ms]
Running reference kernel for correctness test ... [Elapsed time : 37.8041ms]
Discrepancy between two methods : 0.000175476
Running kernel for performance run # 1 ... [Elapsed time : 30.9614ms]
Running kernel for performance run # 2 ... [Elapsed time : 30.8871ms]
Running kernel for performance run # 3 ... [Elapsed time : 30.5422ms]
Running kernel for performance run # 4 ... [Elapsed time : 31.0008ms]
Running kernel for performance run # 5 ... [Elapsed time : 30.6835ms]
Running kernel for performance run # 6 ... [Elapsed time : 30.6332ms]
Running kernel for performance run # 7 ... [Elapsed time : 30.7048ms]
Running kernel for performance run # 8 ... [Elapsed time : 30.46ms]
[...]
```