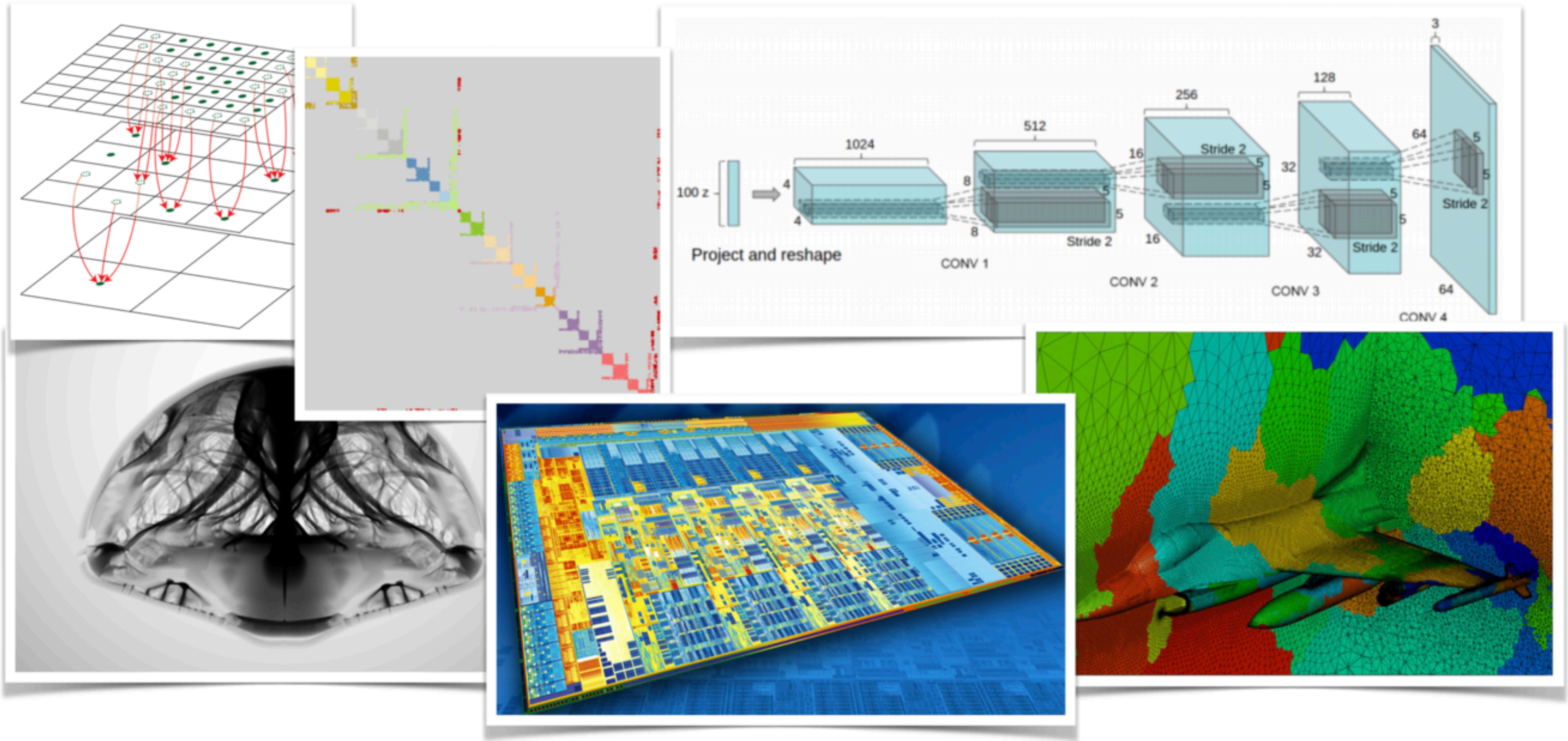


Lecture 12: Continued optimizations on General Matrix-Matrix multiplication (GEMM) - Blocking and SIMD

Thursday March 2nd 2023

Logistics

- Practice midterm to be released by *Saturday*
  (need a bit extra time to incorporate GEMM questions)
- HW#2 due tomorrow
- Practice midterm review on Tuesday.

Quick pointers on getting set up with MKL

- Easiest way to get up-and-running: Lab machines & ICC

Setting up and running ICC:

```
[username@host]$ source /s/intelcompilers-2019/bin/iccvars.sh intel64

[username@host]$ icc --version
icc (ICC) 19.0.5.281 20190815
Copyright (C) 1985-2019 Intel Corporation.  All rights reserved.

[username@host]$ icc *.cpp -qopenmp -mkl
```

- On your local PC:

  - Both Intel Compilers (optional) and Intel MKL are available for free (on all 3 platforms, Linux/Mac/Win); installation might require effort and persistence

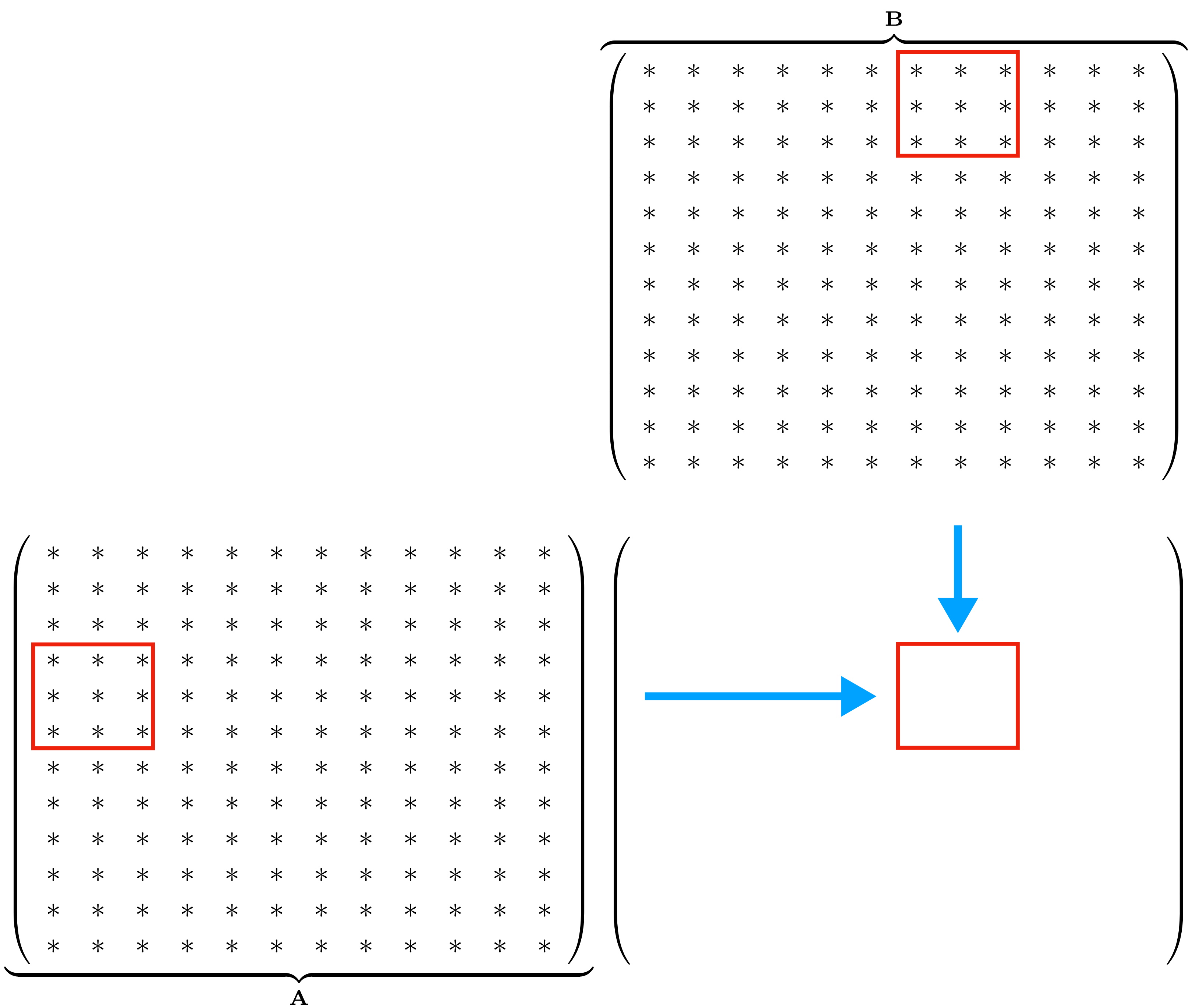
  - Useful links:
    https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html
    https://software.intel.com/content/www/us/en/develop/articles/oneapi-standalone-components.html#onemkl
    (you can find download links for the "Classic" ICC compiler there, too)

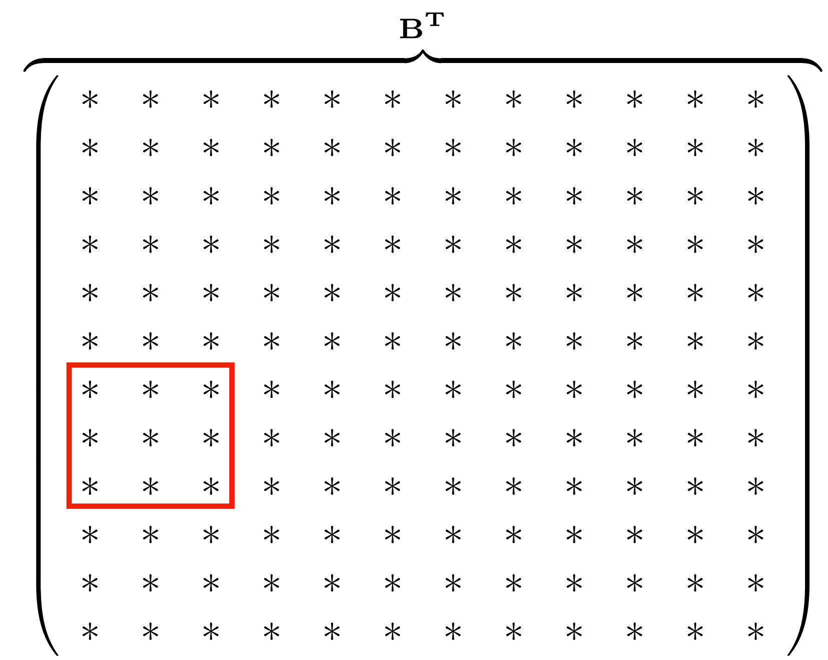
  - Use Piazza! Peer help will be appreciated (and rewarded!)

Today’s lecture

- Continued focus on GEMM operations
- More blocking, and an introduction to more “explicit” SIMD considerations and optimizations
- (you might see some assembly code next week; fear not!)

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

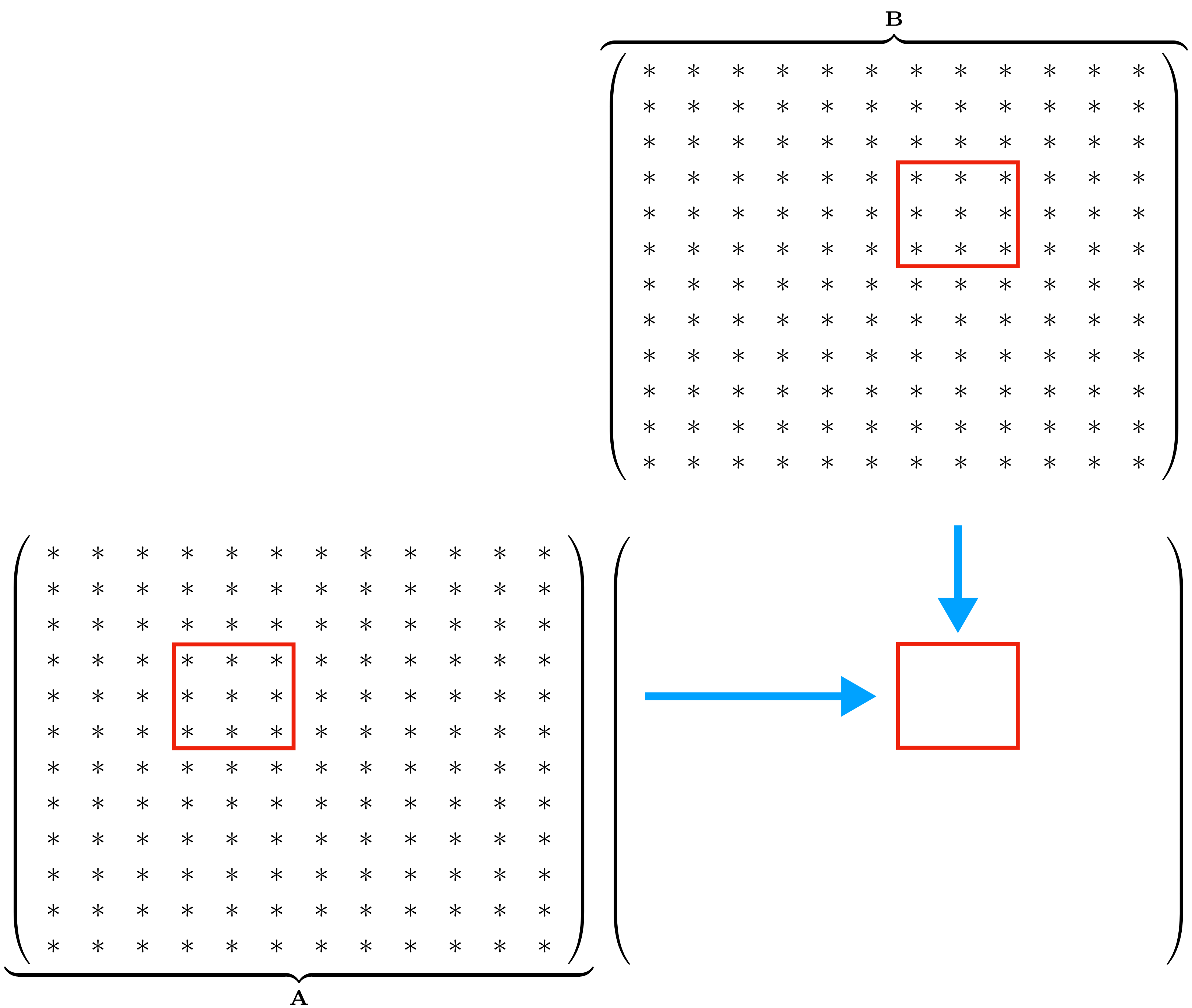


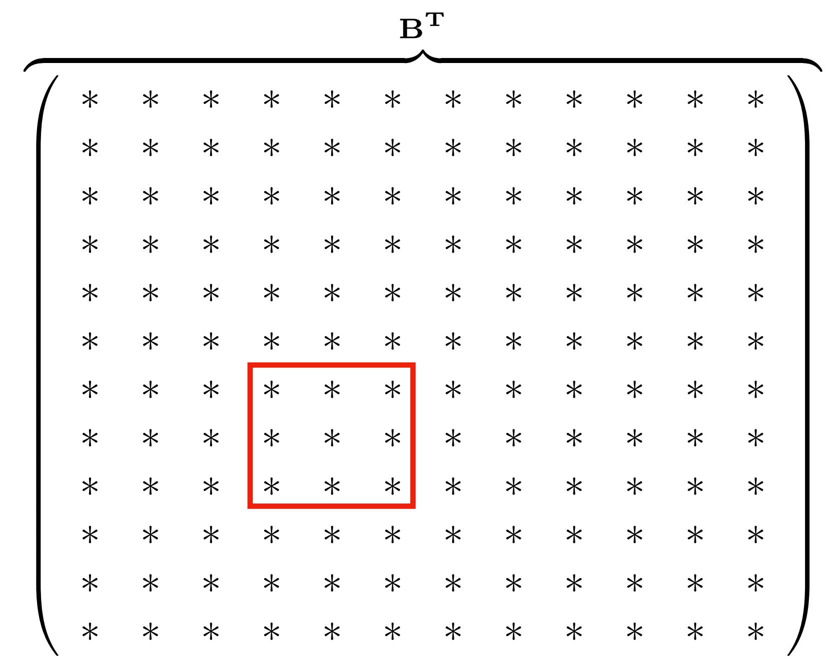


$$
\begin{array}{l}
\texttt{for}\ \ i = 1 \ldots N \\
\quad \texttt{for}\ \ j = 1 \ldots N \\
\quad\quad C_{ij} \leftarrow 0 \\
\quad\quad \texttt{for}\ \ k = 1 \ldots N \\
\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} [B^T]_{jk}
\end{array}
$$

***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 $j = 1 \ldots N$

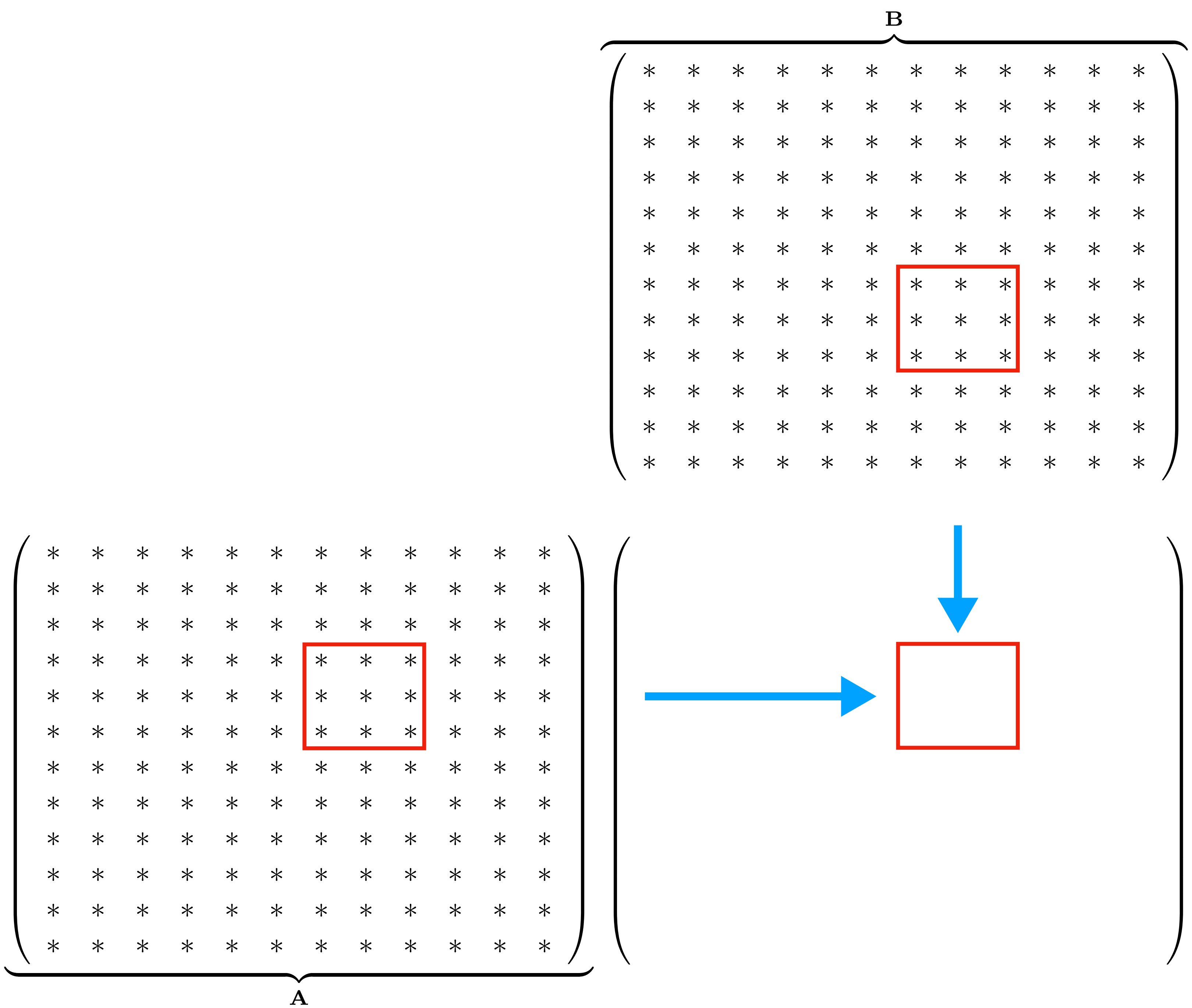
    $C_{ij} \leftarrow 0$

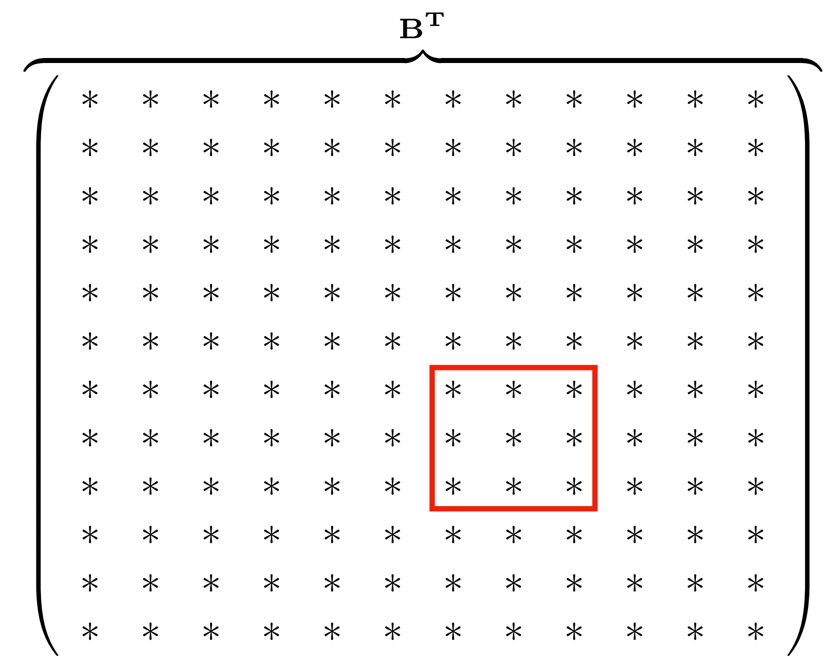
    for $k = 1 \ldots N$

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

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

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

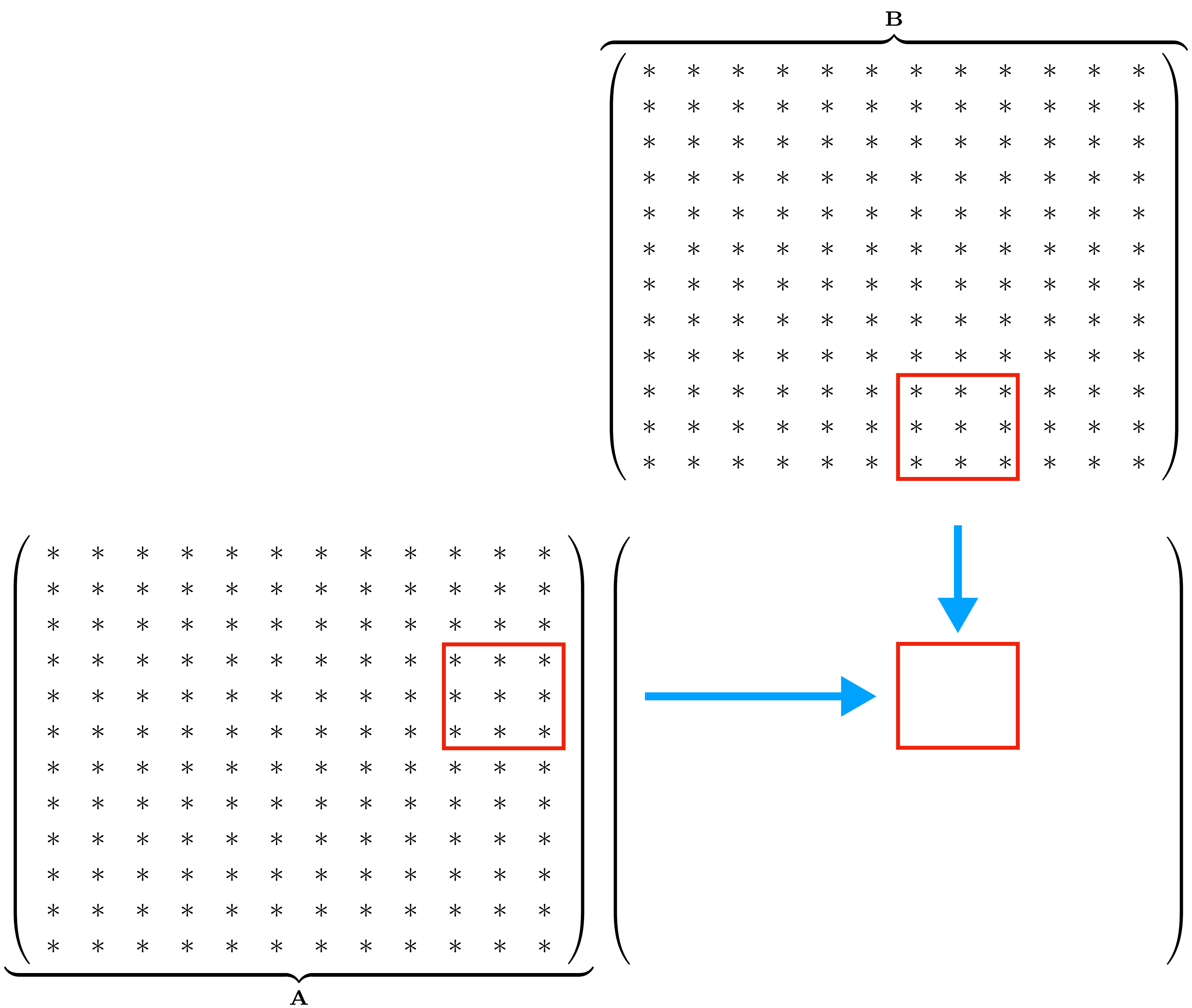


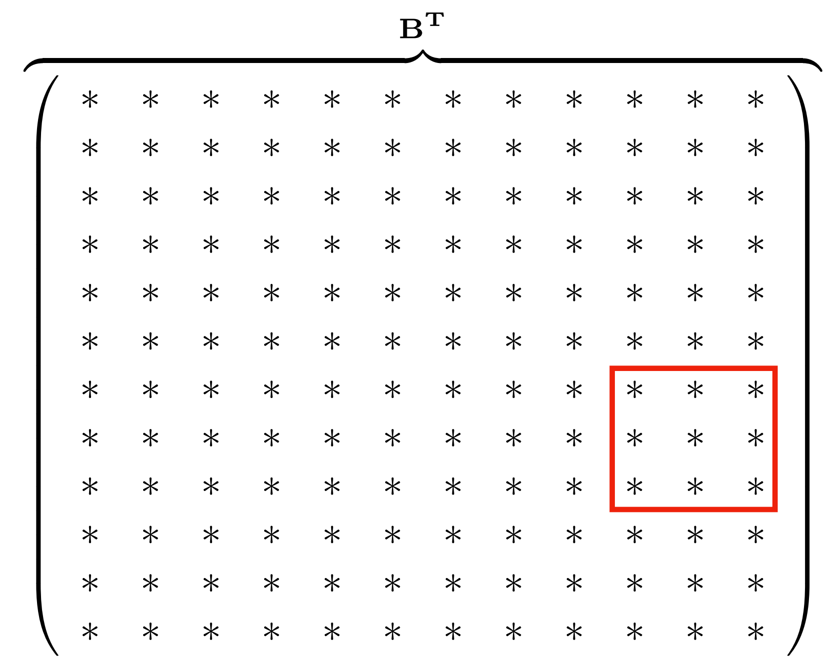


$$
\begin{array}{l}
\texttt{for}\ \ i = 1 \ldots N \\
\quad \texttt{for}\ \ j = 1 \ldots N \\
\quad\quad C_{ij} \leftarrow 0 \\
\quad\quad \texttt{for}\ \ k = 1 \ldots N \\
\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} [B^T]_{jk}
\end{array}
$$

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

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

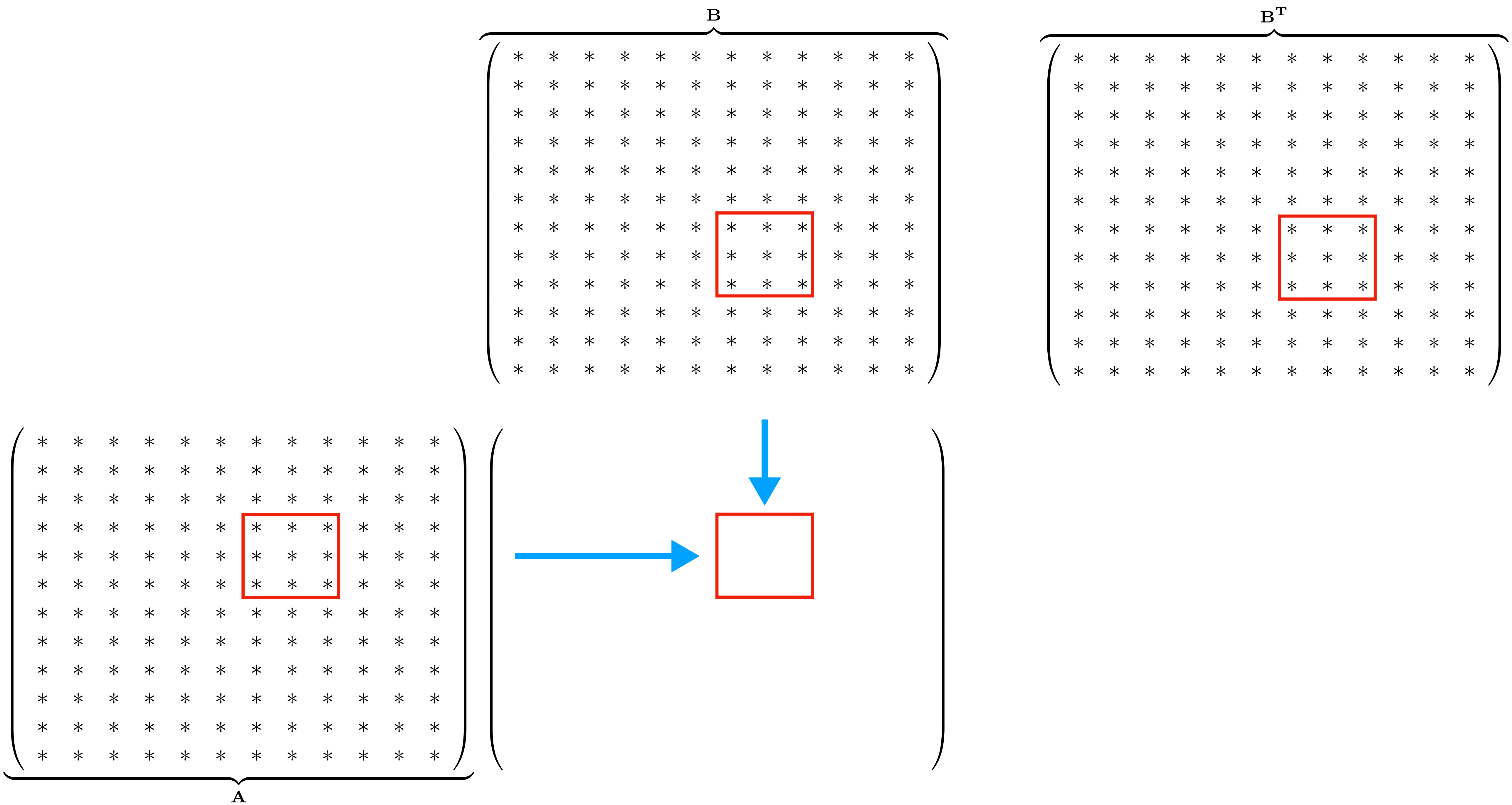




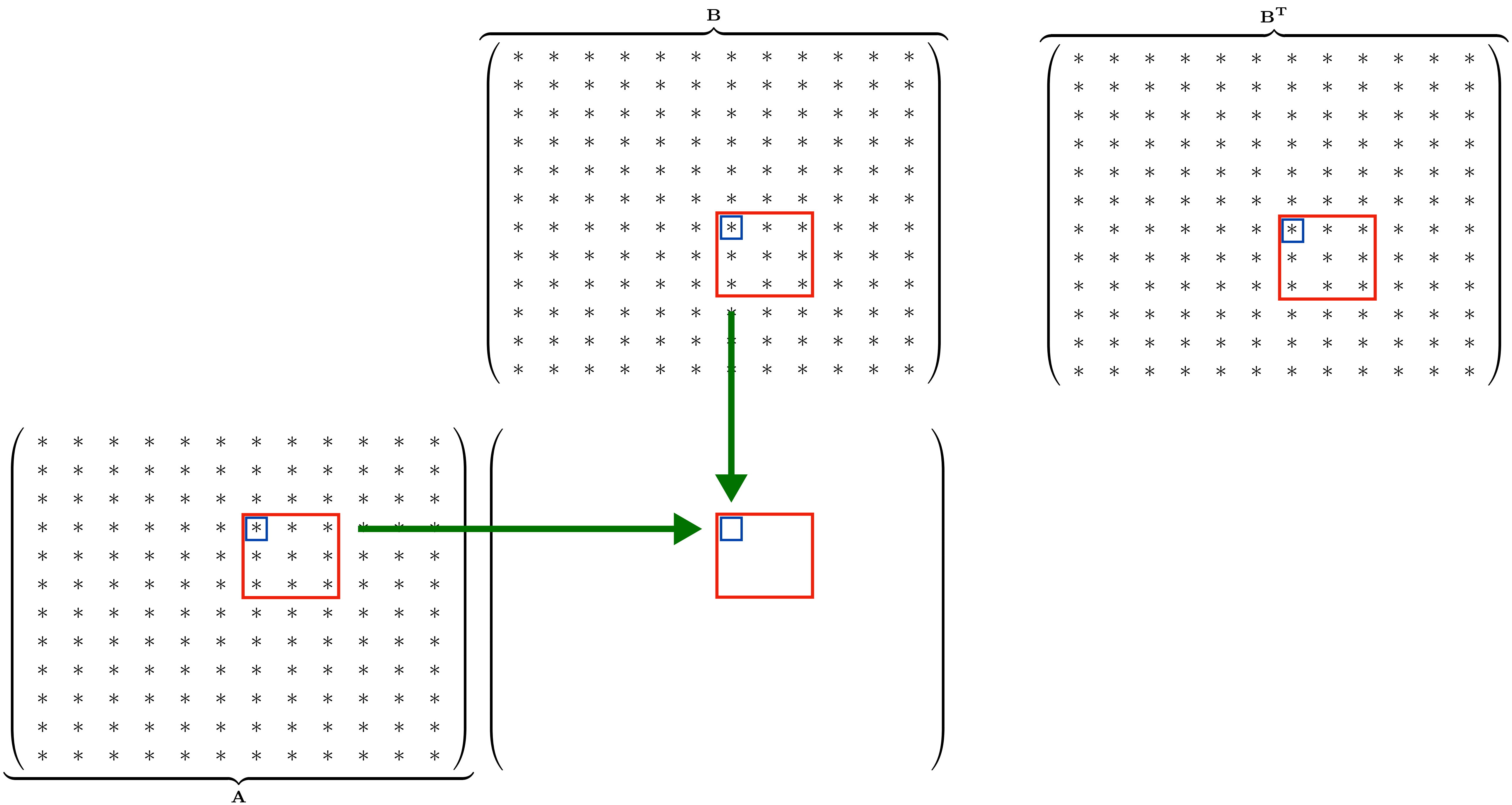
$$
\begin{aligned}
&\texttt{for}\ \ i = 1 \ldots N \\
&\quad \texttt{for}\ \ j = 1 \ldots N \\
&\quad\quad C_{ij} \leftarrow 0 \\
&\quad\quad \texttt{for}\ \ k = 1 \ldots N \\
&\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} [B^T]_{jk}
\end{aligned}
$$

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

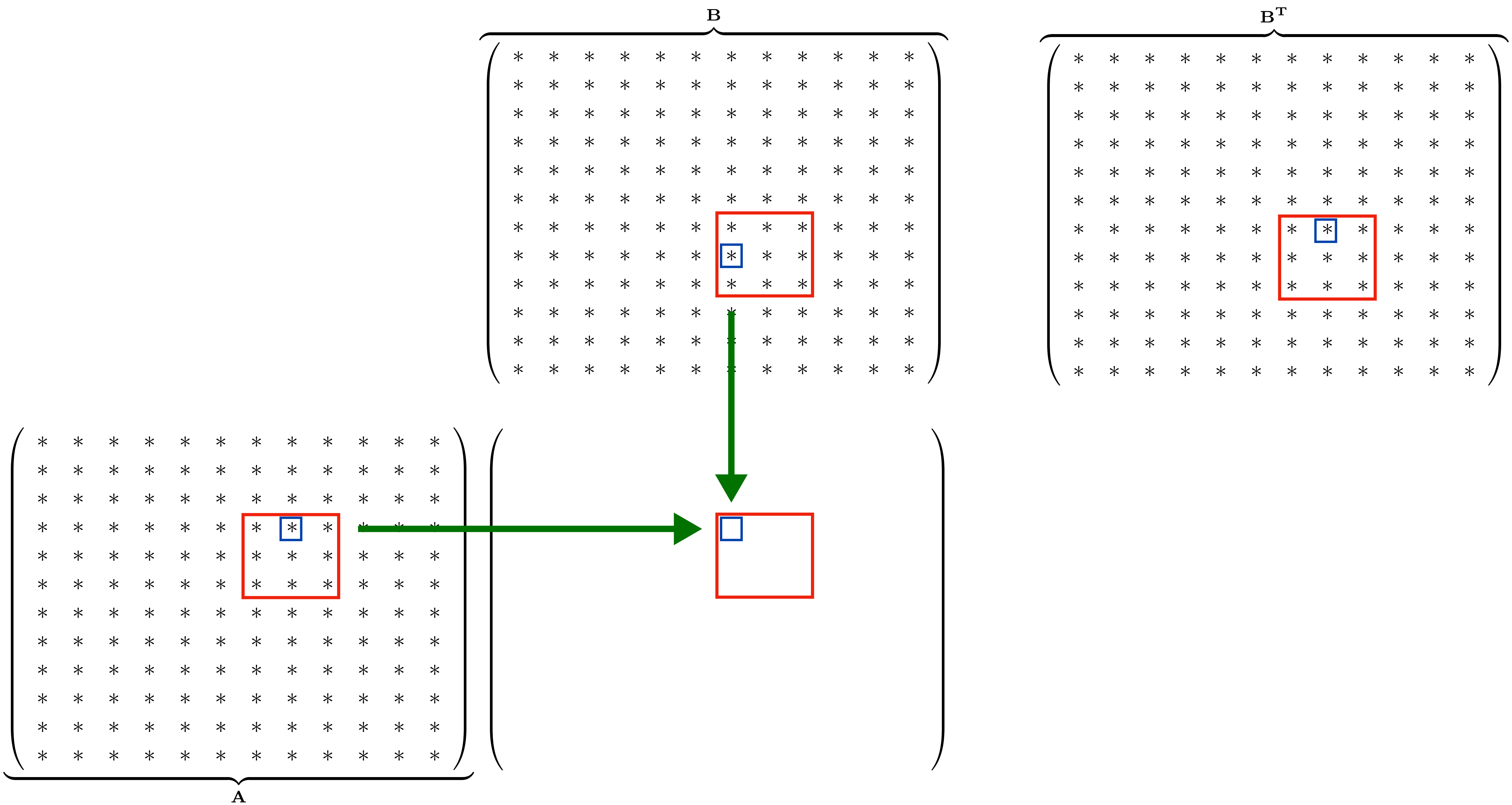
Combining blocking & pre-transposed B (or col-major B)



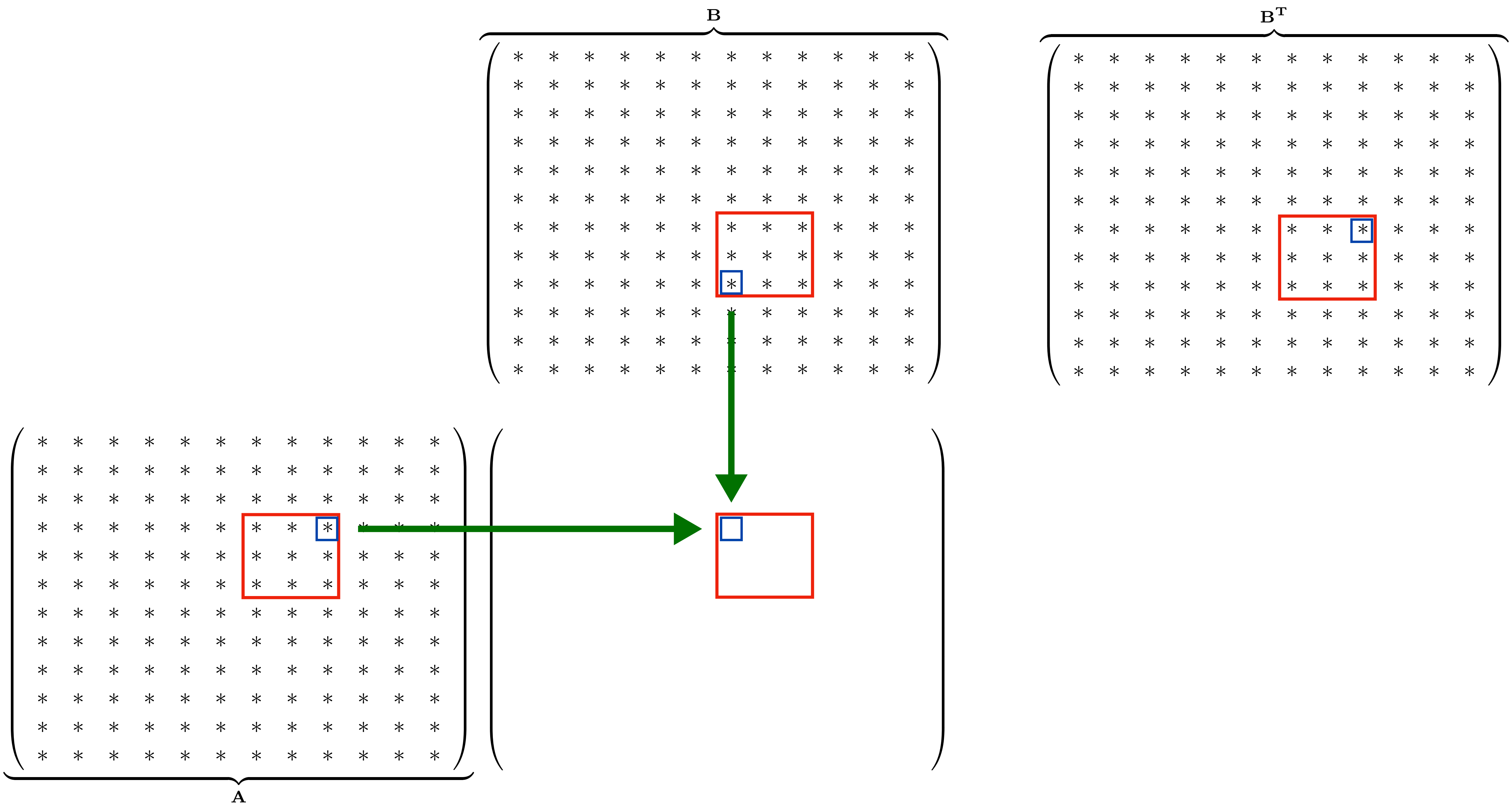
Combining blocking & pre-transposed B (or col-major B)



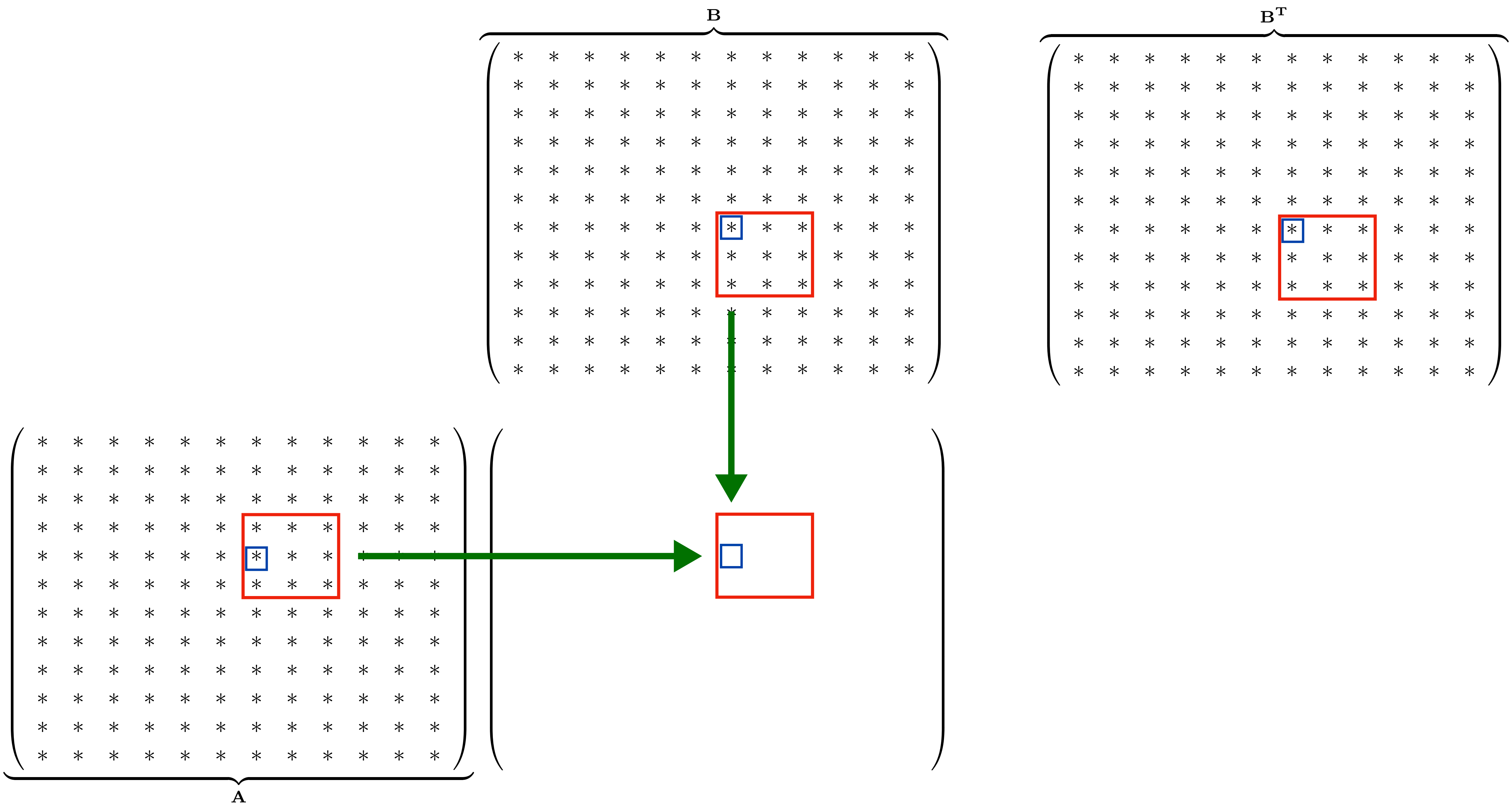
Combining blocking & pre-transposed B (or col-major B)



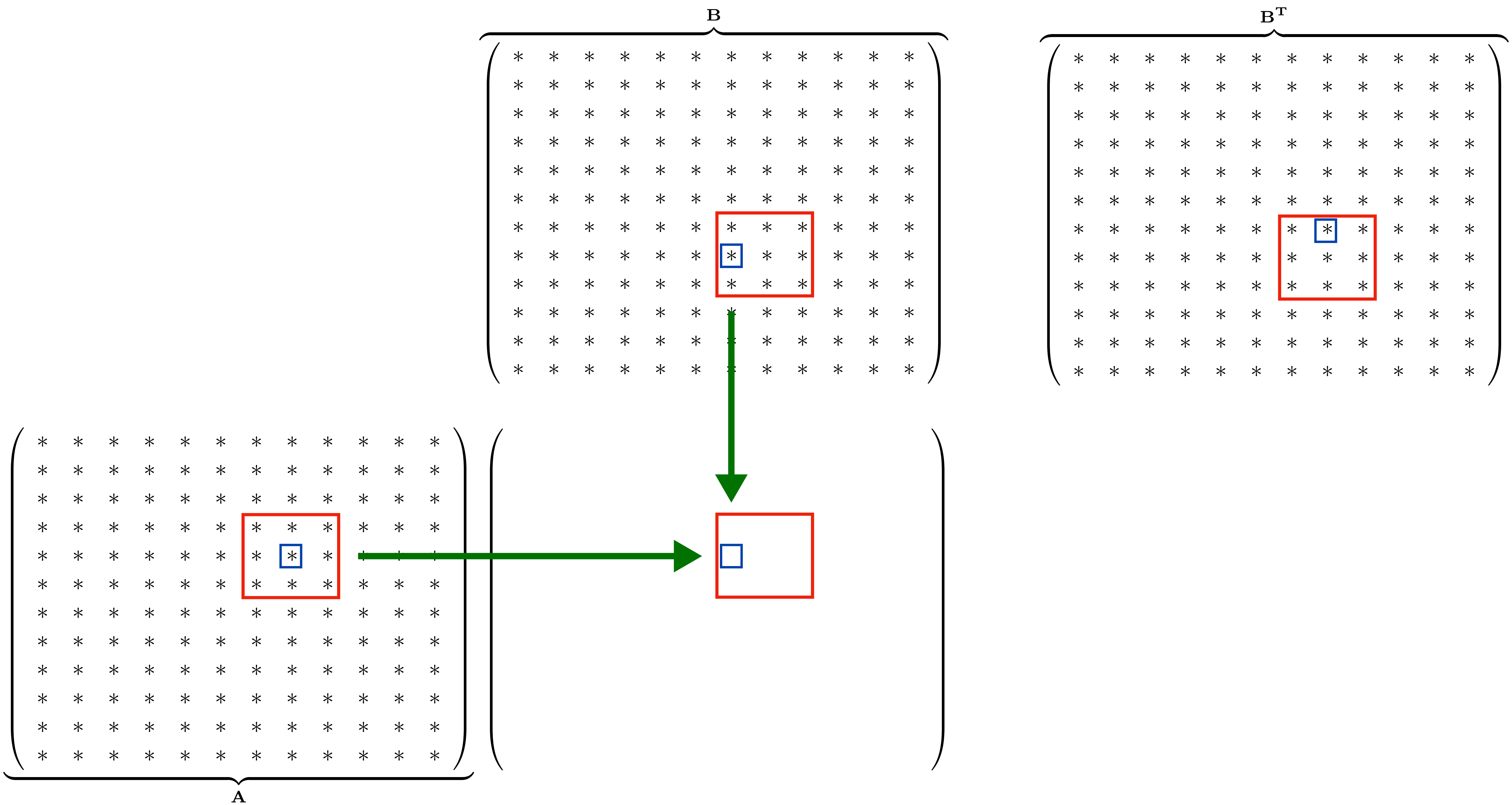
Combining blocking & pre-transposed B (or col-major B)



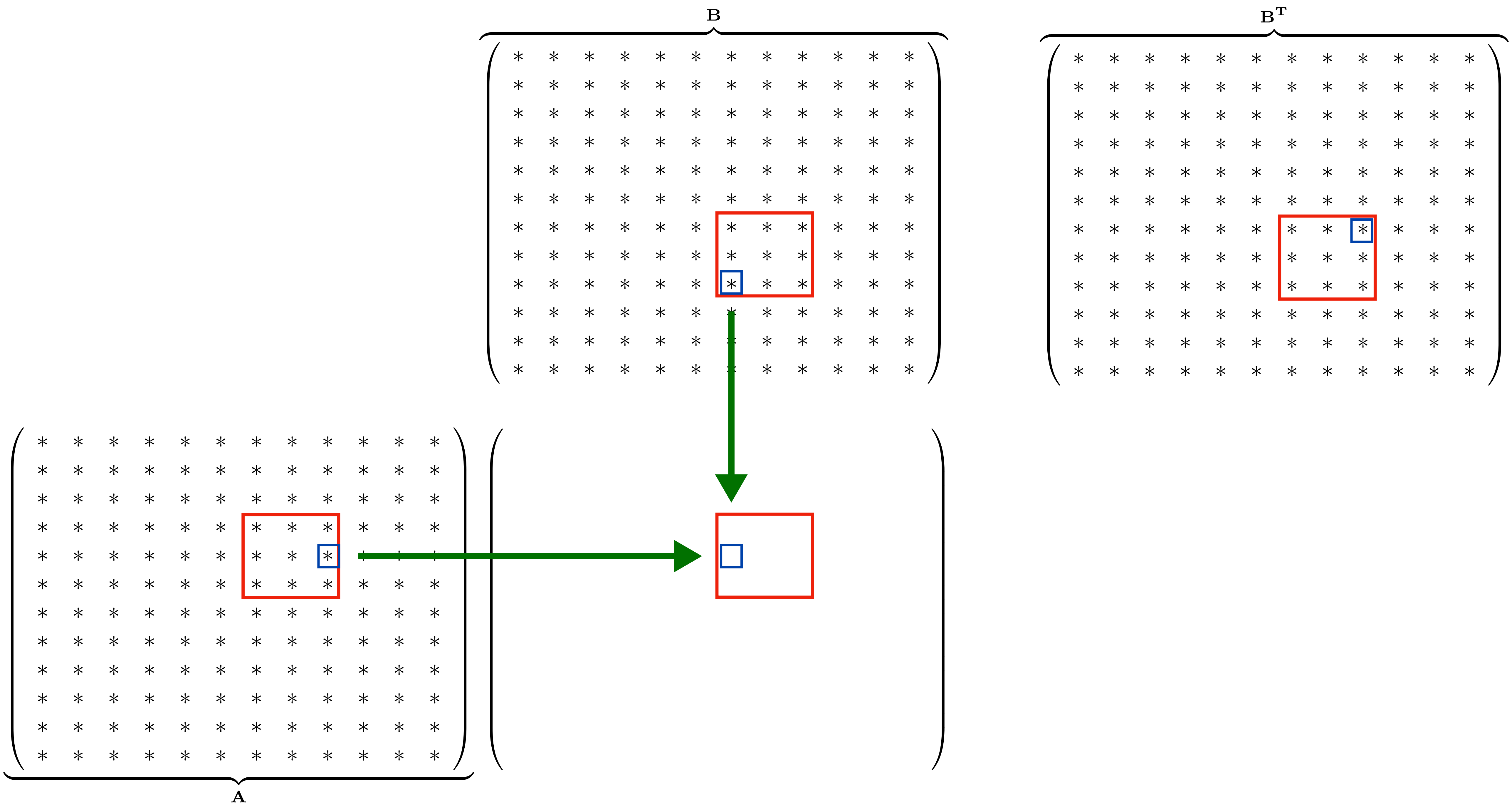
Combining blocking & pre-transposed B (or col-major B)



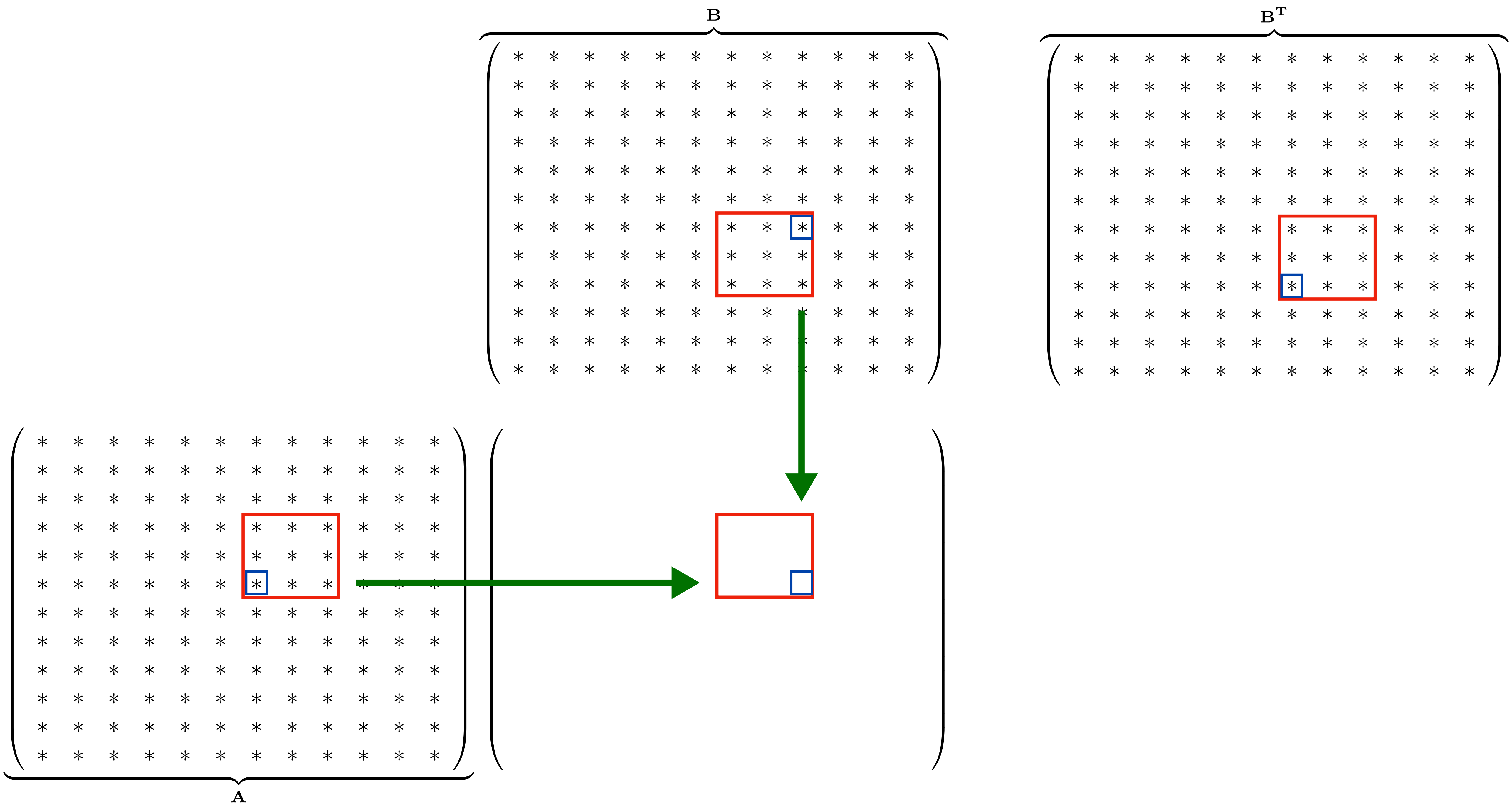
Combining blocking & pre-transposed B (or col-major B)



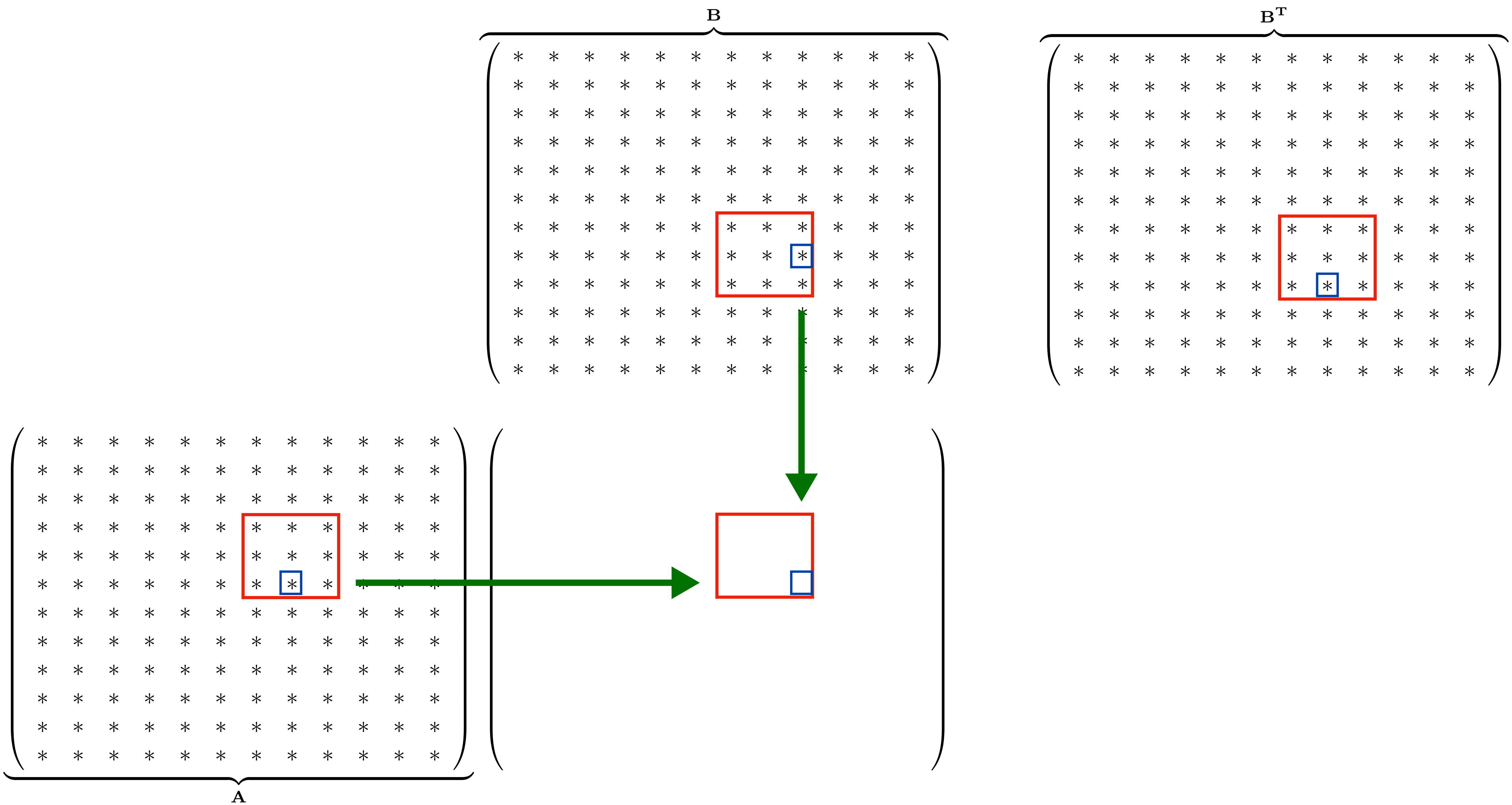
Combining blocking & pre-transposed B (or col-major B)



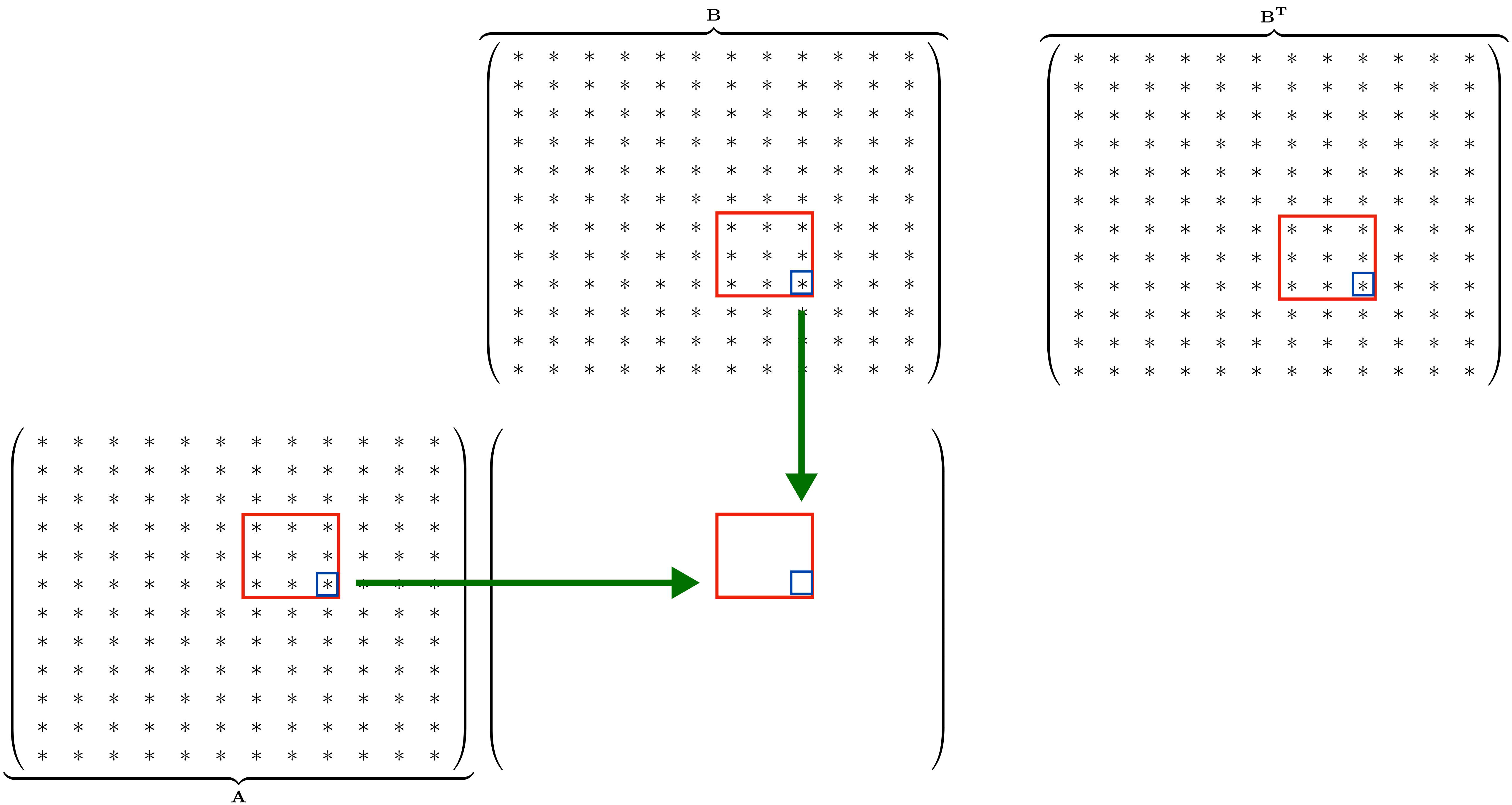
Combining blocking & pre-transposed B (or col-major B)



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

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


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

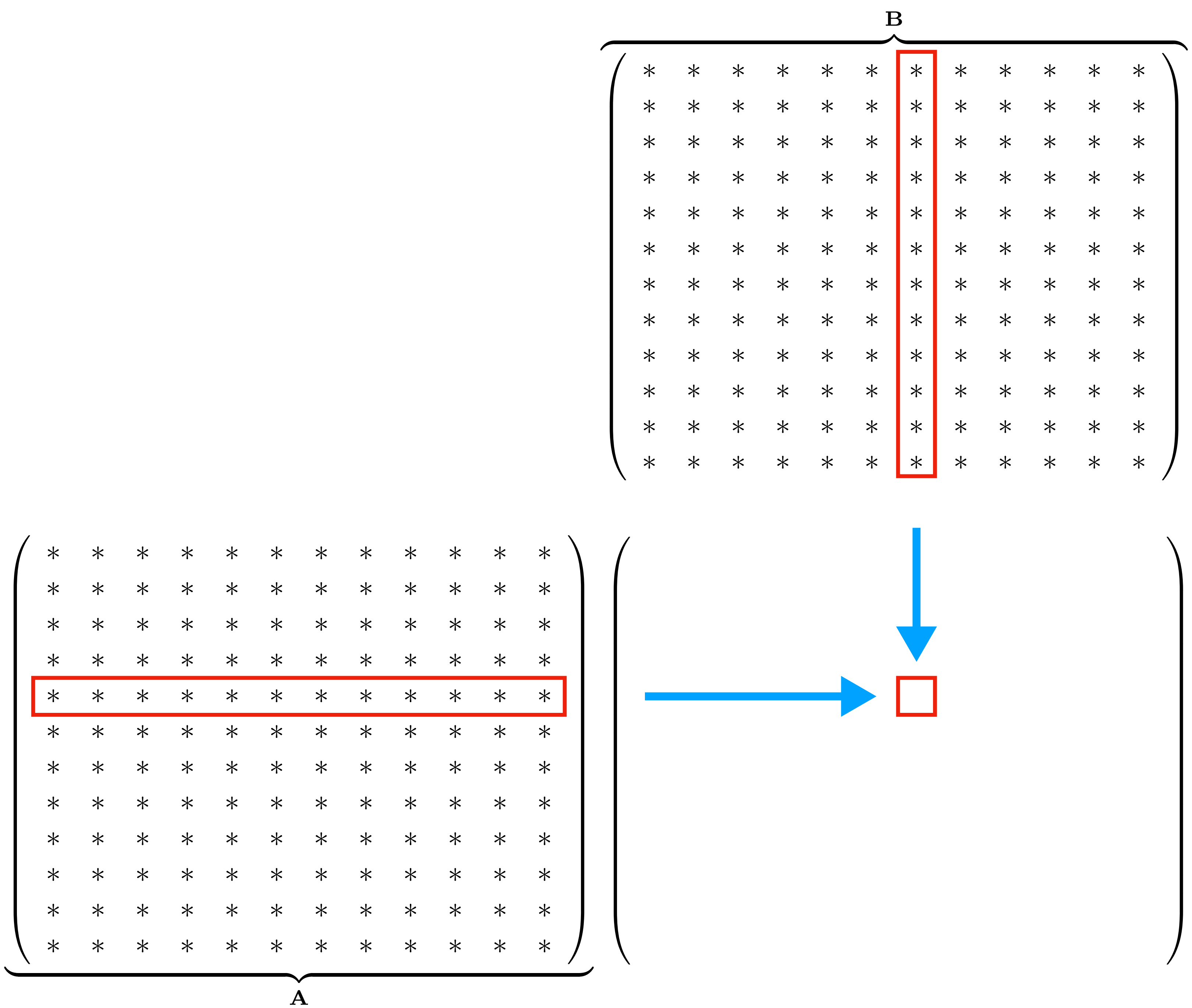
Matrix size 2048 x 2048

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]
[...]
```

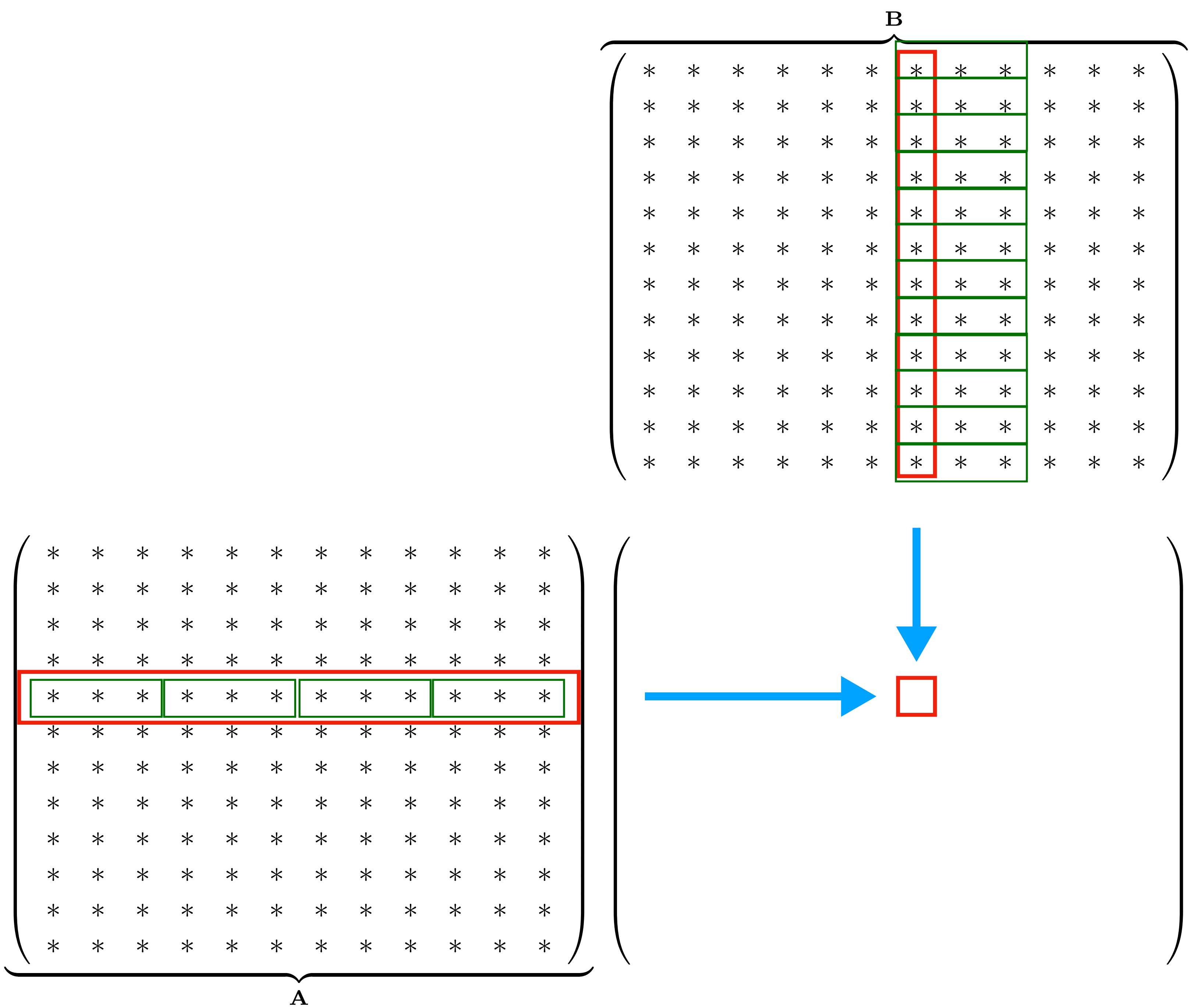
Causes of slowdown

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



Causes of slowdown

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

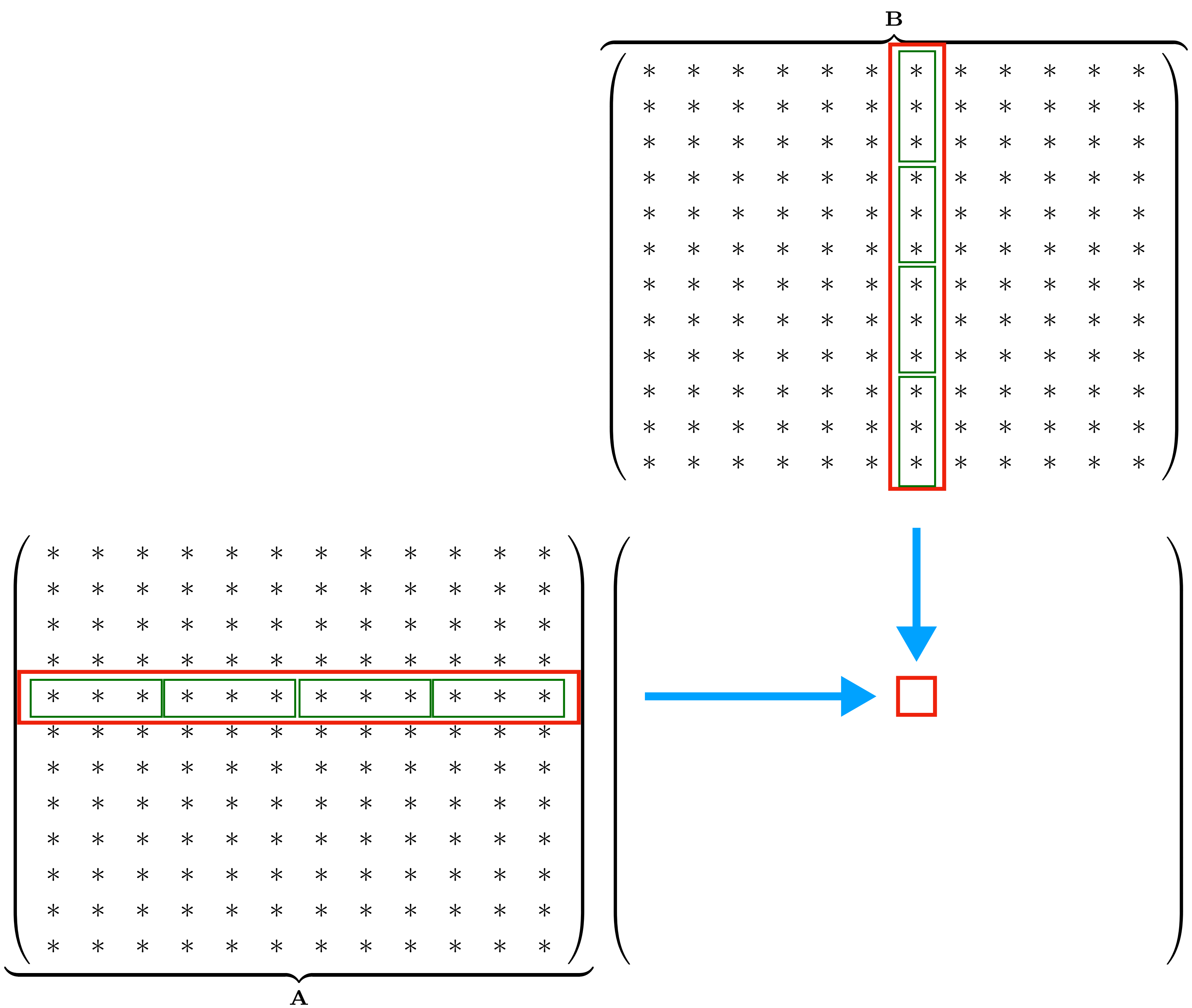


Shapes of cache lines (for row-major matrices)

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

Causes of slowdown

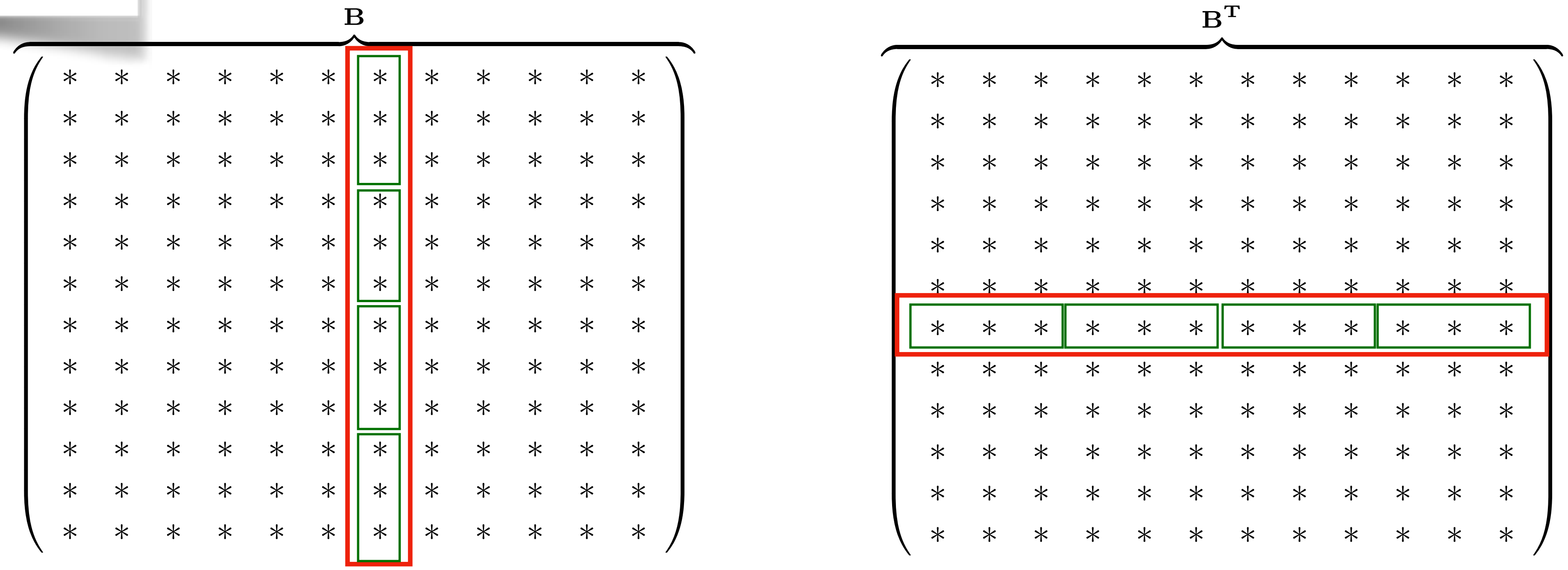
for $i = 1 \ldots N$
    for $j = 1 \ldots N$
        $C_{ij} \leftarrow 0$
        for $k = 1 \ldots N$
            $C_{ij} \leftarrow C_{ij} + A_{ik} B_{kj}$

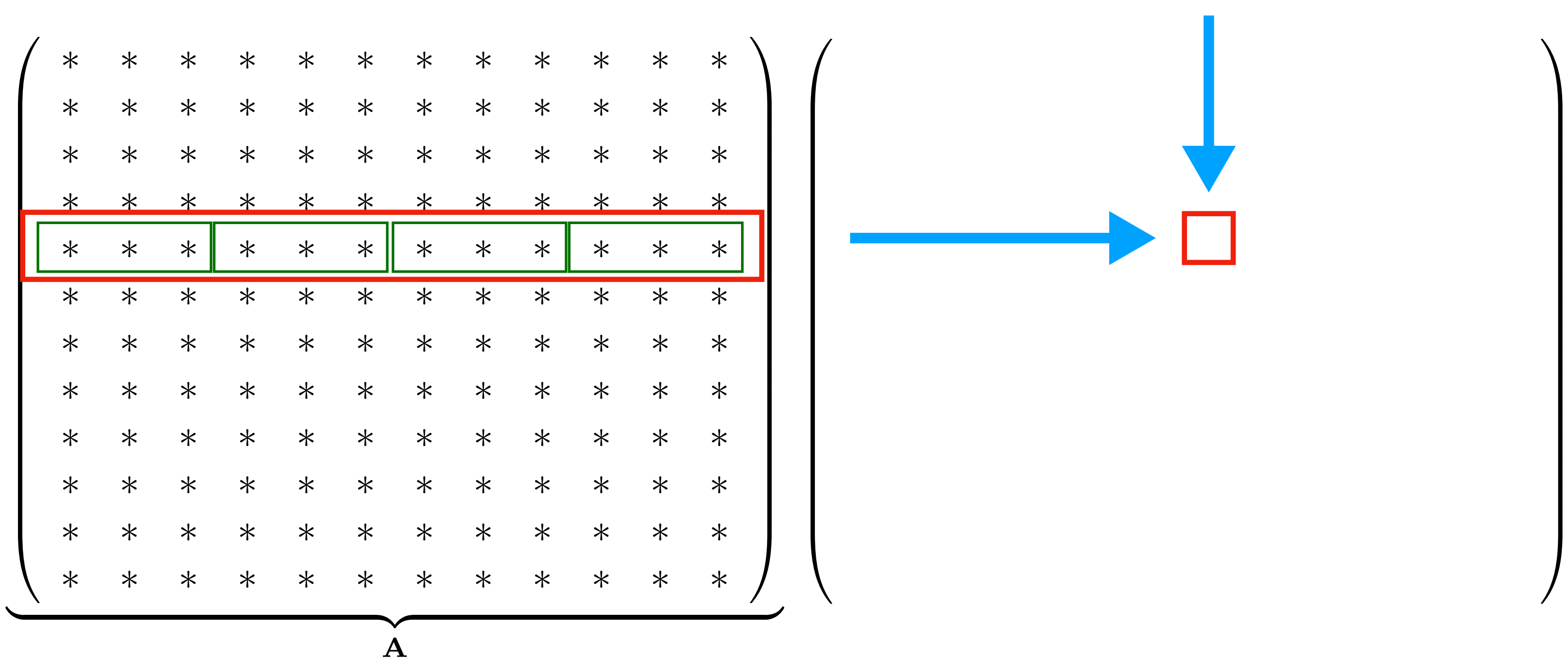


*If, instead, **B** was given as column-major …*

… cache lines are more effectively utilized

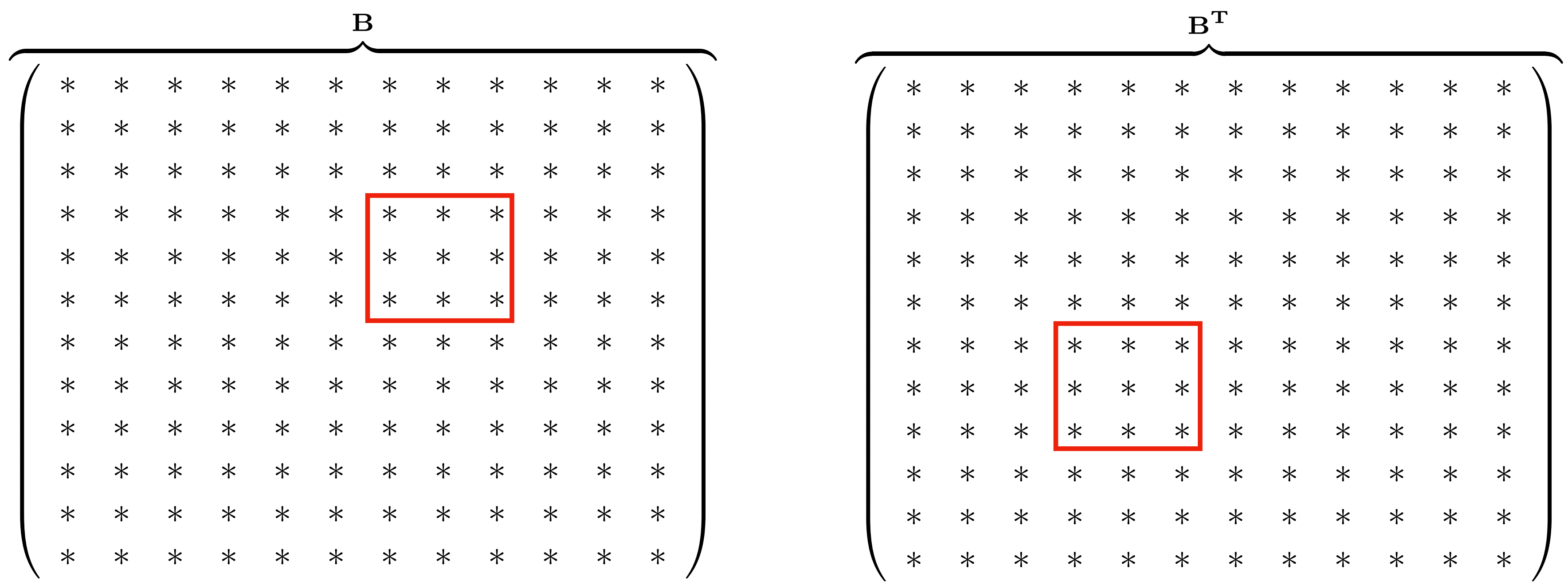
Using transpose ...

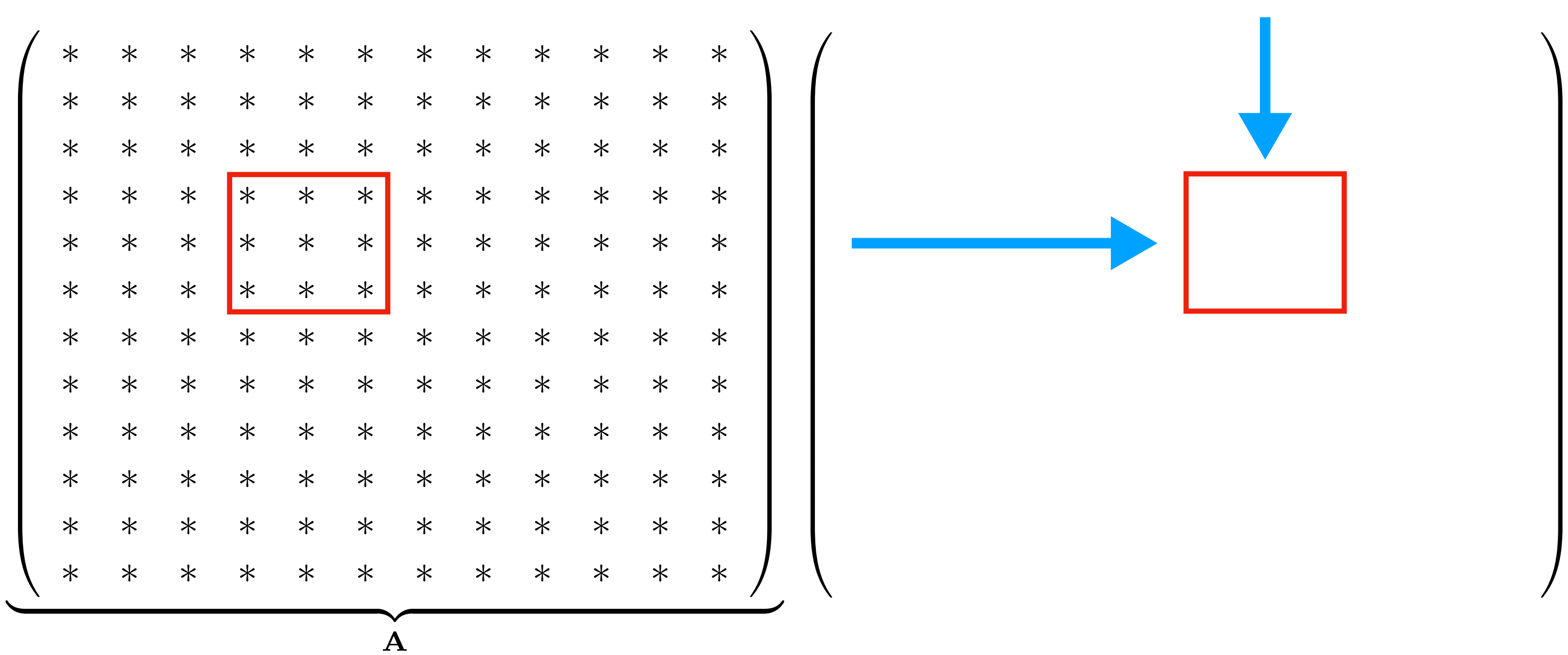




$$
\begin{array}{l}
\texttt{for}\ \ i = 1 \ldots N \\
\quad \texttt{for}\ \ j = 1 \ldots N \\
\quad\quad C_{ij} \leftarrow 0 \\
\quad\quad \texttt{for}\ \ k = 1 \ldots N \\
\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} [B^T]_{jk}
\end{array}
$$

Is that still a problem with blocking?





for $i = 1 \ldots N$

  for $j = 1 \ldots N$

    $C_{ij} \leftarrow 0$

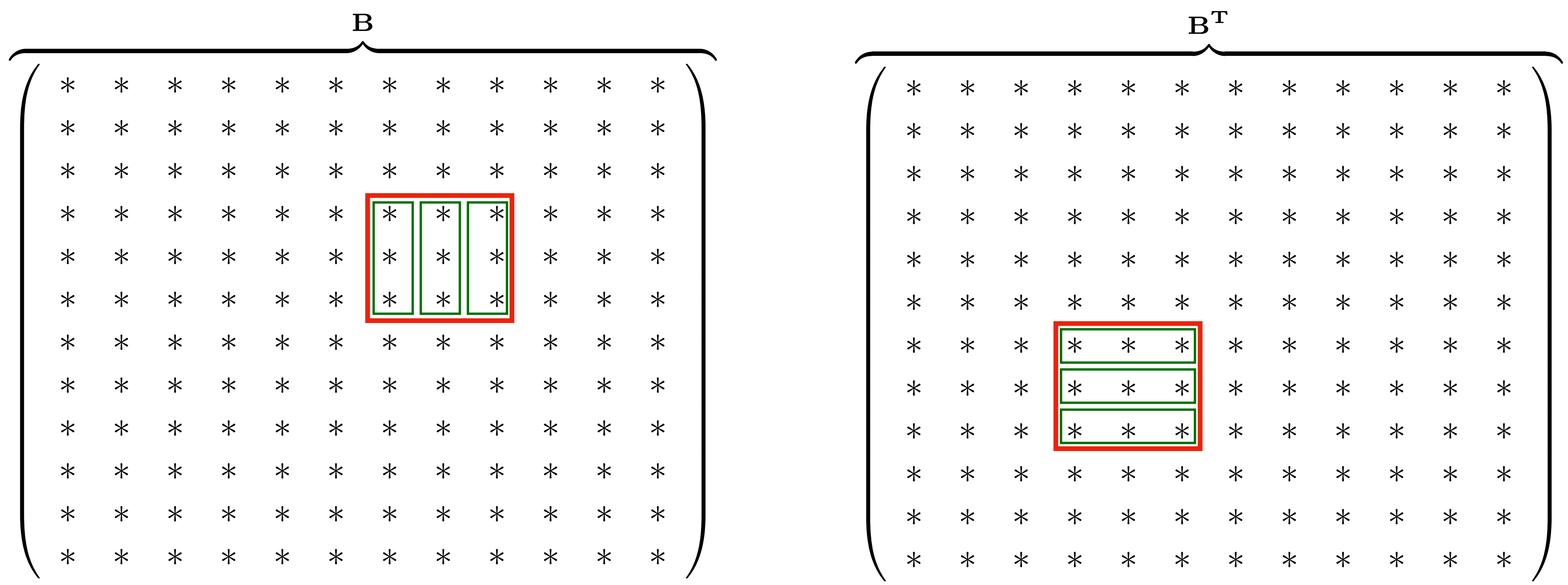
    for $k = 1 \ldots N$

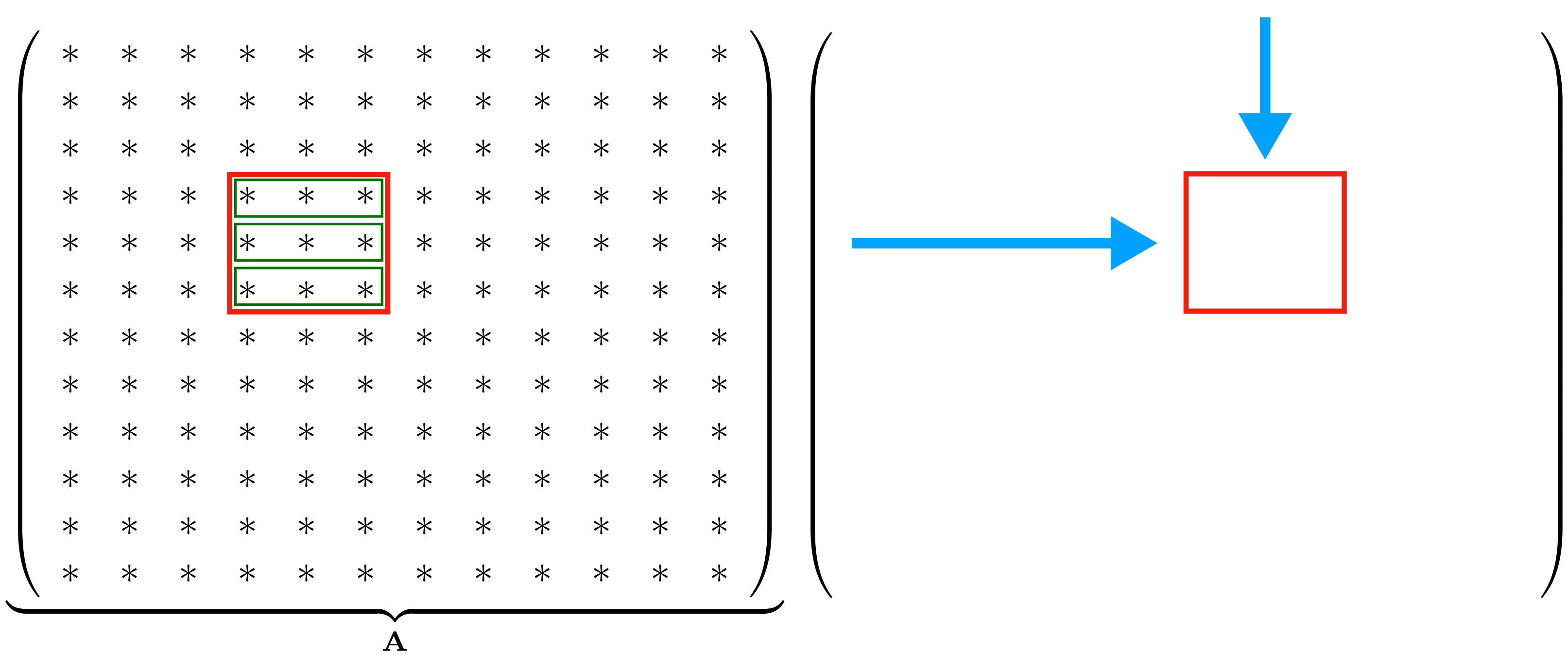
      $C_{ij} \leftarrow C_{ij} + A_{ik}[B^T]_{jk}$

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

Is that still a problem with blocking?

The block fits entire
cache lines for optimum
performance

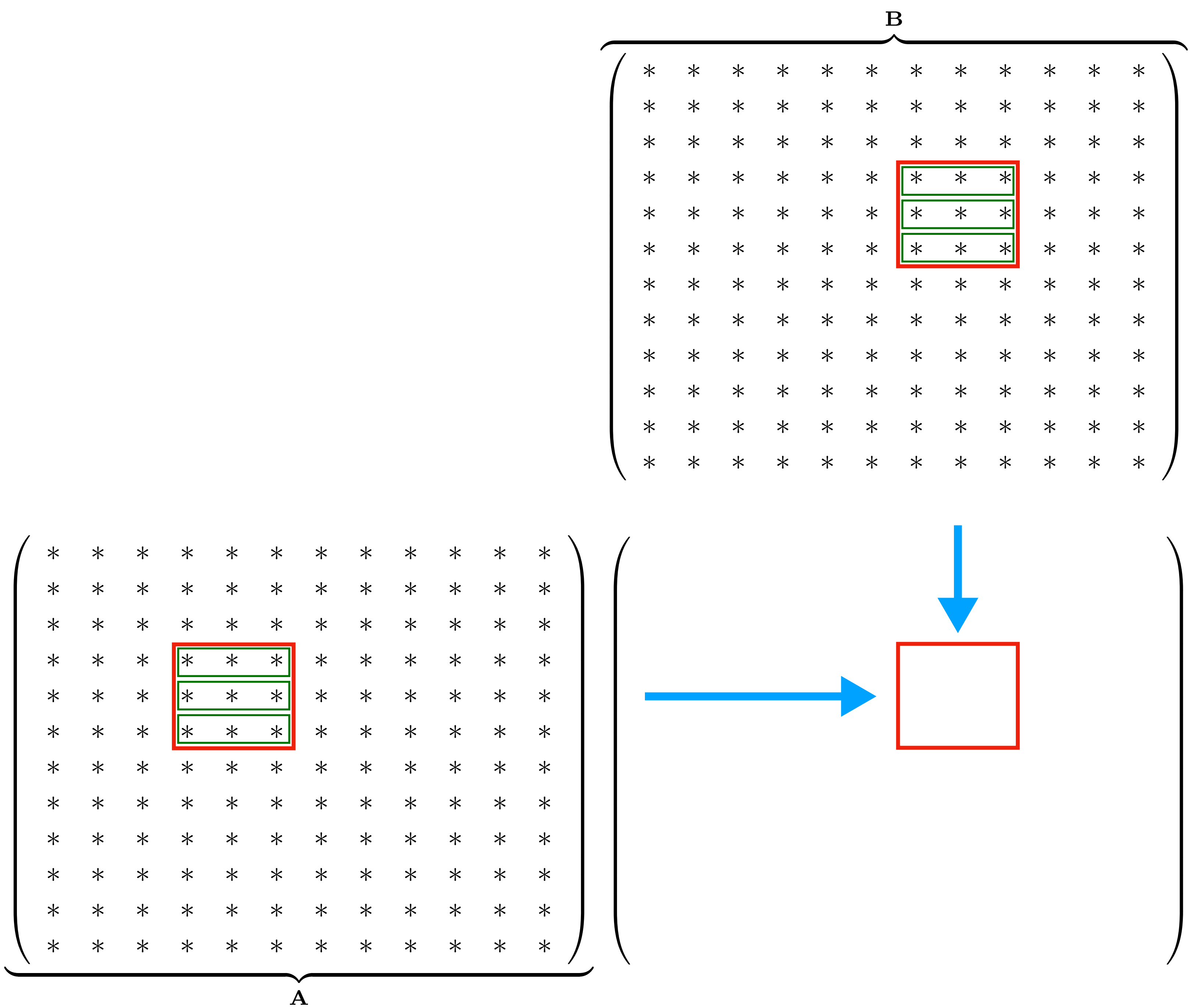




$$
\begin{aligned}
&\texttt{for}\ \ i = 1 \ldots N \\
&\quad \texttt{for}\ \ j = 1 \ldots N \\
&\quad\quad C_{ij} \leftarrow 0 \\
&\quad\quad \texttt{for}\ \ k = 1 \ldots N \\
&\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} [B^T]_{jk}
\end{aligned}
$$

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

Blocks larger than cache lines: no need to transpose



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

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 $\boldsymbol{B^T}$ *in advance …*

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_9

MatMatMultiply without transpose

```
[...]

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 : ");

    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;
}
```

... and here ...

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])
{
[...]
}
```

DenseAlgebra/GEMM_Test_0_8

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 ...

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];
    }
}
```

DenseAlgebra/GEMM_Test_0_9

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

DenseAlgebra/GEMM_Test_0_9

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 (i
            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]
[...]
```

DenseAlgebra/GEMM_Test_0_9

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?)

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_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];
    }
}
```

Accumulate the partial products into the localC variable instead of the blockC

*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)

DenseAlgebra/GEMM_Test_0_9

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];
    }
}
```

With very large BLOCK_SIZES, the parallelism is limited because of being compute bound given the large number of rwos. However, oversubscription when it comes to parallelim is generally preferred. Smaller than cache line sizes are considered bad for performance due to lack of reuse

Restored regularity of computation
Code is more SIMD-friendly

2 is better than 1

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]
[...]
```

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[
    }
}
```

Constrained by L1 cache size (BLOCK_SIZE * BLOCK_SIZE)

*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]
[...]
```

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_1

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

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

At matrix size = 2048

Execution:

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

Progress so far

Starting from a prototype implementation that was ~150x slower than the MKL version, we increased the performance to about ~2.5x of the MKL library

Important considerations that helped achieve this:
- Being conscious of cache line utilization
- Trying to make repeated memory accesses on data that is well cached
- Generating good opportunities for SIMD processing

Types of program transformations we used:
- Recasting/Reshaping data into blocked form
(note that we didn't end up having to "copy" data, in the end, just using casts)
- Reorganizing loops, often braking them into smaller, nested loops
- Using alignment when possible
- Giving a few SIMD hints when appropriate

`called zero copy using recast of existing data`

Most of the practices seen so far would be typical of good implementation design (none are "too extreme" to use in workloads like GEMM)

The next (and last) step

Next, we will look at some "extreme" optimizations
(mostly to get an idea of what extra tricks the MKL library might have used)

There is a price to be paid for going this far on optimizations:
- Some of the code will not be easily portable or compile without changes on all compilers (especially when using assembly-language tricks)
- Performance gains can be volatile; faster on some CPUs, not so fast on others
- Code that is so invasively optimized is more difficult to reuse

You will <u>not</u> be asked to reproduce these kinds of optimizations in homework
(the goal is to just know/appreciate the types of tricks that are possible)

The demonstrations (including code) will presume compilation using the Intel C++ Compiler (not an endorsement; just a point of reference) since syntax may vary across compilers
(but other compilers typically allow this to be done, too, using different syntax)

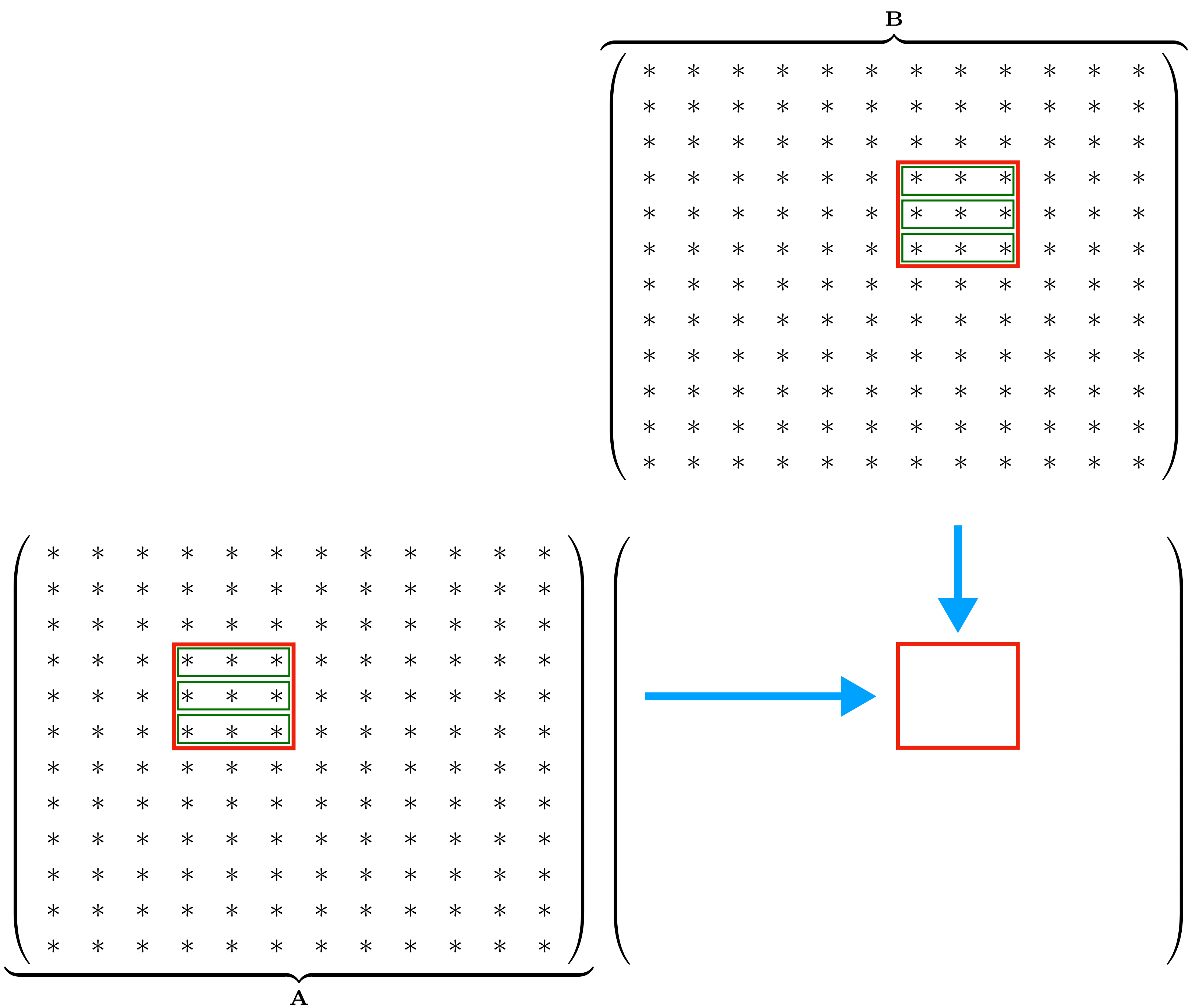
Guidelines & Best Practices

Recommendation:
- Try to <u>isolate</u> the code "hotspot" that has the most dramatic impact on runtime performance (the more you can localize it, the better).
- Not a simple recipe for doing that (although profiling tools help …) but if you have a hunch, you can try to experimentally validate it

Matrix Multiplication (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];
    }
}
```

Blocked multiplication



$\texttt{for } i = 1 \ldots N$

$\quad \texttt{for } j = 1 \ldots N$

$\quad\quad C_{ij} \leftarrow 0$

$\quad\quad \texttt{for } k = 1 \ldots N$

$\quad\quad\quad C_{ij} \leftarrow C_{ij} + A_{ik} B_{kj}$

C_{ij}*, *A_{ik}* and *B_{kj} *represent block 3x3 sub-matrices*

DenseAlgebra/GEMM_Test_0_10

Matrix Multiplication (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];
    }
}
```

We would reasonably suspect this is the code "hotspot" …

DenseAlgebra/GEMM_Test_0_10a

Matrix Multiplication (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;
                localC[ii][jj] = 0.;

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

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

What if we replace the matrix multiplication with an (incorrect, but cheaper) element-by-element multiply?

(Note: this yields incorrect result!)

DenseAlgebra/GEMM_Test_0_10a

Matrix Multiplication (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; 
                localC[ii][jj] = 0.;

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

            for (int ii = 0; ii < BLOCK_SIZE; i
            for (in
                loc
                loc

            for (in
                for

        }

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

What if we replace the matrix multiplication with an (incorrect, but cheaper) element-by-element multiply?
(Note: this yields incorrect result!)

Execution:

```
Running candidate kernel for correctness test ... [Elapsed time : 25.8265ms]
Running reference kernel for correctness test ... [Elapsed time : 40.7448ms]
Discrepancy between two methods : 81.673
Running kernel for performance run # 1 ... [Elapsed time : 9.24573ms]
Running kernel for performance run # 2 ... [Elapsed time : 7.97733ms]
Running kernel for performance run # 3 ... [Elapsed time : 7.85168ms]
Running kernel for performance run # 4 ... [Elapsed time : 7.79981ms]
Running kernel for performance run # 5 ... [Elapsed time : 7.80448ms]
Running kernel for performance run # 6 ... [Elapsed time : 7.80988ms]
Running kernel for performance run # 7 ... [Elapsed time : 7.81487ms]
Running kernel for performance run # 8 ... [Elapsed time : 7.80276ms]
[...]
```

Matrix Multiplication (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];
    }
}
```

Focus our attention on this very code segment (thankfully, it's "small" enough)

Matrix Multiplication (MatMatMultiply.cpp)

```
#include "MatMatMultiply.h"
#include "MatMatMultiplyBlockHelper.h"
[...]
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]; }

            MatMatMultiplyBlockHelper(localA, localB, localC);
        }

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

Factor out the “local” multiplication of the BxB blocks into its own function

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#pragma once

#include "Parameters.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE]);
```

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

Costly inner-matrix multiply factored into separate .cpp file

```
#include "MatMatMultiplyBlockHelper.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    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++)
            bC[ii][jj] += bA[ii][kk] * bB[kk][jj];
    }
}
```

DenseAlgebra/GEMM_Test_0_11

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#include "MatMatMultiplyBlockHelper.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    for (int ii = 0; ii < BLOCK_SIZE; ii++)
        for (int kk = 0; kk < BLOCK_SIZE; kk++) {
#pragma omp simd
        for (int jj
            bC[ii][
    }
}
```

What happened ...?

Execution:

```
Running candidate kernel for correctness test ... [Elapsed time : 72.8214ms]
Running reference kernel for correctness test ... [Elapsed time : 37.2703ms]
Discrepancy between two methods : 0.000164032
Running kernel for performance run # 1 ... [Elapsed time : 49.1049ms]
Running kernel for performance run # 2 ... [Elapsed time : 48.6051ms]
Running kernel for performance run # 3 ... [Elapsed time : 48.8517ms]
Running kernel for performance run # 4 ... [Elapsed time : 48.9314ms]
Running kernel for performance run # 5 ... [Elapsed time : 48.7969ms]
Running kernel for performance run # 6 ... [Elapsed time : 48.7675ms]
Running kernel for performance run # 7 ... [Elapsed time : 48.2165ms]
Running kernel for performance run # 8 ... [Elapsed time : 48.762ms]
[...]
```

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#include "MatMatMultiplyBlockHelper.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    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++)
            bC[ii][jj] += bA[ii][kk] * bB[kk][jj];
    }
}
```

Compler is unaware that there are no aliases for the pointers passed to this function (ie. same pointer passed to multiple args)

To decipher what happened … look into assembly language generated

```
icc -S MatMatMultiplyBlockHelper.cpp -qopenmp -xCOMMON-AVX512
    -o MatMatMultiplyBlockHelper.AVX512.AliasesAllowed.s
```

DenseAlgebra/GEMM_Test_0_11

Assembly code

MatMatMultiplyBlockHelper.AVX512.AliasesAllowed.s

```
# MatMatMultiplyBlockHelper(const float (&)[64][64], const float (&)[64][64], float (&)[64][64])
_Z25MatMatMultiplyBlockHelperRA64_A64_KfS2_RA64_A64_f:
# parameter 1: %rdi
# parameter 2: %rsi
# parameter 3: %rdx
[...]
sdf
..B1.3:                         # Preds ..B1.3 ..B1.2
                                # Execution count [4.10e+03]
        vbroadcastss (%r9,%r10,4), %zmm4                            #10.27
        vfmadd231ps (%r8,%rsi), %zmm4, %zmm0                        #10.13
        vmovups   %zmm0, (%rax,%rdx)                                #10.13
        vbroadcastss (%r9,%r10,4), %zmm5                            #10.27
        vfmadd231ps 64(%r8,%rsi), %zmm5, %zmm1                      #10.13
        vmovups   %zmm1, 64(%rax,%rdx)                              #10.13
        vbroadcastss (%r9,%r10,4), %zmm6                            #10.27
        vfmadd231ps 128(%r8,%rsi), %zmm6, %zmm2                     #10.13
        vmovups   %zmm2, 128(%rax,%rdx)                             #10.13
        vbroadcastss (%r9,%r10,4), %zmm7                            #10.27
        incq      %r10                                              #7.9
        vfmadd231ps 192(%r8,%rsi), %zmm7, %zmm3                     #10.13
        addq      $256, %r8                                         #7.9
        vmovups   %zmm3, 192(%rax,%rdx)                             #10.13
        cmpq      $64, %r10                                         #7.9
        jb        ..B1.3        # Prob 98%                          #7.9
```

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#include "MatMatMultiplyBlockHelper.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    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++)
            bC[ii][jj] += bA[ii][kk] * bB[kk][jj];
    }
}
```

Assembly code

MatMatMultiplyBlockHelper.AVX512.AliasesAllowed.s

DenseAlgebra/GEMM_Test_0_11

```
# MatMatMultiplyBlockHelper(const float (&)[64][64], const float (&)[64][64], float (&)[64][64])
_Z25MatMatMultiplyBlockHelperRA64_A64_KfS2_RA64_A64_f:
# parameter 1: %rdi
# parameter 2: %rsi
# parameter 3: %rdx
[...]
sdf
..B1.3:                         # Preds ..B1.3 ..B1.2
                                # Execution count [4.10e+03]
        vbroadcastss (%r9,%r10,4), %zmm4                        #10.27
        vfmadd231ps (%r8,%rsi), %zmm4, %zmm0                    #10.13
        vmovups   %zmm0, (%rax,%rdx)                            #10.13
        vbroadcastss (%r9,%r10,4), %zmm5                        #10.27
        vfmadd231ps 64(%r8,%rsi), %zmm5, %zmm1                  #10.13
        vmovups   %zmm1, 64(%rax,%rdx)                          #10.13
        vbroadcastss (%r9,%r10,4), %zmm6                        #10.27
        vfmadd231ps 128(%r8,%rsi), %zmm6, %zmm2                 #10.13
        vmovups   %zmm2, 128(%rax,%rdx)                         #10.13
        vbroadcastss (%r9,%r10,4), %zmm7                        #10.27
        incq      %r10                                          #7.9
        vfmadd231ps 192(%r8,%rsi), %zmm7, %zmm3                 #10.13
        addq      $256, %r8                                     #7.9
        vmovups   %zmm3, 192(%rax,%rdx)                         #10.13
        cmpq      $64, %r10                                     #7.9
        jb        ..B1.3        # Prob 98%                      #7.9
```

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#include "MatMatMultiplyBlockHelper.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    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++)
            bC[ii][jj] += bA[ii][kk] * bB[kk][jj];
    }
}
```

Assembly code

MatMatMultiplyBlockHelper.AVX512.AliasesAllowed.s

```
# MatMatMultiplyBlockHelper(const float (&)[64][64], const float (&)[64][64], float (&)[64][64])
_Z25MatMatMultiplyBlockHelperRA64_A64_KfS2_RA64_A64_f:
# parameter 1: %rdi
# parameter 2: %rsi
# parameter 3: %rdx
[...]
sdf
..B1.3:                         # Preds ..B1.3 ..B1.2
                                # Execution count [4.10e+03]
        vbroadcastss (%r9,%r10,4), %zmm4                         #10.27
        vfmadd231ps (%r8,%rsi), %zmm4, %zmm0                     #10.13
        vmovups   %zmm0, (%rax,%rdx)                             #10.13
        vbroadcastss (%r9,%r10,4), %zmm5                         #10.27
        vfmadd231ps 64(%r8,%rsi), %zmm5, %zmm1                   #10.13
        vmovups   %zmm1, 64(%rax,%rdx)                           #10.13
        vbroadcastss (%r9,%r10,4), %zmm6                         #10.27
        vfmadd231ps 128(%r8,%rsi), %zmm6, %zmm2                  #10.13
        vmovups   %zmm2, 128(%rax,%rdx)                          #10.13
        vbroadcastss (%r9,%r10,4), %zmm7                         #10.27
        incq      %r10                                           #7.9
        vfmadd231ps 192(%r8,%rsi), %zmm7, %zmm3                  #10.13
        addq      $256, %r8                                      #7.9
        vmovups   %zmm3, 192(%rax,%rdx)                          #10.13
        cmpq      $64, %r10                                      #7.9
        jb        ..B1.3        # Prob 98%                       #7.9
```

DenseAlgebra/GEMM_Test_0_11

Assembly code

MatMatMultiplyBlockHelper.AVX512.AliasesAllowed.s

```
# MatMatMultiplyBlockHelper(const float (&)[64][64], const float (&)[64][64], float (&)[64][64])
_Z25MatMatMultiplyBlockHelperRA64_A64_KfS2_RA64_A64_f:
# parameter 1: %rdi
# parameter 2: %rsi
# parameter 3: %rdx
[...]
sdf
..B1.3:                         # Preds ..B1.3 ..B1.2
                                # Execution count [4.10e+03]
        vbroadcastss (%r9,%r10,4), %zmm4                        #10.27
        vfmadd231ps (%r8,%rsi), %zmm4, %zmm0                    #10.13
        vmovups   %zmm0, (%rax,%rdx)                            #10.13
        vbroadcastss (%r9,%r10,4), %zmm5                        #10.27
        vfmadd231ps 64(%r8,%rsi), %zmm5, %zmm1                  #10.13
        vmovups   %zmm1, 64(%rax,%rdx)                          #10.13
        vbroadcastss (%r9,%r10,4), %zmm6                        #10.27
        vfmadd231ps 128(%r8,%rsi), %zmm6, %zmm2                 #10.13
        vmovups   %zmm2, 128(%rax,%rdx)                         #10.13
```

For comprehensive documentation on assembly instructions:
https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-software-developers-manual-volume-2d-instruction-set-reference

```
        vmovups   %zmm3, 192(%rax,%rdx)                         #10.13
        cmpq      $64, %r10                                     #7.9
        jb        ..B1.3        # Prob 98%                      #7.9
```

DenseAlgebra/GEMM_Test_0_11

Assembly code

MatMatMultiplyBlockHelper.AVX512.AliasesAllowed.s

Transmit the value bA[ii][kk] into all 16-entries of the vector register %zmm4

```
# MatMatMultiplyBlockHelper(const float (&)[64][64], const float (&)[64][64], float (&)[64][64])
_Z25MatMatMultiplyBlockHelperRA64_A64_KfS2_RA64_A64_f:
# parameter 1: %rdi
# parameter 2: %rsi
# parameter 3
[...]
sdf
..B1.3:                         # Preds ..B1.3 ..B1.2
                                # Execution count [4.10e+03]
        vbroadcastss (%r9,%r10,4), %zmm4                        #10.27
        vfmadd231ps (%r8,%rsi), %zmm4, %zmm0                    #10.13
        vmovups   %zmm0, (%rax,%rdx)                            #10.13
        vbroadcastss (%r9,%r10,4), %zmm5                        #10.27
        vfmadd231ps 64(%r8,%rsi), %zmm5, %zmm1                  #10.13
        vmovups   %zmm1, 64(%rax,%rdx)                          #10.13
        vbroadcastss (%r9,%r10,4), %zmm6                        #10.27
        vfmadd231ps 128(%r8,%rsi), %zmm6, %zmm2                 #10.13
        vmovups   %zmm2, 128(%rax,%rdx)                         #10.13
        vbroadcastss (%r9,%r10,4), %zmm7                        #10.27
        incq      %r10                                          #7.9
        vfmadd231ps 192(%r8,%rsi), %zmm7, %zmm3                 #10.13
        addq      $256, %r8                                     #7.9
        vmovups   %zmm3, 192(%rax,%rdx)                         #10.13
        cmpq      $64, %r10                                     #7.9
        jb        ..B1.3        # Prob 98%                      #7.9
```

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#include "MatMatMultiplyBlockHelper.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    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++)
            bC[ii][jj] += bA[ii][kk] * bB[kk][jj];
    }
}
```

Assembly code

MatMatMultiplyBlockHelper.AVX512.AliasesAllowed.s

Read bB[kk][jj] from memory, multiply element-by element with %zmm4 and add the results to an accumulation register %zmm0 corresponding to bC[ii][jj]

```
# MatMatMultiplyBlockHelper(const float (&)[64][64], const float (&)[64][64], float (&)[64][64])
_Z25MatMatMultiplyBlockHelperRA64_A64_KfS2_RA64_A64_f:
# parameter 1: %rdi
# parameter 2: %rsi
# paramete
[...]
sdf
..B1.3:                         # Preds ..B1.3 ..B1.2
                                # Execution count [4.10e+03]
        vbroadcastss (%r9,%r10,4), %zmm4                    #10.27
        vfmadd231ps (%r8,%rsi), %zmm4, %zmm0                #10.13
        vmovups   %zmm0, (%rax,%rdx)                        #10.13
        vbroadcastss (%r9,%r10,4), %zmm5                    #10.27
        vfmadd231ps 64(%r8,%rsi), %zmm5, %zmm1              #10.13
        vmovups   %zmm1, 64(%rax,%rdx)                      #10.13
        vbroadcastss (%r9,%r10,4), %zmm6                    #10.27
        vfmadd231ps 128(%r8,%rsi), %zmm6, %zmm2             #10.13
        vmovups   %zmm2, 128(%rax,%rdx)                     #10.13
        vbroadcastss (%r9,%r10,4), %zmm7                    #10.27
        incq      %r10                                      #7.9
        vfmadd231ps 192(%r8,%rsi), %zmm7, %zmm3             #10.13
        addq      $256, %r8                                 #7.9
        vmovups   %zmm3, 192(%rax,%rdx)                     #10.13
        cmpq      $64, %r10                                 #7.9
        jb        ..B1.3        # Prob 98%                  #7.9
```

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#include "MatMatMultiplyBlockHelper.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    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++)
            bC[ii][jj] += bA[ii][kk] * bB[kk][jj];
    }
}
```

DenseAlgebra/GEMM_Test_0_11

Assembly code

MatMatMultiplyBlockHelper.AVX512.AliasesAllowed.s

```
# MatMatMultiplyBlockHelper(const float (&)[64][64], const float (&)[64][64], float (&)[64][64])
_Z25MatMatMultiplyBlockHelperRA64_A64_KfS2_RA64_A64_f:
# parameter 1: %rdi
# parameter 2: %rsi
# p
[.
sd
```

Then, re-transmit the value bA[ii][kk] into all 16-entries of the vector register %zmm5

(WHY DO THIS AGAIN??)

```
..B1.3:                         # Preds ..B1.3 ..B1.2
                                # Execution count [4.10e+03]
        vbroadcastss (%r9,%r10,4), %zmm4                        #10.27
        vfmadd231ps (%r8,%rsi), %zmm4, %zmm0                    #10.13
        vmovups   %zmm0, (%rax,%rdx)                            #10.13
        vbroadcastss (%r9,%r10,4), %zmm5                        #10.27
        vfmadd231ps 64(%r8,%rsi), %zmm5, %zmm1                  #10.13
        vmovups   %zmm1, 64(%rax,%rdx)                          #10.13
        vbroadcastss (%r9,%r10,4), %zmm6                        #10.27
        vfmadd231ps 128(%r8,%rsi), %zmm6, %zmm2                 #10.13
        vmovups   %zmm2, 128(%rax,%rdx)                         #10.13
        vbroadcastss (%r9,%r10,4), %zmm7                        #10.27
        incq      %r10                                          #7.9
        vfmadd231ps 192(%r8,%rsi), %zmm7, %zmm3                 #10.13
        addq      $256, %r8                                     #7.9
        vmovups   %zmm3, 192(%rax,%rdx)                         #10.13
        cmpq      $64, %r10                                     #7.9
        jb        ..B1.3        # Prob 98%                      #7.9
```

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#include "MatMatMultiplyBlockHelper.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    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++)
            bC[ii][jj] += bA[ii][kk] * bB[kk][jj];
    }
}
```

"Argument/Variable aliasing"
The compiler has no way of knowing that bA, bB & bC are non-overlapping arrays (hence, it needs to account for the possibility that writing into bC might have changed the contents of bA!!)

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#include "MatMatMultiplyBlockHelper.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    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++)
            bC[ii][jj] += bA[ii][kk] * bB[kk][jj];
    }
}
```

Instruct the compiler that no aliases are present

```
icc -S MatMatMultiplyBlockHelper.cpp -qopenmp -xCOMMON-AVX512
 -fno-alias -o MatMatMultiplyBlockHelper.AVX512.NoAliases.s
```

Assembly code

MatMatMultiplyBlockHelper.AVX512.NoAliases.s

```
# MatMatMultiplyBlockHelper(const float (&)[64][64], const float (&)[64][64], float (&)[64][64])
_Z25MatMatMultiplyBlockHelperRA64_A64_KfS2_RA64_A64_f:
# parameter 1: %rdi
# parameter 2: %rsi
# parameter 3: %rdx
[...]
..B1.3:                         # Preds ..B1.3 ..B1.2
                                # Execution count [4.10e+03]
        vbroadcastss (%r9,%r10,4), %zmm4                          #10.27
        incq      %r10                                            #7.9
        vfmadd231ps 64(%r8,%rsi), %zmm4, %zmm0                    #10.13
        vfmadd231ps 128(%r8,%rsi), %zmm4, %zmm1                   #10.13
        vfmadd231ps 192(%r8,%rsi), %zmm4, %zmm3                   #10.13
        vfmadd231ps (%r8,%rsi), %zmm4, %zmm2                      #10.13
        vmovups   %zmm0, 64(%rax,%rdx)                            #10.13
        vmovups   %zmm1, 128(%rax,%rdx)                           #10.13
        addq      $256, %r8                                       #7.9
        cmpq      $64, %r10                                       #7.9
        jb        ..B1.3        # Prob 98%                        #7.9
..B1.4:                         # Preds ..B1.3
                                # Execution count [6.40e+01]
        incb      %cl                                             #6.5
        vmovups   %zmm3, 192(%rax,%rdx)                           #10.13
        vmovups   %zmm2, (%rax,%rdx)                              #10.13
        addq      $256, %rax                                      #6.5
        cmpb      $64, %cl                                        #6.5
        jb        ..B1.2        # Prob 98%                        #6.5
```

DenseAlgebra/GEMM_Test_0_11

Assembly code

MatMatMultiplyBlockHelper.AVX512.NoAliases.s

```
# MatMatMultiplyBlockHelper(const float (&)[64][64], const float (&)[64][64], float (&)[64][64])
_Z25MatMatMultiplyBlockHelperRA64_A64_KfS2_RA64_A64_f:
# parameter 1: %rdi
# parameter 2: %rsi
# parameter 3: %rdx
[...]
..B1.3:                         # Preds ..B1.3 ..B1.2
                                # Execution count [4.10e+03]
        vbroadcastss (%r9,%r10,4), %zmm4                        #10.27
        incq      %r10                                          #7.
        vfmadd231ps 64(%r8,%rsi), %zmm4, %zmm0                  #1
        vfmadd231ps 128(%r8,%rsi), %zmm4, %zmm1                 #10.13
        vfmadd231ps 192(%r8,%rsi), %zmm4, %zmm3                 #10.13
        vfmadd231ps (%r8,%rsi), %zmm4, %zmm2                    #10.13
        vmovups   %zmm0, 64(%rax,%rdx)                          #10.13
        vmovups   %zmm1, 128(%rax,%rdx)                         #10.13
        addq      $256, %r8                                     #7.9
        cmpq      $64, %r10                                     #7.9
        jb        ..B1.3        # Prob 98%                      #7.9
..B1.4:                         # Preds ..B1.3
                                # Execution count [6.40e+01]
        incb      %cl                                           #6.5
        vmovups   %zmm3, 192(%rax,%rdx)                         #10.13
        vmovups   %zmm2, (%rax,%rdx)                            #10.13
        addq      $256, %rax                                    #6.5
        cmpb      $64, %cl                                      #6.5
        jb        ..B1.2        # Prob 98%                      #6.5
```

Read bB[kk][jj] just once!

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#include "MatMatMultiplyBlockHelper.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    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++)
            bC[ii][jj] += bA[ii][kk] * bB[kk][jj];
    }
}
```

Assembly code

MatMatMultiplyBlockHelper.AVX512.NoAliases.s

DenseAlgebra/GEMM_Test_0_11

```
# MatMatMultiplyBlockHelper(const float (&)[64][64], const float (&)[64][64], float (&)[64][64])
_Z25MatMatMultiplyBlockHelperRA64_A64_KfS2_RA64_A64_f:
# parameter 1: %rdi
# parameter 2: %rsi
# parameter 3: %rdx
[...]
..B1.3:                         # Preds ..B1.3 ..B1.2
                                # Execution count [4.10e+03]
        vbroadcastss (%r9,%r10,4), %zmm4
        incq      %r10
        vfmadd231ps 64(%r8,%rsi), %zmm4, %zmm0
        vfmadd231ps 128(%r8,%rsi), %zmm4, %zmm1
        vfmadd231ps 192(%r8,%rsi), %zmm4, %zmm3
        vfmadd231ps (%r8,%rsi), %zmm4, %zmm2
        vmovups   %zmm0, 64(%rax,%rdx)
        vmovups   %zmm1, 128(%rax,%rdx)                          #10.13
        addq      $256, %r8                                      #7.9
        cmpq      $64, %r10                                      #7.9
        jb        ..B1.3        # Prob 98%                       #7.9
..B1.4:                         # Preds ..B1.3
                                # Execution count [6.40e+01]
        incb      %cl                                            #6.5
        vmovups   %zmm3, 192(%rax,%rdx)                          #10.13
        vmovups   %zmm2, (%rax,%rdx)                             #10.13
        addq      $256, %rax                                     #6.5
        cmpb      $64, %cl                                       #6.5
        jb        ..B1.2        # Prob 98%                       #6.5
```

Do the multiplication & addition for the 64 entries of the row in 4 tightly-packed fused-multiply-add instructions (16 entries at a time)

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#include "MatMatMultiplyBlockHelper.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    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++)
            bC[ii][jj] += bA[ii][kk] * bB[kk][jj];
    }
}
```

Assembly code

MatMatMultiplyBlockHelper.AVX512.NoAliases.s

```
# MatMatMultiplyBlockHelper(const float (&)[64][64], const float (&)[64][64], float (&)[64][64])
_Z25MatMatMultiplyBlockHelperRA64_A64_KfS2_RA64_A64_f:
# parameter 1: %rdi
# parameter 2: %rsi
# parameter 3: %rdx
[...]
..B1.3:                         # Preds ..B1.3 ..B1.2
                                # Execution count [4.10e+03]
        vbroadcastss (%r9,%r10,4), %zmm4
        incq      %r10
        vfmadd231ps 64(%r8,%rsi), %zmm4, %zmm
        vfmadd231ps 128(%r8,%rsi), %zmm4, %zm
        vfmadd231ps 192(%r8,%rsi), %zmm4, %zm
        vfmadd231ps (%r8,%rsi), %zmm4, %zmm2                        #10.13
        vmovups   %zmm0, 64(%rax,%rdx)                              #10.13
        vmovups   %zmm1, 128(%rax,%rdx)                             #10.13
        addq      $256, %r8                                         #7.9
        cmpq      $64, %r10                                         #7.9
        jb        ..B1.3        # Prob 98%                          #7.9
..B1.4:                         # Preds ..B1.3
                                # Execution count [6.40e+01]
        incb      %cl                                               #6.5
        vmovups   %zmm3, 192(%rax,%rdx)                             #10.13
        vmovups   %zmm2, (%rax,%rdx)                                #10.13
        addq      $256, %rax                                        #6.5
        cmpb      $64, %cl                                          #6.5
        jb        ..B1.2        # Prob 98%                          #6.5
```

Write the result back into bC[ii][jj] using 4x 16-wide store ("move") instructions (the compiler does a bit extra loop reordering, hence the split of the "move" instructions)

DenseAlgebra/GEMM_Test_0_11

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#include "MatMatMultiplyBlockHelper.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    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++)
            bC[ii][jj] += bA[ii][kk] * bB[kk][jj];
    }
}
```

Building the entire executable
(it's important to only use the "no aliases" option on the isolated Block-matrix-multiply code file, as we do employ aliases widely elsewhere in the code, i.e. all the matrix casts …)

```
icc -c MatMatMultiplyBlockHelper.cpp -qopenmp -xCOMMON-AVX512 -fno-alias

icc main.cpp MatMatMultiply.cpp Utilities.cpp MatMatMultiplyBlockHelper.o
    -xCOMMON-AVX512 -mkl -xHost
```

DenseAlgebra/GEMM_Test_0_11

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#include "MatMatMultiplyBlockHelper.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    for (int ii = 0; ii < BLOCK_SIZE; ii++)
        for (int kk = 0; kk < BLOCK_SIZE; kk++) {
#pragma omp simd
        for (int jj
            bC[ii][
    }
}
```

1.8x the runtime of MKL code!

Execution:

```
Running candidate kernel for correctness test ... [Elapsed time : 41.0696ms]
Running reference kernel for correctness test ... [Elapsed time : 39.3287ms]
Discrepancy between two methods : 0.000160217
Running kernel for performance run # 1 ... [Elapsed time : 23.0806ms]
Running kernel for performance run # 2 ... [Elapsed time : 22.582ms]
Running kernel for performance run # 3 ... [Elapsed time : 22.1493ms]
Running kernel for performance run # 4 ... [Elapsed time : 21.97ms]
Running kernel for performance run # 5 ... [Elapsed time : 22.8829ms]
Running kernel for performance run # 6 ... [Elapsed time : 22.8694ms]
Running kernel for performance run # 7 ... [Elapsed time : 21.5426ms]
Running kernel for performance run # 8 ... [Elapsed time : 22.7655ms]
[...]
```