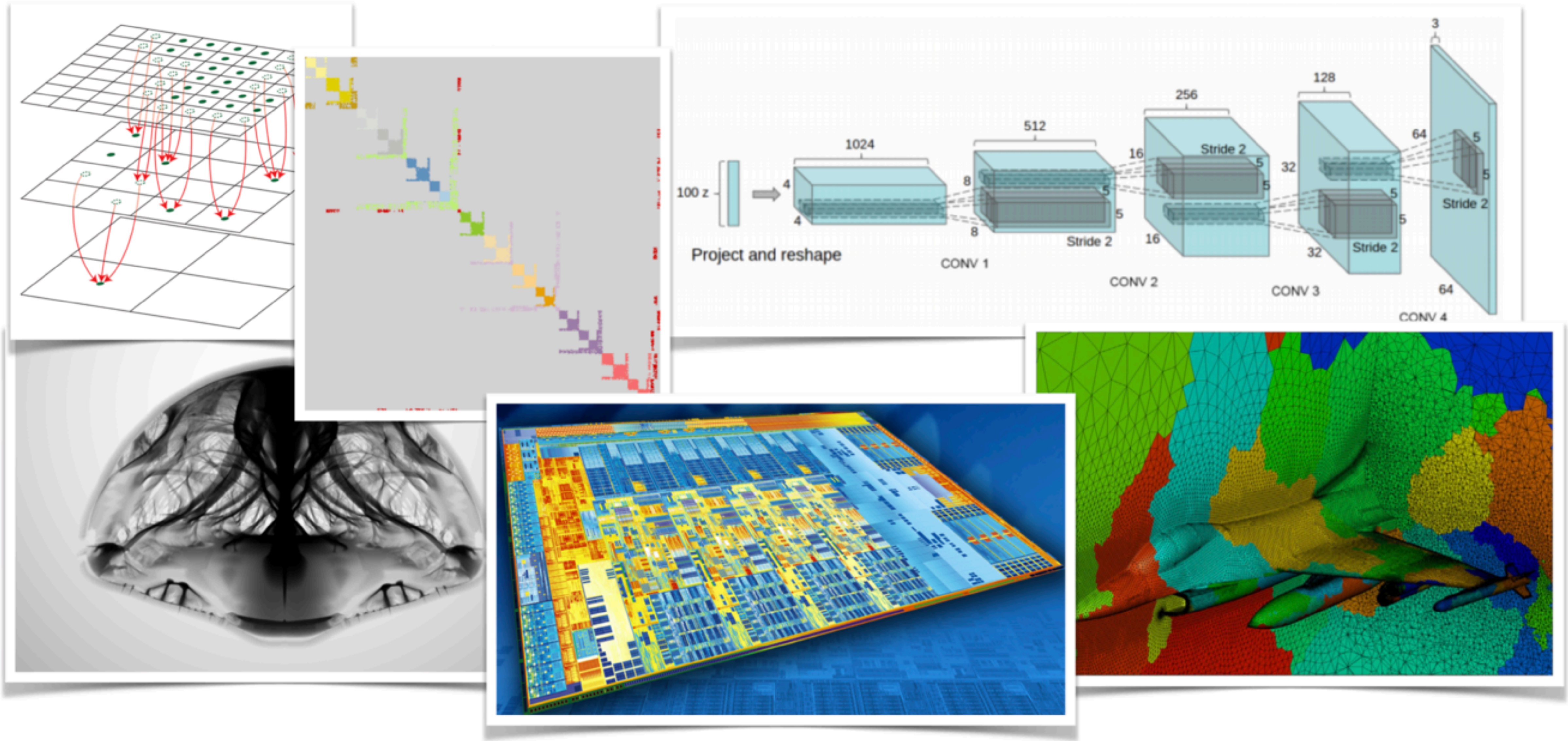


Lecture 15: Continued optimizations on GEMM kernels - Assembly-language level optimizations (via intrinsics)

Tuesday March 21st 2023

Logistics

- Homework #3 to be released this week
  - As usual you'll have at least 1 week to turn it in.
  - Please check that you can use MKL (if you haven't already!). See the discussion in the March 2nd lecture, too.
  - Hope to have midterm grades by end of week
    (and working on HW#1 grading)

Today's lecture

- Continued focus on GEMM operations

- We will look into the most aggressive (and ultimately, last-mile) optimizations, that will involve explicitly looking into, and maybe writing assembly-level code.

- We will use intrinsics to access assembly-level functionality within the framework of the C compiler.

GEMM optimization (getting towards the end)

Today, we 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

RECAP

Recommendation:
- Try to isolate 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

DenseAlgebra/GEMM_Test_0_10

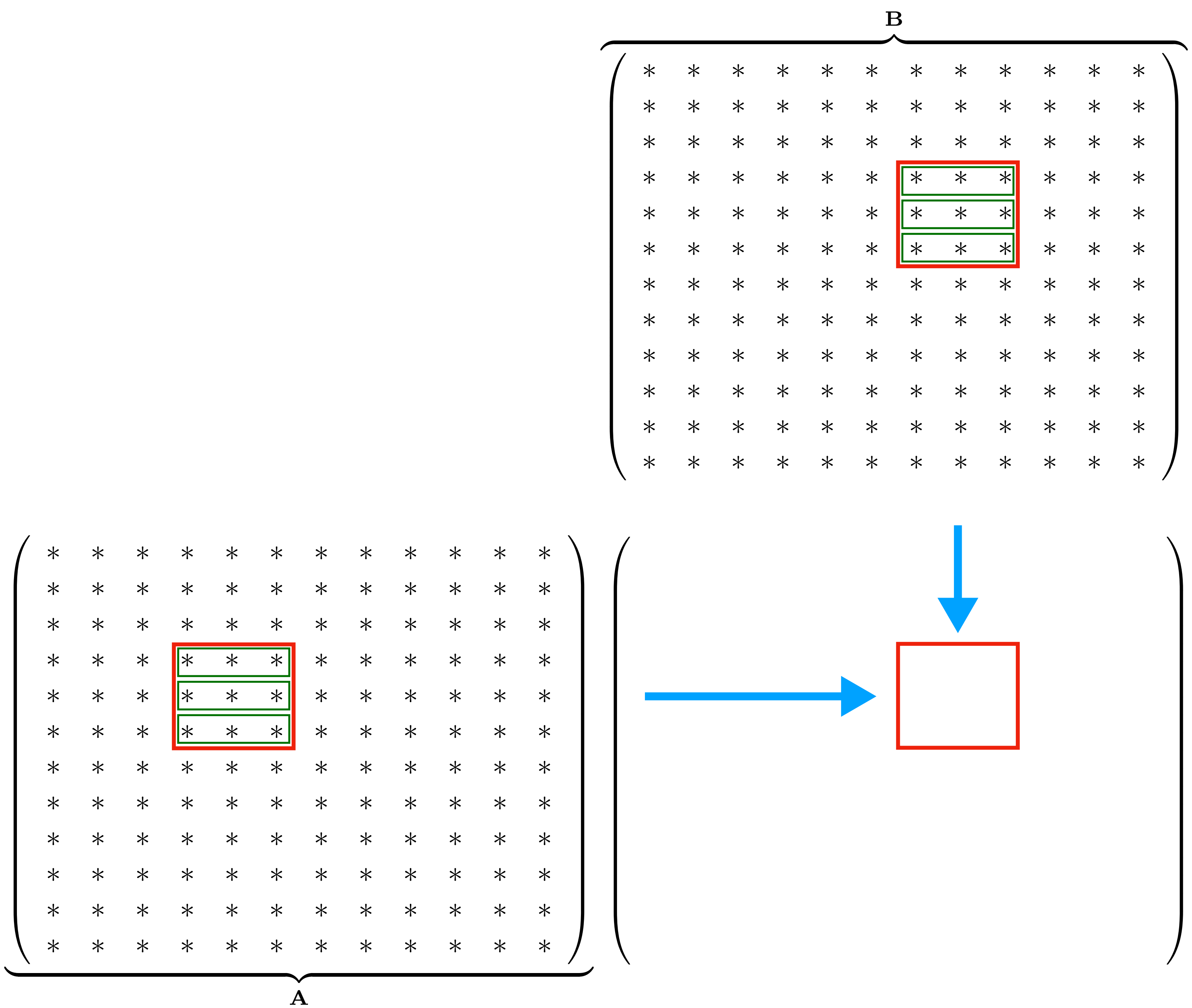
Matrix Multiplication (MatMatMultiply.cpp)

RECAP

```
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

RECAP



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

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

DenseAlgebra/GEMM_Test_0_10

Matrix Multiplication (MatMatMultiply.cpp)

RECAP

```
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)

RECAP

```
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)

RECAP

```
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; j
                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]
[...]
```

DenseAlgebra/GEMM_Test_0_10

Matrix Multiplication (MatMatMultiply.cpp)

RECAP

```
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)

DenseAlgebra/GEMM_Test_0_11

Matrix Multiplication (MatMatMultiply.cpp)

RECAP

```
#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)

RECAP

```
#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]);
```

DenseAlgebra/GEMM_Test_0_11

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

RECAP

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];
    }
}
```

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

RECAP

```
#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];
    }
}
```

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

```
# 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
```

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
```

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:                          #
                                 #
        vbroadcastss (%r9,%r10,4),
        vfmadd231ps (%r8,%rsi), %z
        vmovups   %zmm0, (%rax,%rd
        vbroadcastss (%r9,%r10,4),
        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
```

The first part of an "assembly source" line of code is the "mnemonic" of the operation itself (a somewhat human-readable version of the instruction as opposed to a binary value, or opcode, used to store this in memory)

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,
        vmovups   %zmm1, 64(%rax,%rdx)
        vbroadcastss (%r9,%r10,4), %zmm6
        vfmadd231ps 128(%r8,%rsi), %zmm6
        vmovups   %zmm2, 128(%rax,%rdx)
        vbroadcastss (%r9,%r10,4), %zmm7
        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
```

*These symbols represent **vector** registers.*
AVX512 registers are %zmm0 through %zmm31
AVX registers are %ymm0, …
SSE registers are %xmm0, …

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), %zmm
        incq      %r10
        vfmadd231ps 192(%r8,%rsi), %zmm
        addq      $256, %r8
        vmovups   %zmm3, 192(%rax,%rdx)
        cmpq      $64, %r10                                     #7.9
        jb        ..B1.3        # Prob 98%                      #7.9
```

We also have integer registers (typically 64-bit)
%r0, %r1, … are general purpose registers
%rdi, %rsi, %rax, %rdx are "special purpose" ones
(some with historically established roles)

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)
        vbroadcastss (%r9,%r10,4), %zmm
        incq      %r10
        vfmadd231ps 192(%r8,%rsi), %zmm
        addq      $256, %r8
        vmovups   %zmm3, 192(%rax,%rdx)
        cmpq      $64, %r10                                     #7.9
        jb        ..B1.3        # Prob 98%                      #7.9
```

Expressions in parentheses are essentially memory addresses; think them as "dereferencing" a pointer. Here (%rax, %rdx) refers to the memory location at position %rax + %rdx

[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)
        vbroadcastss (%r9,%r10,4), %zm
        incq      %r10
        vfmadd231ps 192(%r8,%rsi), %zm
        addq      $256, %r8
        vmovups   %zmm3, 192(%rax,%rdx
        cmpq      $64, %r10
        jb        ..B1.3        # Prob 98%                      #7.9
```

We can also include a constant "offset" in such memory-address expressions.
*Here **64(%rax, %rdx)** refers to the memory location at position **%rax + %rdx + 64***

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,%r
        vmovups   %zmm1, 64(%r
        vbroadcastss (%r9,%r1
        vfmadd231ps 128(%r8,%r
        vmovups   %zmm2, 128(%
        vbroadcastss (%r9,%r1
        incq      %r10
        vfmadd231ps 192(%r8,%r
        addq      $256, %r8
        vmovups   %zmm3, 192(%
        cmpq      $64, %r10
        jb        ..B1.3
```

Or, we can provide a constant scale (1x, 4x, or 8x typically) for the second operand of such expression. Here ***(%r9, %r10, 4)*** *refers to the memory location at position* ***%r9 + %r10 * 4***

For example, if we had an array ***float myArr[...]*** *and %r9 points to the start of the array, while %r10 contains an integer index "i", then the above expression essentially accesses the element* ***myArr[i]***

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
..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
```

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

(WHY DO THIS AGAIN??)

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!!)

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];
    }
}
```

When we had this operation in-line, the compiler could know for sure that localA, localB, and localC are completely distinct arrays …

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];
        }
}
```

… because we had declared them ourselves, in the same file in fact.

DenseAlgebra/GEMM_Test_0_11

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; i
            for (int jj = 0; jj < BLOCK_SIZE; j
                localA[ii][jj] = blockA[bi][ii]
                localB[ii][jj] = blockB[bk][ii]

            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];
    }
}
```

But in our last code transformation, function MatMatMultiplyBlockHelper() was in its own .cpp file; no way to know that its arguments are non-overlapping/distinct.

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

```
# 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)

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];
    }
}
```

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
        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
```

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]
[...]
```

Last-mile optimizations …

So far, we looked at instances where performance of transformed (or refactored) code would be best understood by looking at assembly code.

However, we did not write such assembly code directly … today we will.

These optimizations can be CPU-specific and compiler-specific (we will focus on machines that support AVX2 and/or AVX512)

You can look up what your processor supports at ark.intel.com (if Intel CPU)

Goal : Understand the nature, style and motivation of these optimizations
You will not be required to reproduce such optimizations in homework or exams

Note on examples

- Makefiles for Linux (should be ok in OS X too) are included
- Typing "make assembly" should produce the assembly code for MatMatMultiplyBlockHelper.cpp (with an .s extension)
- Many directories will include the assembly file into the repository, for reference
*- All examples in **GEMM_Test_1_XXX***

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];
    }
}
```

Build with:

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

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

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]
[...]
```

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];
    }
}
```

We still had to rely on OpenMP to (hopefully) generate SIMD code

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_0_avx512

```
#include "MatMatMultiplyBlockHelper.h"
#include "immintrin.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of SIMD vectors

    for (int ii = 0; ii < BLOCK_SIZE; ii++)
    for (int kk = 0; kk < BLOCK_SIZE; kk++)
        for (int jj = 0; jj < BLOCK_SIZE; jj += nW) {
            __m512 vB = _mm512_load_ps(&bB[kk][jj]);
            __m512 vA = _mm512_set1_ps(bA[ii][kk]);
            __m512 vC = _mm512_load_ps(&bC[ii][jj]);
            vC = _mm512_fmadd_ps(vA, vB, vC);
            _mm512_store_ps(&bC[ii][jj], vC);
        }
}
```

This code explicitly intended for an AVX512-compatible CPU …

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_0_avx2

```
#include "MatMatMultiplyBlockHelper.h"
#include "immintrin.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 8; // Width of SIMD vectors

    for (int ii = 0; ii < BLOCK_SIZE; ii++)
    for (int kk = 0; kk < BLOCK_SIZE; kk++)
        for (int jj = 0; jj < BLOCK_SIZE; jj += nW) {
            __m256 vB = _mm256_load_ps(&bB[kk][jj]);
            __m256 vA = _mm256_set1_ps(bA[ii][kk]);
            __m256 vC = _mm256_load_ps(&bC[ii][jj]);
            vC = _mm256_fmadd_ps(vA, vB, vC);
            _mm256_store_ps(&bC[ii][jj], vC);
        }
}
```

… but a version is provided for AVX2-compatible CPUs

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_0_avx512

We will use <u>assembly intrinsics</u> …

```
#include "MatMatMultiplyBlockHelper.h"
#include "immintrin.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of SIMD vectors

    for (int ii = 0; ii < BLOCK_SIZE; ii++)
    for (int kk = 0; kk < BLOCK_SIZE; kk++)
        for (int jj = 0; jj < BLOCK_SIZE; jj += nW) {
            __m512 vB = _mm512_load_ps(&bB[kk][jj]);
            __m512 vA = _mm512_set1_ps(bA[ii][kk]);
            __m512 vC = _mm512_load_ps(&bC[ii][jj]);
            vC = _mm512_fmadd_ps(vA, vB, vC);
            _mm512_store_ps(&bC[ii][jj], vC);
        }
}
```

What are (assembly) intrinsics?

An <u>intrinsic function</u> is a subroutine built-in to the compiler, that has a special meaning (beyond what C++ would suggest)

Assembly intrinsics in C, in particular, are subroutines that (in principle) encapsulate the operation of one (or a few) CPU assembly instruction

The compiler typically also provides special data types to encapsulate special types of registers (what's relevant to us: SIMD registers)

Major benefit of using intrinsics
(as opposed to in-line assembly, or editing the assembly code file directly) :
- No need for allocating registers (compiler does it)
- Even ok to use more "vector" variables than available on processor
(the compiler will take care of stashing "spilling" them to temporary memory)
- Significantly easier syntax
- Intrinsics are available that map to several assembly instructions
(or can be collapsed into even fewer ones)

What are (assembly) intrinsics?

We will offer a brief walkthrough-by-example

Reference materials (if you want to dive deeper; not essential for this class)

Intel 64 & IA-32 Architectures Optimization Reference Manual
https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

Intel 64 and IA-32 Architectures Software Developer's Manual
https://software.intel.com/en-us/articles/intel-sdm

Intel Intrinsics Guide
https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_0_avx512

```
#include "MatMatMultiplyBlockHelper.h"
#include "immintrin.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of SIMD vectors

    for (int ii = 0; ii < BLOCK_SIZE; ii++)
    for (int kk = 0; kk < BLOCK_SIZE; kk++)
        for (int jj = 0; jj < BLOCK_SIZE; jj += n
            __m512 vB = _mm512_load_ps(&bB[kk][jj
            __m512 vA = _mm512_set1_ps(bA[ii][kk]
            __m512 vC = _mm512_load_ps(&bC[ii][jj
            vC = _mm512_fmadd_ps(vA, vB, vC);
            _mm512_store_ps(&bC[ii][jj], vC);
        }
}
```

Data types (encapsulating registers)

`__m512` *Register with 16 floats (512 bits)*

`__m256` *Register with 8 floats (256 bits)*

`__m512d` *Register with 8 doubles (512 bits)*

`__m256i` *Register with 8 32-bit ints (512 bits)*

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_0_avx512

```
#include "MatMatMultiplyBlockHelper.h"
#include "immintrin.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of SIMD vectors

    for (int ii = 0; ii < BLOCK_SIZE; ii++)
    for (int kk = 0; kk < BLOCK_SIZE; kk++)
        for (int jj = 0; jj < BLOCK_SIZE; jj += nW) {
            __m512 vB = _mm512_load_ps(&bB[kk][jj]);
            __m512 vA = _mm512_set1_ps(bA[ii][kk]);
            __m512 vC = _mm512_load_ps(&bC[ii][jj]);
            vC = _mm512_fmadd_ps(vA, vB, vC);
            _mm512_store_ps(&bC[ii][jj], vC);
        }
}
```

Load vector register

`_m512_load_ps` *Load register with 16 floats from a 64-byte aligned memory address (AVX512)*

`_m256_load_ps` *Load register with 8 floats from a 32-byte aligned memory address (AVX2)*

`vmovaps` *is the corresponding assembly instruction*

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_0_avx512

```
#include "MatMatMultiplyBlockHelper.h"
#include "immintrin.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of SIMD vectors

    for (int ii = 0; ii < BLOCK_SIZE; ii++)
    for (int kk = 0; kk < BLOCK_SIZE; kk++)
        for (int jj = 0; jj < BLOCK_SIZE; jj += nW) {
            __m512 vB = _mm512_load_ps(&bB[kk][jj]);
            __m512 vA = _mm512_set1_ps(bA[ii][kk]);
            __m512 vC = _mm512_load_ps(&bC[ii][jj]);
            vC = _mm512_fmadd_ps(vA, vB, vC);
            _mm512_store_ps(&bC[ii][jj], vC);
        }
}
```

Since we're working on 16-wide vectors, we are stepping by 16 each time

intel Intrinsics Guide

Technologies

- MMX
- SSE
- SSE2
- SSE3
- SSSE3
- SSE4.1
- SSE4.2
- AVX
- AVX2
- FMA
- AVX-512
- KNC
- SVML
- Other

Categories

- Application-Targeted
- Arithmetic
- Bit Manipulation
- Cast
- Compare
- Convert
- Cryptography
- Elementary Math Functions

The Intel Intrinsics Guide is an interactive reference tool for Intel intrinsic instructions, which are C style functions that provide access to many Intel instructions - including Intel® SSE, AVX, AVX-512, and more - without the need to write assembly code.

_mm512_load_ps

`__m512 _mm512_load_ps (void const* mem_addr)` vmovaps

Synopsis

```
__m512 _mm512_load_ps (void const* mem_addr)
#include <immintrin.h>
Instruction: vmovaps zmm, m512
CPUID Flags: AVX512F for AVX-512, KNCNI for KNC
```

Description

Load 512-bits (composed of 16 packed single-precision (32-bit) floating-point elements) from memory into `dst`. `mem_addr` must be aligned on a 64-byte boundary or a general-protection exception may be generated.

Operation

```
dst[511:0] := MEM[mem_addr+511:mem_addr]
dst[MAX:512] := 0
```

Performance

Architecture	Latency	Throughput (CPI)
Icelake	8	0.5
Skylake	1	0.5

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_0_avx512

```
#include "MatMatMultiplyBlockHelper.h"
#include "immintrin.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of SIMD vectors

    for (int ii = 0; ii < BLOCK_SIZE; ii++)
    for (int kk = 0; kk < BLOCK_SIZE; kk++)
        for (int jj = 0; jj < BLOCK_SIZE; jj += nW) {
            __m512 vB = _mm512_load_ps(&bB[kk][jj]);
            __m512 vA = _mm512_set1_ps(bA[ii][kk]);
            __m512 vC = _mm512_load_ps(&bC[ii][jj]);
            vC = _mm512_fmadd_ps(vA, vB, vC);
            _mm512_store_ps(&bC[ii][jj], vC);
        }
}
```

Store vector register

`_m512_store_ps` *Store register with 16 floats to a 64-byte aligned memory address (AVX512)*

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_0_avx512

```
#include "MatMatMultiplyBlockHelper.h"
#include "immintrin.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of SIMD vectors

    for (int ii = 0; ii < BLOCK_SIZE; ii++)
    for (int kk = 0; kk < BLOCK_SIZE; kk++)
        for (int jj = 0; jj < BLOCK_SIZE; jj += nW) {
            __m512 vB = _mm512_load_ps(&bB[kk][jj]);
            __m512 vA = _mm512_set1_ps(bA[ii][kk]);
            __m512 vC = _mm512_load_ps(&bC[ii][jj]);
            vC = _mm512_fmadd_ps(vA, vB, vC);
            _mm512_store_ps(&bC[ii][jj], vC);
        }
}
```

Fill vector register with copies of a single value

`_m512_set1_ps` *Fill all entries of a 16-float register with a single value (could be a constant, or a memory location)*

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_0_avx512

```
#include "MatMatMultiplyBlockHelper.h"
#include "immintrin.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of SIMD vectors

    for (int ii = 0; ii < BLOCK_SIZE; ii++)
    for (int kk = 0; kk < BLOCK_SIZE; kk++)
        for (int jj = 0; jj < BLOCK_SIZE; jj += nW) {
            __m512 vB = _mm512_load_ps(&bB[kk][jj]);
            __m512 vA = _mm512_set1_ps(bA[ii][kk]);
            __m512 vC = _mm512_load_ps(&bC[ii][jj]);
            vC = _mm512_fmadd_ps(vA, vB, vC);
            _mm512_store_ps(&bC[ii][jj], vC);
        }
}
```

`_m512_set1_ps` *is a example of an intrinsic without a "clear" 1-to-1 correspondence to a unique assembly instruction*

- If the argument is a constant, the compiler will take care of allocating/loading it

- If argument is a memory location, the `vbroadcastss` *instruction <u>may</u> be issued*

- In some cases the operation can be "embedded" in arithmetic assembly instructions

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_0_avx512

```
#include "MatMatMultiplyBlockHelper.h"
#include "immintrin.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of SIMD vectors

    for (int ii = 0; ii < BLOCK_SIZE; ii++)
    for (int kk = 0; kk < BLOCK_SIZE; kk++)
        for (int jj = 0; jj < BLOCK_SIZE; jj += nW) {
            __m512 vB = _mm512_load_ps(&bB[kk][jj]);
            __m512 vA = _mm512_set1_ps(bA[ii][kk]);
            __m512 vC = _mm512_load_ps(&bC[ii][jj]);
            vC = _mm512_fmadd_ps(vA, vB, vC);
            _mm512_store_ps(&bC[ii][jj], vC);
        }
}
```

Fused multiply-add

`_m512_fmadd_ps` *Multiplies first and second argument, add the third argument to the product, and returns the result to a vector register (kind of like the saxpy function we saw previously)*

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_0_avx512

```
#include "MatMatMultiplyBlockHelper.h"
#include "immintrin.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of SIMD vectors

    for (int ii = 0; ii < BLOCK_SIZE; ii++)
    for (int kk = 0; kk < BLOCK_SIZE; kk++)
        for (int jj = 0; jj < BLOCK_SIZE; jj += nW) {
            __m512 vB = _mm512_load_ps(&bB[kk][jj]);
            __m512 vA = _mm512_set1_ps(bA[ii][kk]);
            __m512 vC = _mm512_load_ps(&bC[ii][jj]);
            vC = _mm512_fmadd_ps(vA, vB, vC);
            _mm512_store_ps(&bC[ii][jj], vC);
        }
}
```

Fused multiply-add

`_m512_fmadd_ps` *may actually be translated to one of many assembly instructions such as* `vfmadd213ps`, `vfmadd132`, `vfmadd231` *depending on which of the 3 inputs we are writing the result to*

(or multiple assembly instructions if we are writing to a different register)

Assembly code

MatMatMultiplyBlockHelper.s

DenseAlgebra/GEMM_Test_1_0_avx512

```
[...]
        vmovups   192(%rax,%rdx), %zmm3                               #14.41
        xorl      %r8d, %r8d                                          #10.5
        vmovups   128(%rax,%rdx), %zmm2                               #14.41
        vmovups   64(%rax,%rdx), %zmm1                                #14.41
        vmovups   (%rax,%rdx), %zmm0                                  #14.41
                                # LOE rax rdx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 cl zmm0
zmm1 zmm2 zmm3
..B1.3:                         # Preds ..B1.3 ..B1.2
                                # Execution count [4.10e+03]
        vbroadcastss (%r9,%r10,4), %zmm4                              #13.40
        incq      %r10                                                #10.5
        vfmadd231ps (%r8,%rsi), %zmm4, %zmm0                          #15.18
        vfmadd231ps 64(%r8,%rsi), %zmm4, %zmm1                        #15.18
        vfmadd231ps 128(%r8,%rsi), %zmm4, %zmm2                       #15.18
        vfmadd231ps 192(%r8,%rsi), %zmm4, %zmm3                       #15.18
        addq      $256, %r8                                           #10.5
        cmpq      $64, %r10                                           #10.5
        jb        ..B1.3        # Prob 98%                            #10.5
                                # LOE rax rdx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 cl zmm0
zmm1 zmm2 zmm3
..B1.4:                         # Preds ..B1.3
                                # Execution count [6.40e+01]
        incb      %cl                                                 #9.5
        vmovups   %zmm3, 192(%rax,%rdx)                               #16.30
        vmovups   %zmm2, 128(%rax,%rdx)                               #16.30
        vmovups   %zmm1, 64(%rax,%rdx)                                #16.30
        vmovups   %zmm0, (%rax,%rdx)                                  #16.30
 [...]
```

DenseAlgebra/GEMM_Test_1_0_avx512

Assembly code

MatMatMultiplyBlockHelper.s

```
[...]
        vmovups   192(%rax,%rdx), %zmm3                           #14.41
        xorl      %r8d, %r8d                                      #10.5
        vmovups   128(%rax,%rdx), %zmm2                           #14.41
        vmovups   64(%rax,%rdx), %zmm1                            #14.41
        vmovups   (%rax,%rdx), %zmm0                              #14.41
                                # LOE rax rdx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 cl zmm0
zmm1 zmm2 zmm3
..B1.3:                         # Preds ..B1.3 ..B1.2
                                # Execution
        vbroadcastss (%r9,%r10,4), %zmm4
        incq      %r10
        vfmadd231ps (%r8,%rsi), %zmm4, %zmm0
        vfmadd231ps 64(%r8,%rsi), %zmm4, %zm
        vfmadd231ps 128(%r8,%rsi), %zmm4, %zmm2                   #15.18
        vfmadd231ps 192(%r8,%rsi), %zmm4, %zmm3                   #15.18
        addq      $256, %r8                                       #10.5
        cmpq      $64, %r10                                       #10.5
        jb        ..B1.3        # Prob 98%                        #10.5
                                # LOE rax rdx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 cl zmm0
zmm1 zmm2 zmm3
..B1.4:                         # Preds ..B1.3
                                # Execution count [6.40e+01]
        incb      %cl                                             #9.5
        vmovups   %zmm3, 192(%rax,%rdx)                           #16.30
        vmovups   %zmm2, 128(%rax,%rdx)                           #16.30
        vmovups   %zmm1, 64(%rax,%rdx)                            #16.30
        vmovups   %zmm0, (%rax,%rdx)                              #16.30
 [...]
```

Read the 64 floats starting at bB[kk][0] using 4x 16-wide store ("move") instructions (the compiler does a bit extra loop reordering, hence the split of the "move" instructions)

Assembly code

MatMatMultiplyBlockHelper.s

DenseAlgebra/GEMM_Test_1_0_avx512

```
[...]
        vmovups   192(%rax,%rdx), %zmm3                         #14.41
        xorl      %r8d, %r8d                                    #10.5
        vmovups   128(%rax,%rdx), %zmm2                         #14.41
        vmovups   64(%rax,%rdx), %zmm1                          #14.41
        vmovups   (%rax,%rdx), %zmm0
                                  # LOE rax rdx rb
zmm1 zmm2 zmm3
..B1.3:                           # Preds ..B1.3 ..B1.2
                                  # Execution count [4.10e+03]
        vbroadcastss (%r9,%r10,4), %zmm4                        #13.40
        incq      %r10                                          #10.5
        vfmadd231ps (%r8,%rsi), %zmm4, %zmm0                    #15.18
        vfmadd231ps 64(%r8,%rsi), %zmm4, %zmm1                  #15.18
        vfmadd231ps 128(%r8,%rsi), %zmm4, %zmm2                 #15.18
        vfmadd231ps 192(%r8,%rsi), %zmm4, %zmm3                 #15.18
        addq      $256, %r8                                     #10.5
        cmpq      $64, %r10                                     #10.5
        jb        ..B1.3          # Prob 98%                    #10.5
                                  # LOE rax rdx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 cl zmm0
zmm1 zmm2 zmm3
..B1.4:                           # Preds ..B1.3
                                  # Execution count [6.40e+01]
        incb      %cl                                           #9.5
        vmovups   %zmm3, 192(%rax,%rdx)                         #16.30
        vmovups   %zmm2, 128(%rax,%rdx)                         #16.30
        vmovups   %zmm1, 64(%rax,%rdx)                          #16.30
        vmovups   %zmm0, (%rax,%rdx)                            #16.30
 [...]
```

Broadcast (replicate) the value of bA[ii][kk] $_0$ *into all 16 values of register %zmm4*

Assembly code

MatMatMultiplyBlockHelper.s

DenseAlgebra/GEMM_Test_1_0_avx512

```
[...]
        vmovups   192(%rax,%rdx), %zmm3                                #14.41
        xorl      %r8d, %r8d                                           #10.5
        vmovups   128(%rax,%rdx), %zmm2                                #14.41
        vmovups   64(%rax,%rdx), %zmm1                                 #14.41
        vmovups   (%rax,%rdx), %zmm0                                   #14.41
                                # LOE rax rdx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 cl zmm0
zmm1 zmm2 zmm3
..B1.3:                         # Preds ..B1.3 ..B1.2
                                # Execution count [4.10e+03]
        vbroadcastss (%r9,%r10,4), %zmm4                               #13.40
        incq      %r10                                                 #10.5
        vfmadd231ps (%r8,%rsi), %zmm4, %zmm0
        vfmadd231ps 64(%r8,%rsi), %zmm4, %zmm1
        vfmadd231ps 128(%r8,%rsi), %zmm4, %zmm2
        vfmadd231ps 192(%r8,%rsi), %zmm4, %zmm3
        addq      $256, %r8
        cmpq      $64, %r10
        jb        ..B1.3        # Prob 98%                             #10.5
                                # LOE rax rdx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 cl zmm0
zmm1 zmm2 zmm3
..B1.4:                         # Preds ..B1.3
                                # Execution count [6.40e+01]
        incb      %cl                                                  #9.5
        vmovups   %zmm3, 192(%rax,%rdx)                                #16.30
        vmovups   %zmm2, 128(%rax,%rdx)                                #16.30
        vmovups   %zmm1, 64(%rax,%rdx)                                 #16.30
        vmovups   %zmm0, (%rax,%rdx)                                   #16.30
 [...]
```

Perform fused-multiply-add operation on 64-values (with 4 instructions). Values of bC[ii][jj] are directly read/written from/to memory

Assembly code

MatMatMultiplyBlockHelper.s

DenseAlgebra/GEMM_Test_1_0_avx512

```
[...]
        vmovups   192(%rax,%rdx), %zmm3                          #14.41
        xorl      %r8d, %r8d                                     #10.5
        vmovups   128(%rax,%rdx), %zmm2                          #14.41
        vmovups   64(%rax,%rdx), %zmm1                           #14.41
        vmovups   (%rax,%rdx), %zmm0                             #14.41
                                # LOE rax rdx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 cl zmm0
zmm1 zmm2 zmm3
..B1.3:                         # Preds ..B1.3 ..B1.2
                                # Execution count [4.10e+03]
        vbroadcastss (%r9,%r10,4), %zmm4                         #13.40
        incq      %r10                                           #10.5
        vfmadd231ps (%r8,%rsi), %zmm4, %zmm0                     #15.18
        vfmadd231ps 64(%r8,%rsi), %zmm4, %zmm1                   #15.18
        vfmadd231ps 128(%r8,%rsi), %zmm4, %zmm2                  #15.18
        vfmadd231ps 192(%r8,%rsi), %zmm4, %zmm3                  #15.18
        addq      $256, %r8                                      #10.5
        cmpq      $64, %r10                                      #10.5
        jb        ..B1.3        # Prob 98%
                                # LOE rax rdx rbx rbp                                        0
zmm1 zmm2 zmm3
..B1.4:                         # Preds ..B1.3
                                # Execution count [6.40e+01]
        incb      %cl                                            #9.5
        vmovups   %zmm3, 192(%rax,%rdx)                          #16.30
        vmovups   %zmm2, 128(%rax,%rdx)                          #16.30
        vmovups   %zmm1, 64(%rax,%rdx)                           #16.30
        vmovups   %zmm0, (%rax,%rdx)                             #16.30
 [...]
```

Overall : Slightly better code density, data/register reuse than what we had before (with OpenMP)

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_0_avx512

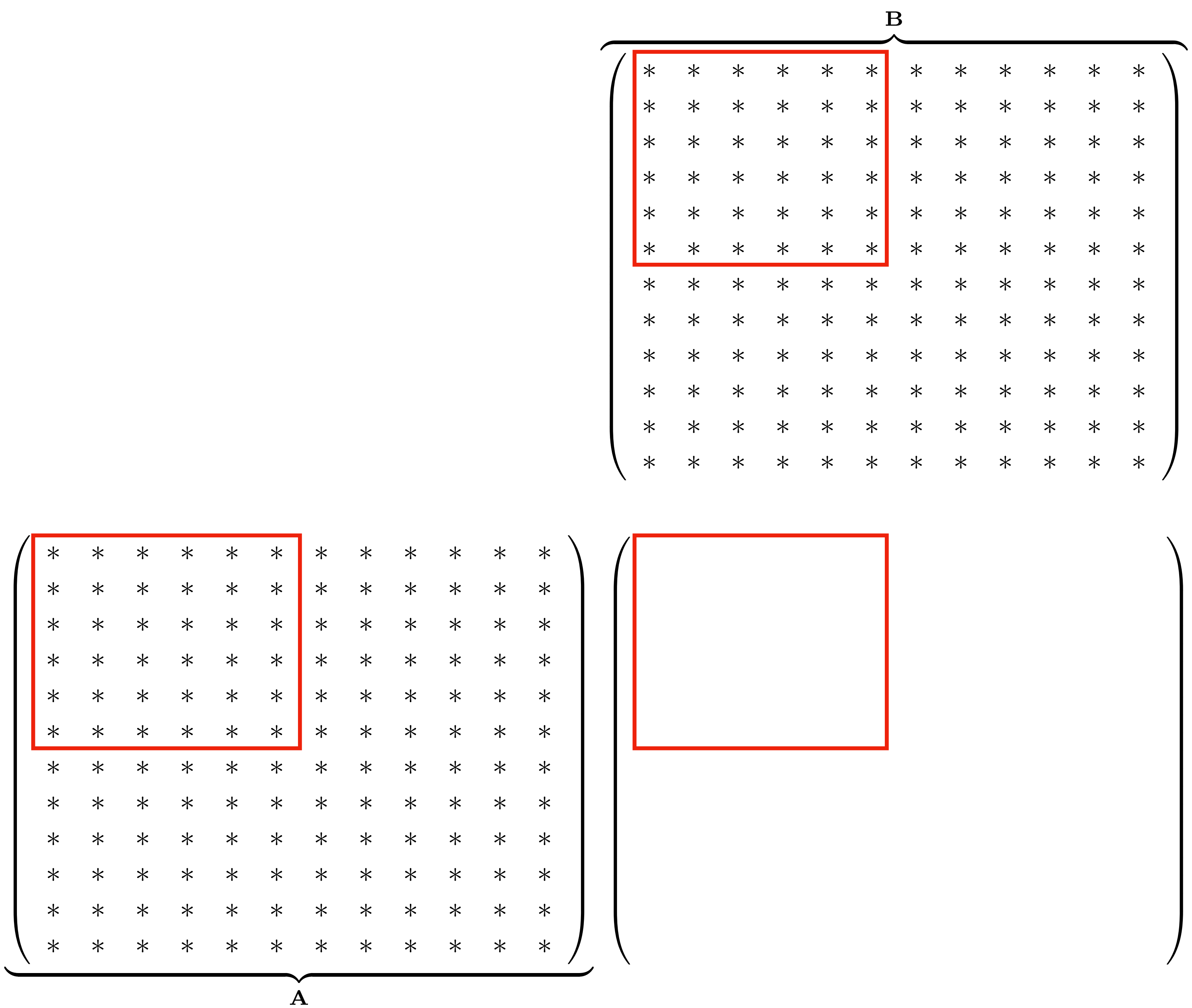
```
#include "MatMatMultiplyBlockHelper.h"
#include "immintrin.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of SIMD vectors

    for (int ii = 0; ii < BLOCK_SIZE; ii++)
    for (int kk = 0; kk < BLOCK_SIZE; kk++)
        for (int jj = 0; jj < BLOCK_SIZE; jj += nW) {
            __m512 vB = _mm512_load_ps(&bB[kk][jj]);
            __m512 vA = _mm512_set1_ps(bA[ii][kk]);
            __m512
            vC = _m
            _mm512_
        }
}
```

1.6x the runtime of MKL code!

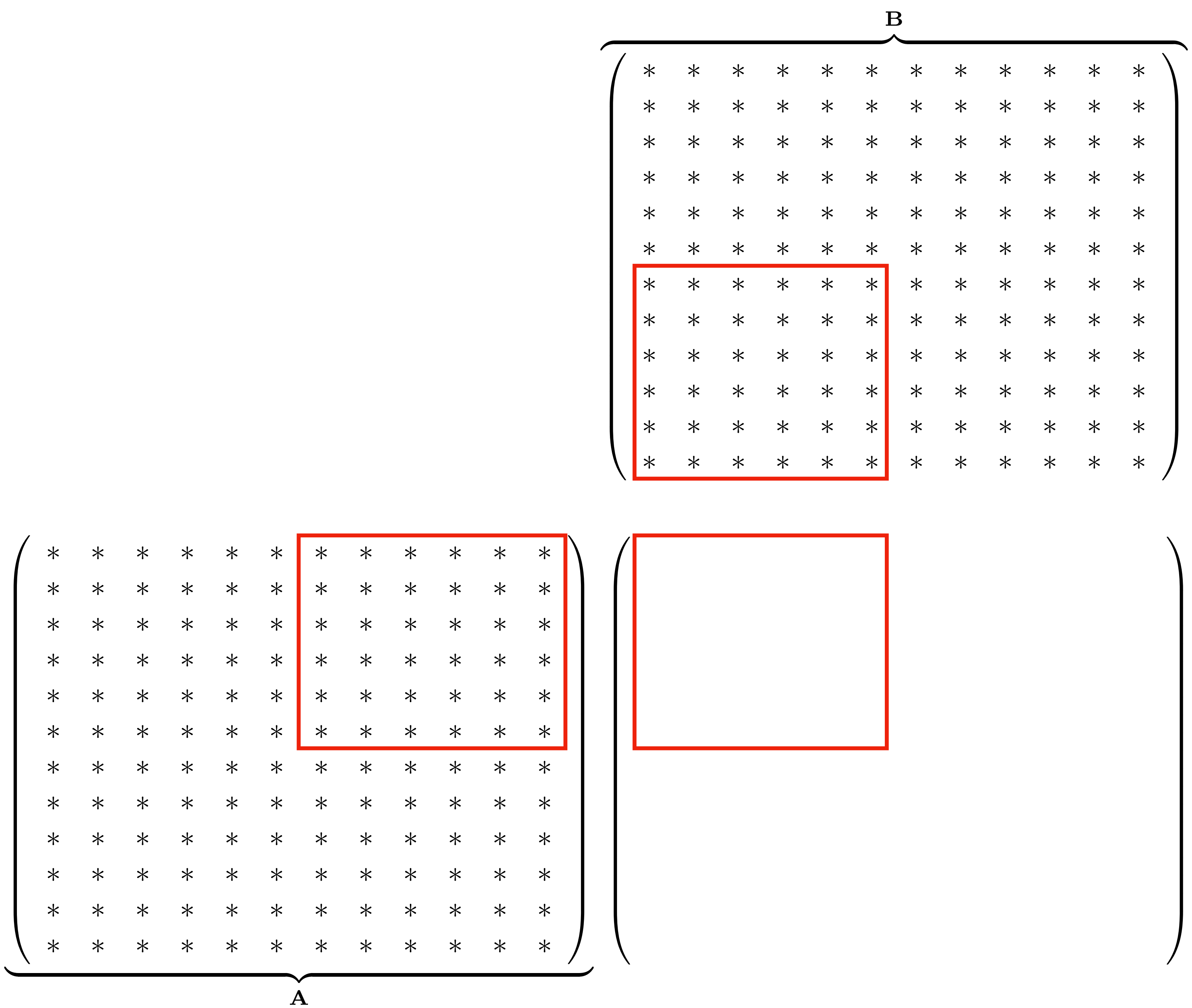
Execution:

```
Running candidate kernel for correctness test ... [Elapsed time : 36.7904ms]
Running reference kernel for correctness test ... [Elapsed time : 40.292ms]
Discrepancy between two methods : 0.000154495
Running kernel for performance run # 1 ... [Elapsed time : 19.5309ms]
Running kernel for performance run # 2 ... [Elapsed time : 19.0874ms]
Running kernel for performance run # 3 ... [Elapsed time : 19.2922ms]
Running kernel for performance run # 4 ... [Elapsed time : 19.1488ms]
Running kernel for performance run # 5 ... [Elapsed time : 19.2003ms]
Running kernel for performance run # 6 ... [Elapsed time : 19.8611ms]
Running kernel for performance run # 7 ... [Elapsed time : 19.0866ms]
Running kernel for performance run # 8 ... [Elapsed time : 19.1242ms]
[...]
```

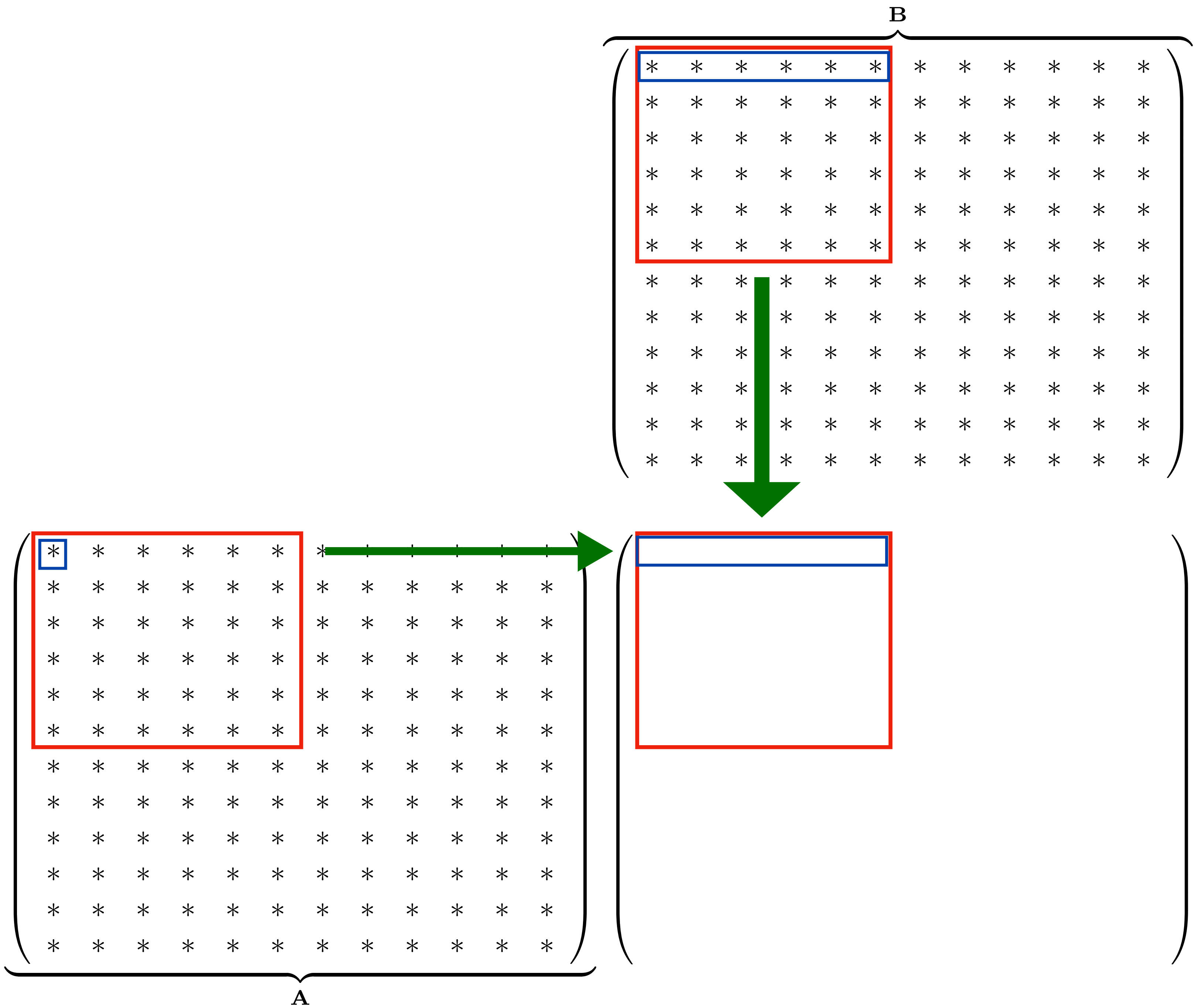
Blocking for vectorization (once again …)



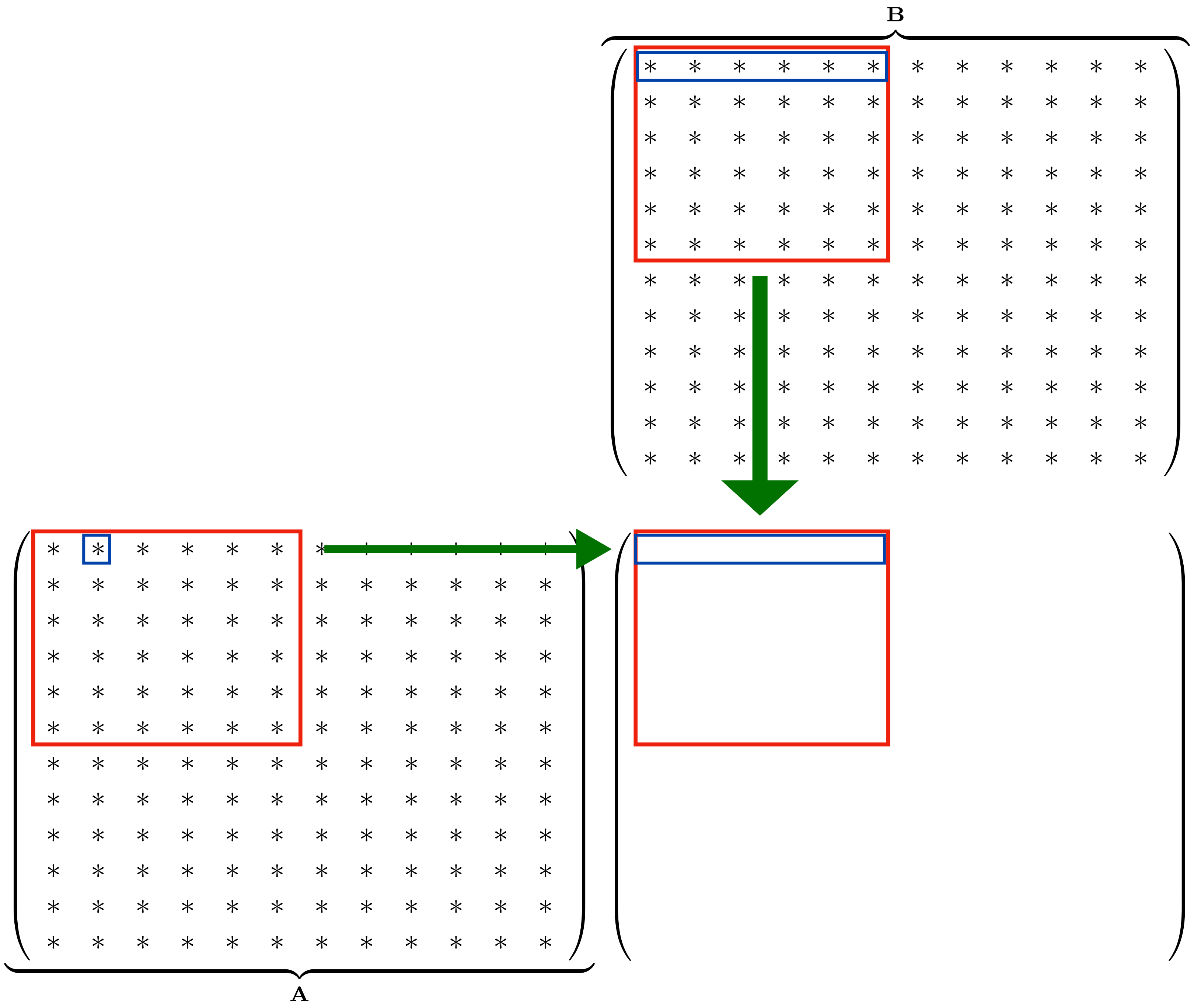
Blocking for vectorization (once again ...)



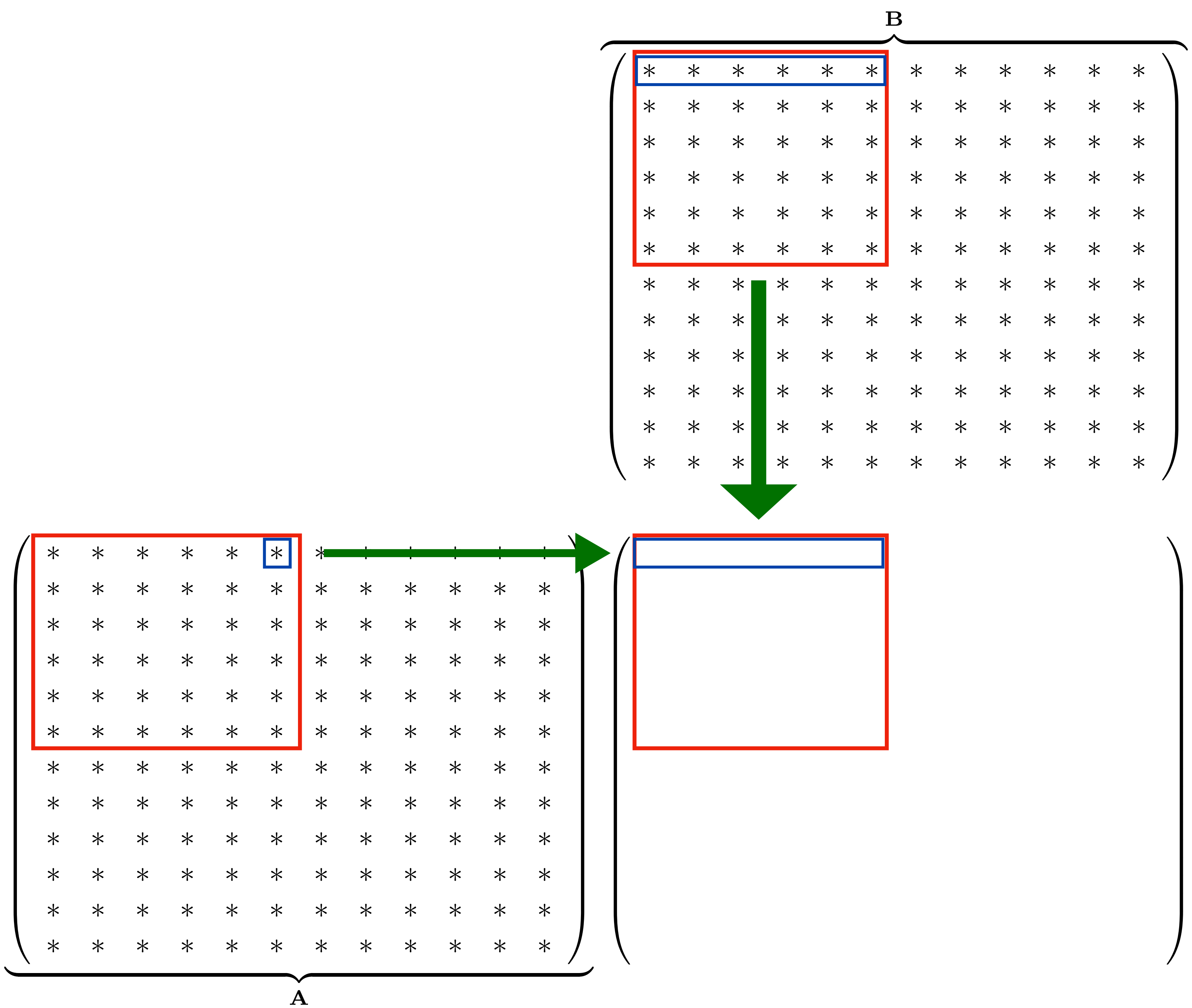
Blocking for vectorization (once again ...)



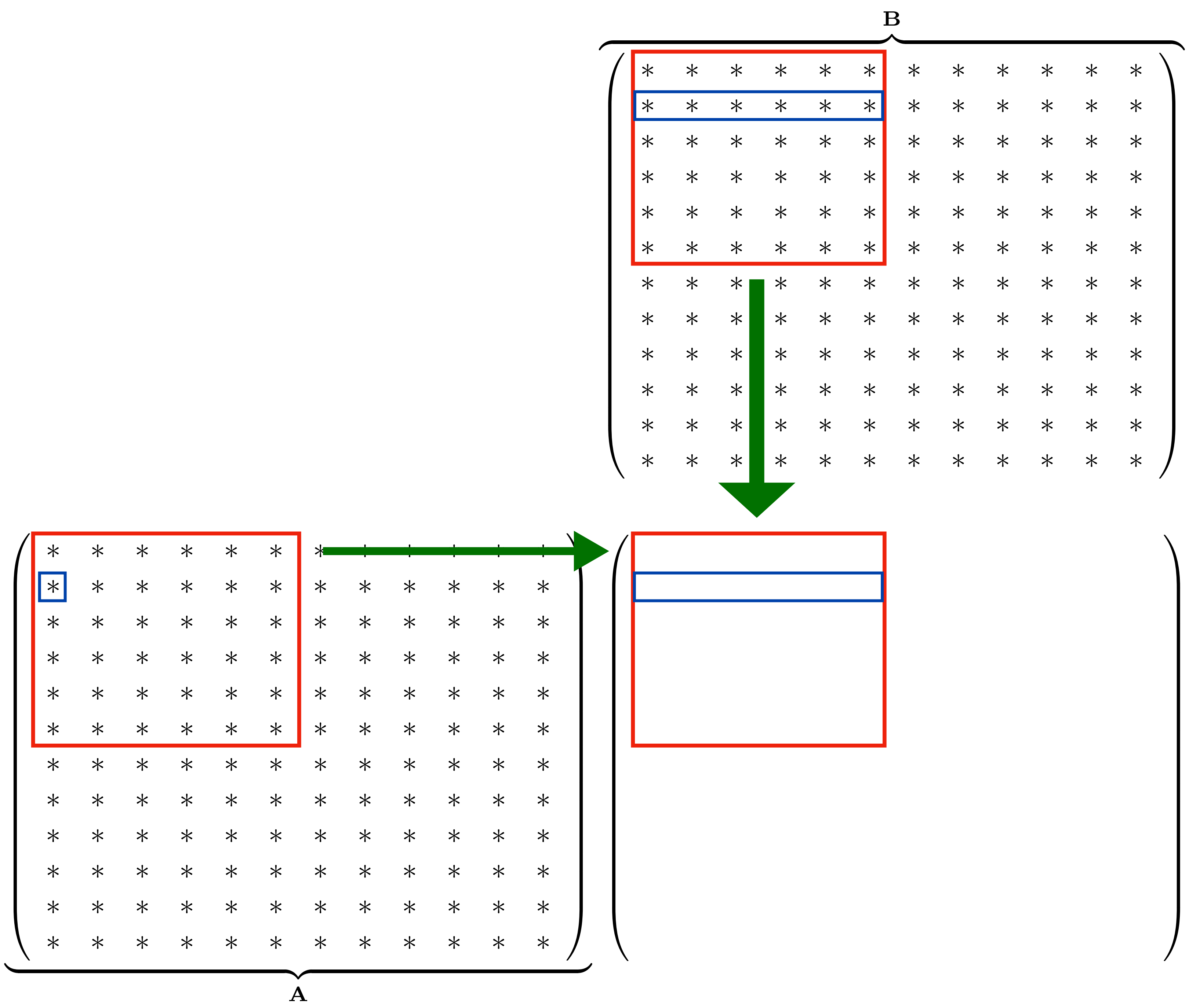
Blocking for vectorization (once again ...)



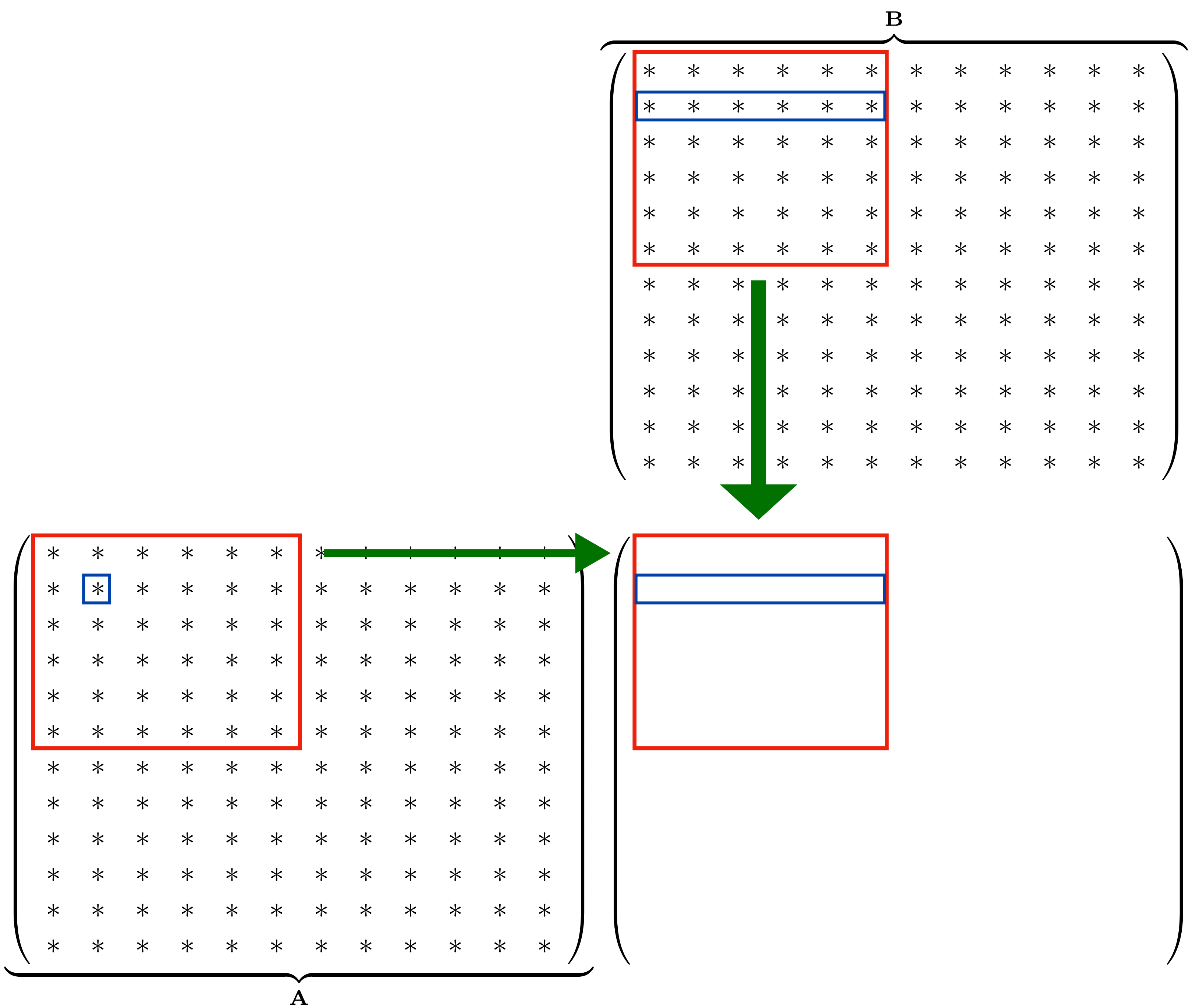
Blocking for vectorization (once again ...)



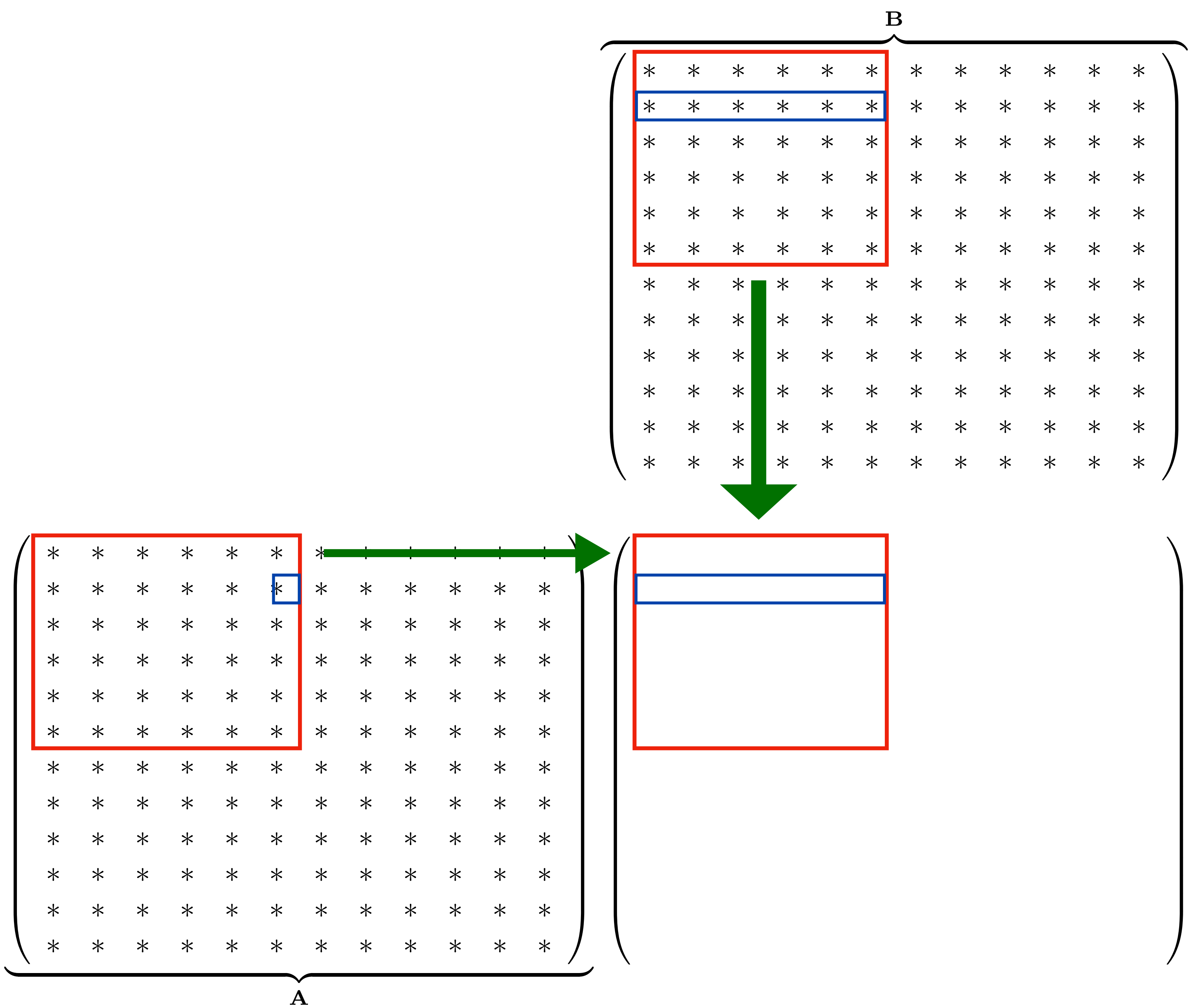
Blocking for vectorization (once again ...)



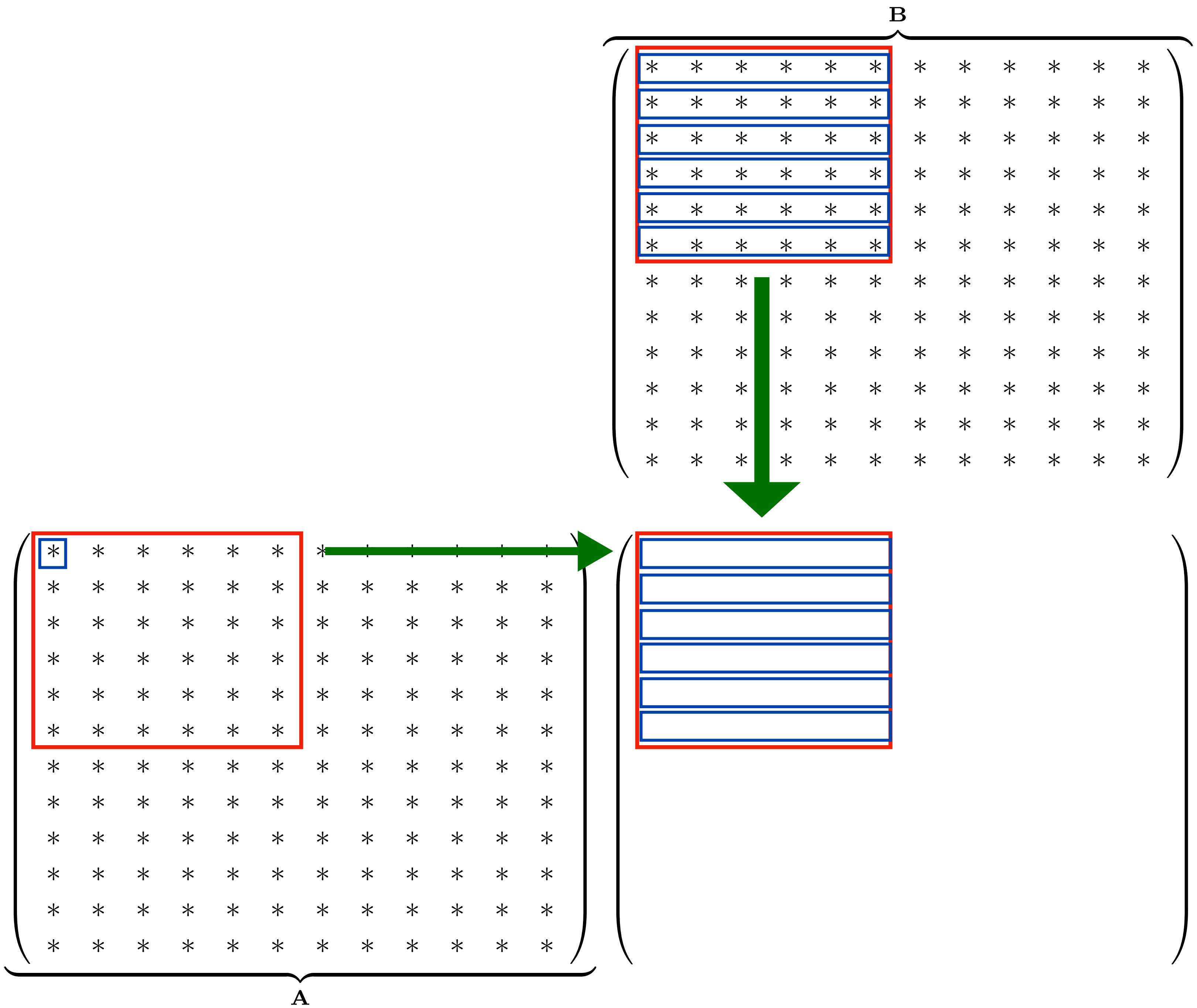
Blocking for vectorization (once again ...)



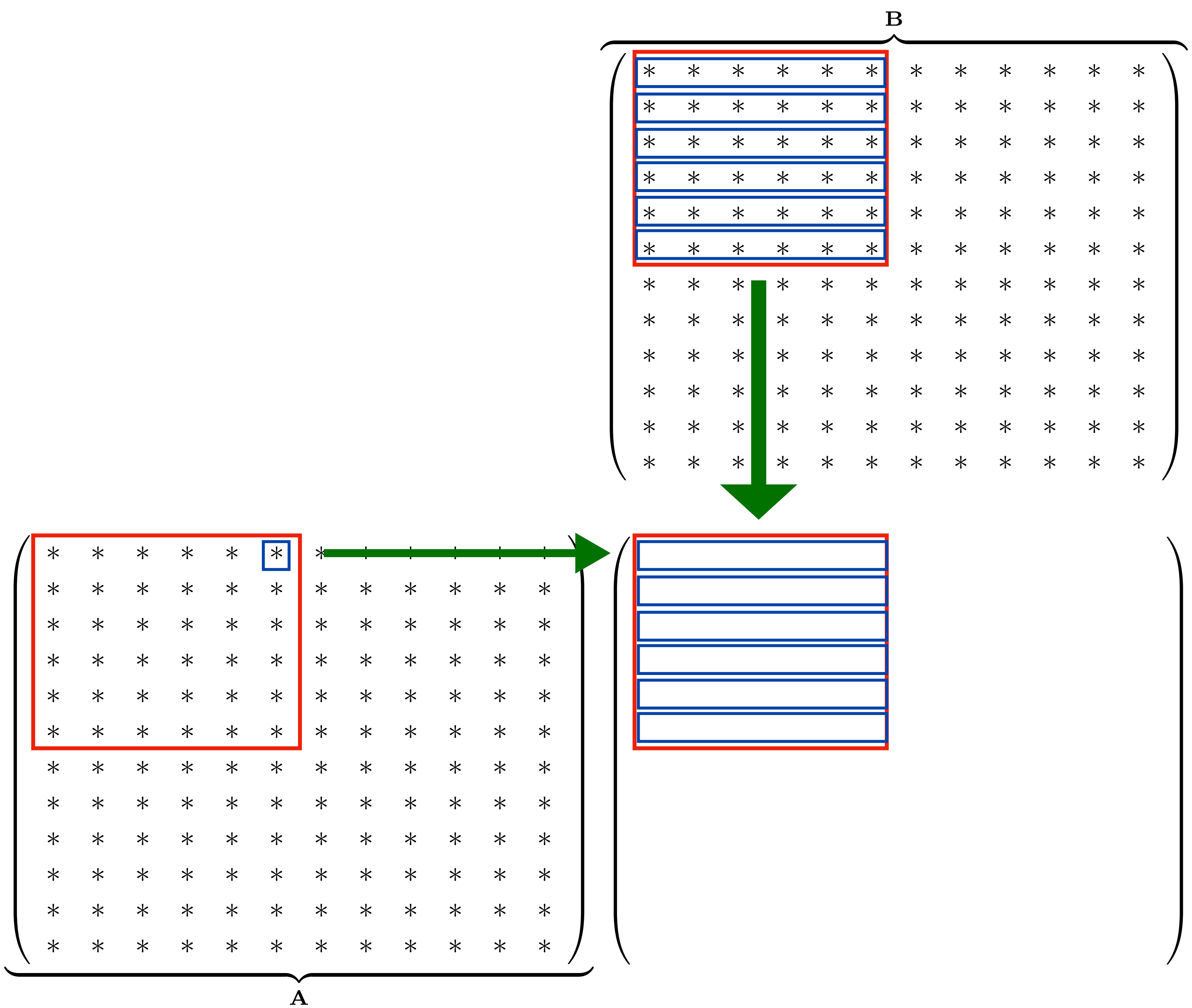
Blocking for vectorization (once again ...)



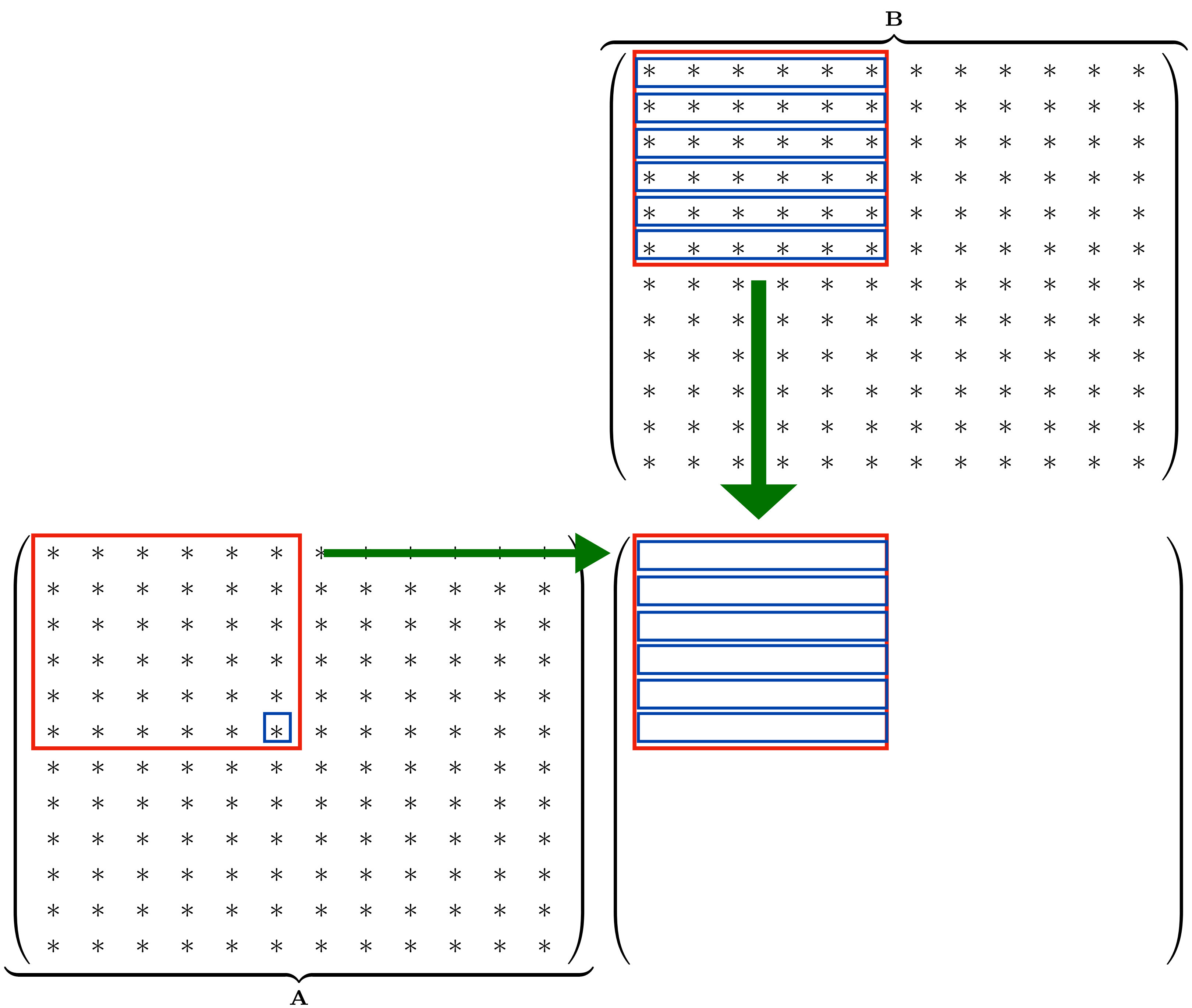
Blocking for vectorization (once again ...)



Blocking for vectorization (once again ...)



Blocking for vectorization (once again ...)



Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_1

```
void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Entries in a SIMD vector (for AVX512; use 8 for AVX2)
    static constexpr int nB = BLOCK_SIZE / nW; // Number of blocks

    using const_blocked_matrix_t = const float (&) [nB][nW][nB][nW];
    using blocked_matrix_t = float (&) [nB][nW][nB][nW];

    // Matrix bA has indices [i][k], or in block form (bbA) [bi][ii][bk][kk]
    // Matrix bB has indices [k][j], or in block form (bbB) [bk][kk][bj][jj]
    // matrix bC has indices [i][j], or in block form (bbC) [bi][ii][bj][jj]

    auto bbA = reinterpret_cast<const_blocked_matrix_t>(bA[0][0]);
    auto bbB = reinterpret_cast<const_blocked_matrix_t>(bB[0][0]);
    auto bbC = reinterpret_cast<blocked_matrix_t>(bC[0][0]);

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

    {
        for (int kk = 0; kk < nW; kk++)
        for (int ii = 0; ii < nW; ii++)
#pragma omp simd
        for (int jj = 0; jj < nW; jj++)
                bbC[bi][ii][bj][jj] += bbA[bi][ii][bk][kk] * bbB[bk][kk][bj][jj];
    }
}
```

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_1

```
void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Entries in a SIMD vector (for AVX512; use 8 for AVX2)
    static constexpr int nB = BLOCK_SIZE / nW; // Number of blocks

    using const_blocked_matrix_t = const float (&) [nB][nW][nB][nW];
    using blocked_matrix_t = float (&) [nB][nW][nB][nW];

    // Matrix bA has indices [i][k], or in block form (bbA) [bi][ii][bk][kk]
    // Matrix bB has indices [k][j], or in block form (bbB) [bk][kk][bj][jj]
    // matrix bC has indices [i][j], or in block form (bbC) [bi][ii][bj][jj]

    auto bbA = reinterpret_cast<const_blocked_matrix_t>(bA[0][0]);
    auto bbB = reinterpret_cast<const_blocked_matrix_t>(bB[0][0]);
    auto bbC = reinterpret_cast<blocked_matrix_t>(bC[0][0]);

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

    {
        for (int kk = 0; kk < nW; kk++)
        for (int ii = 0; ii < nW; ii++)
#pragma omp simd
        for (int jj = 0; jj < nW; jj++)
            bbC[bi][ii][bj][jj] += bbA[bi][ii][bk][kk] * bbB[bk][kk][bj][jj];
    }
}
```

Using blocking, once again …
(for the purposes of vectorization this time)

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_1

```
void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Entries in a SIMD vector (for AVX512; use 8 for AVX2)
    static constexpr int nB = BLOCK_SIZE / nW; // Number of blocks

    using const_blocked_matrix_t = const float (&) [nB][nW][nB][nW];
    using blocked_matrix_t = float (&) [nB][nW][nB][nW];

    // Matrix bA has indices [i][k], or in block form (bbA) [bi][ii][bk][kk]
    // Matrix bB has indices [k][j], or in block form (bbB) [bk][kk][bj][jj]
    // matrix bC has indices [i][j], or in block form (bbC) [bi][ii][bj][jj]

    auto bbA = reinterpret_cast<const_blocked_matrix_t>(bA[0][0]);
    auto bbB = reinterpret_cast<const_blocked_matrix_t>(bB[0][0]);
    auto bbC = reinterpret_cast<blocked_matrix_t>(bC[0][0]);

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

    {
        for (int kk = 0; kk < nW; kk++)
        for (int ii = 0; ii < nW; ii++)
#pragma omp simd
        for (int jj = 0; jj < nW; jj++)
                bbC[bi][ii][bj][jj] += bbA[bi][ii][bk][kk] * bbB[bk][kk][bj][jj];
    }
}
```

Using blocking, once again …
(for the purposes of vectorization this time)

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_1

```
void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Entries in a SIMD vector (for AVX512; use 8 for AVX2)
    static constexpr int nB = BLOCK_SIZE / nW; // Number of blocks

    using const_blocked_matrix_t = const float (&) [nB][nW][nB][nW];
    using blocked_matrix_t = float (&) [nB][nW][nB][nW];

    // Matrix bA has indices [i][k], or in block form (bbA) [bi][ii][bk][kk]
    // Matrix bB has indices [k][j], or in block form (bbB) [bk][kk][bj][jj]
    // matrix bC has indices [i][j], or in block form (bbC) [bi][ii][bj][jj]

    auto bbA = reinterpret_cast<const_blocked_matrix_t>(bA[0][
    auto bbB = reinterpret_cast<const_blocked_matrix_t>(bB[0][0]);
    auto bbC = reint

    for (int bi = 0;
    for (int bj = 0;
    for (int bk = 0;

    {
        for (int kk
        for (int ii
#pragma omp simd
        for (int jj
                bbC[
    }
}
```

1.55x the runtime of MKL code!

Execution:

```
Running candidate kernel for correctness test ... [Elapsed time : 35.129ms]
Running reference kernel for correctness test ... [Elapsed time : 41.255ms]
Discrepancy between two methods : 0.00015349
Running kernel for performance run # 1 ... [Elapsed time : 18.9509ms]
Running kernel for performance run # 2 ... [Elapsed time : 18.2873ms]
Running kernel for performance run # 3 ... [Elapsed time : 18.3924ms]
Running kernel for performance run # 4 ... [Elapsed time : 18.2958ms]
Running kernel for performance run # 5 ... [Elapsed time : 18.8063ms]
Running kernel for performance run # 6 ... [Elapsed time : 18.8611ms]
Running kernel for performance run # 7 ... [Elapsed time : 18.7826ms]
Running kernel for performance run # 8 ... [Elapsed time : 18.6212ms]
[...]
```

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_2_avx512

```
void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of a SIMD vector
    static constexpr int nB = BLOCK_SIZE/nW;
    using const_blocked_matrix_t = const float (&) [nB][nW][nB][nW];
    using blocked_matrix_t = float (&) [nB][nW][nB][nW];
    auto bbA = reinterpret_cast<const_blocked_matrix_t>(bA[0][0]);
    auto bbB = reinterpret_cast<const_blocked_matrix_t>(bB[0][0]);
    auto bbC = reinterpret_cast<blocked_matrix_t>(bC[0][0]);

    for (int bi = 0; bi < nB; bi++) for (int bj = 0; bj < nB; bj++)
    {
        __m512 vC[nW];
        for (int ii = 0; ii < nW; ii++)
            vC[ii] = _mm512_load_ps(&bbC[bi][ii][bj][0]);

        for (int bk = 0; bk < nB; bk++) {
            __m512 vB[nW];
            for (int kk = 0; kk < nW; kk++)
                vB[kk] = _mm512_load_ps(&bbB[bk][kk][bj][0]);

            for (int ii = 0; ii < nW; ii++) for (int kk = 0; kk < nW; kk++)
                vC[ii] = _mm512_fmadd_ps(_mm512_set1_ps(bbA[bi][ii][bk][kk]), vB[kk], vC[ii]); }

        for (int ii = 0; ii < nW; ii++)
            _mm512_store_ps(&bbC[bi][ii][bj][0],vC[ii]);
    }
}
```

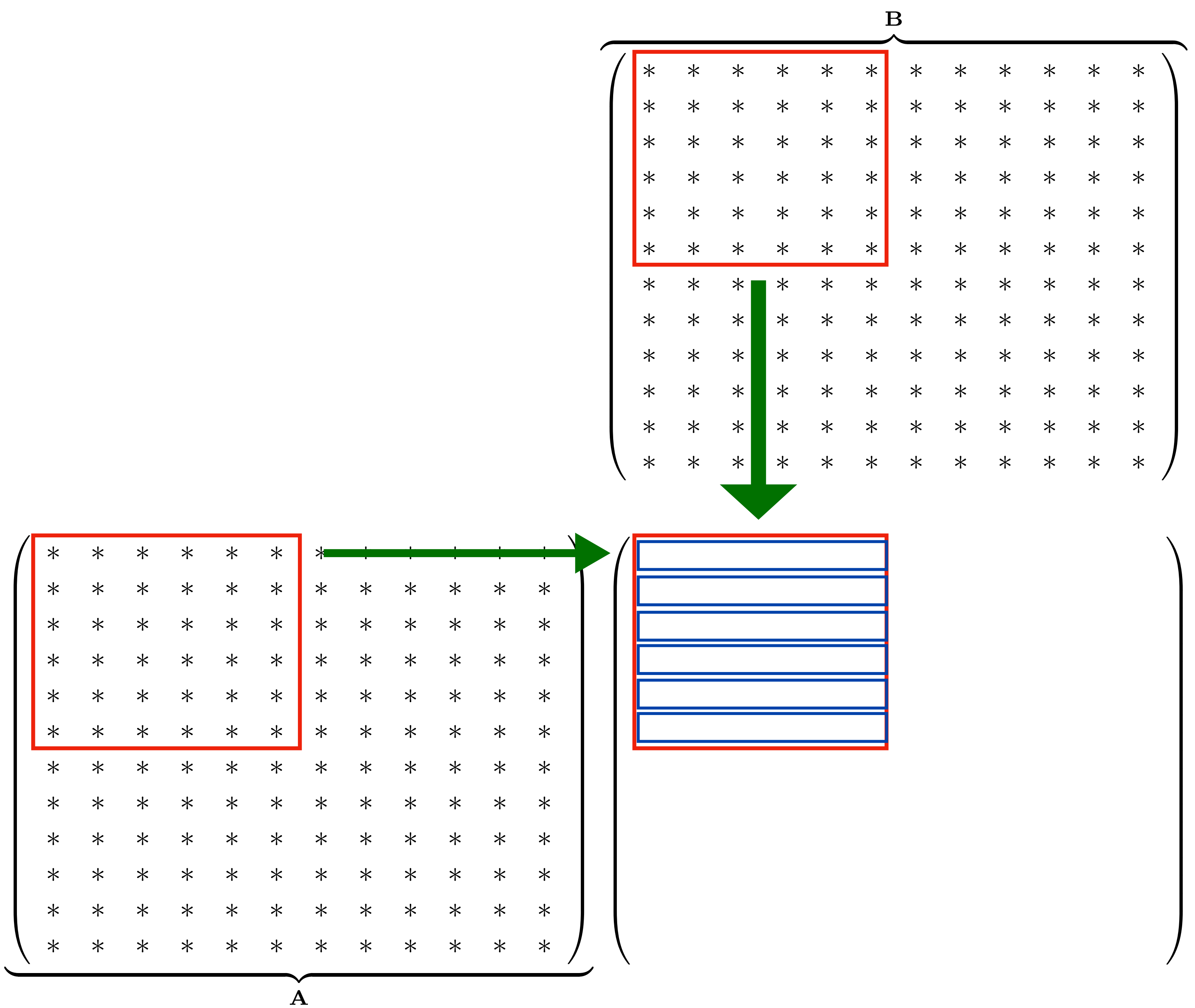
Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_2_avx512

```
void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of a SIMD vector
    static constexpr int nB = BLOCK_SIZE/nW;
    using const_blocked_matrix_t = const float (&) [nB][nW][nB][nW];
    using blocked_matrix_t = float (&) [nB][nW][nB][nW];
    auto bbA = reinterpret_cast<const_blocked_matrix_t>(bA[0][0]);
    auto bbB = reinterpret_cast<const_blocked_matrix_t>(bB[0][0]);
    auto bbC = reinterpret_cast<blocked_matrix_t>(bC[0][0]);

    for (int bi = 0; bi < nB; bi++) for (int bj = 0; bj < nB; bj++)
    {
        __m512 vC[nW];
        for (int ii = 0; ii < nW; ii++)
            vC[ii] = _mm512_load_ps(&bbC[bi][ii][bj][0]);

        for (int bk = 0; bk < nB; bk++) {
            __m512 vB[nW];
            for (int kk = 0; kk < nW; kk++)
                vB[kk] = _mm512_load_ps(&bbB[bk][kk][bj][0]);

            for (int ii = 0; ii < nW; ii++) for (int kk = 0; kk < nW; kk++)
                vC[ii] = _mm512_fmadd_ps(_mm512_set1_ps(bbA[bi][ii][bk][kk]), vB[kk], vC[ii]); }

        for (int ii = 0; ii < nW; ii++)
            _mm512_store_ps(&bbC[bi][ii][bj][0],vC[ii]);
    }
}
```

- Define 16 “registers” vC[0] through vC[15] which will hold the contents of the C block

Blocking for vectorization (once again ...)



Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_2_avx512

```
void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of a SIMD vector
    static constexpr int nB = BLOCK_SIZE/nW;
    using const_blocked_matrix_t = const float (&) [nB][nW][nB][nW];
    using blocked_matrix_t = float (&) [nB][nW][nB][nW];
    auto bbA = reinterpret_cast<const_blocked_matrix_t>(bA[0][0]);
    auto bbB = reinterpret_cast<const_blocked_matrix_t>(bB[0][0]);
    auto bbC = reinterpret_cast<blocked_matrix_t>(bC[0][0]);

    for (int bi = 0; bi < nB; bi++) for (int bj = 0; bj < nB; bj++)
    {
        __m512 vC[nW];
        for (int ii = 0; ii < nW; ii++)
            vC[ii] = _mm512_load_ps(&bbC[bi][ii][bj][0]);

        for (int bk = 0; bk < nB; bk++) {
            __m512 vB[nW];
            for (int kk = 0; kk < nW; kk++)
                vB[kk] = _mm512_load_ps(&bbB[bk][

            for (int ii = 0; ii < nW; ii++) for (
                vC[ii] = _mm512_fmadd_ps(_mm512_s

        for (int ii = 0; ii < nW; ii++)
            _mm512_store_ps(&bbC[bi][ii][bj][0],v
    }
}
```

*- Define 16 "registers" vC[0] through vC[15] which will hold the contents of the **C** block*

*- Populate them with the previous values from the blocked matrix **bbC***

<u>Note:</u> We are doing this outside the "bk" loop! No need to re-read C every time

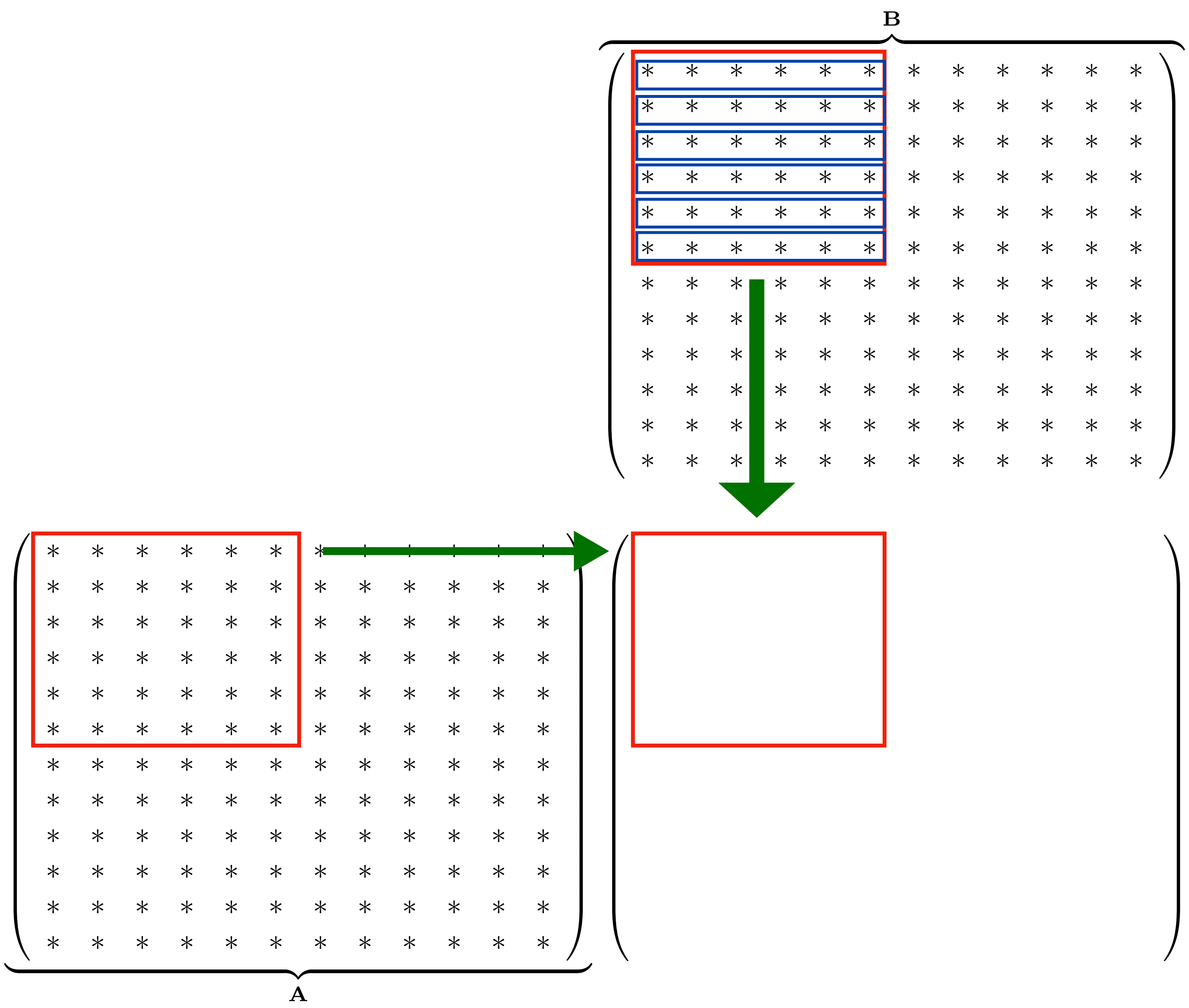
Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_2_avx512

```
void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of a SIMD vector
    static constexpr int nB = BLOCK_SIZE/nW;
    using const_blocked_matrix_t = const float (&) [nB][nW][nB][nW];
    using blocked_matrix_t = float (&) [nB][nW][nB][nW];
    auto bbA = reinterpret_cast<const_blocked_matrix_t>(bA[0][0]);
    auto bbB = reinterpret_cast<const_bl
    auto bbC = reinterpret_cast<blocked_

    for (int bi = 0; bi < nB; bi++) for
    {
        __m512 vC[nW];
        for (int ii = 0; ii < nW; ii++)
            vC[ii] = _mm512_load_ps(&bbC[bi][ii][bj][0]);

        for (int bk = 0; bk < nB; bk++) {
            __m512 vB[nW];
            for (int kk = 0; kk < nW; kk++)
                vB[kk] = _mm512_load_ps(&bbB[bk][kk][bj][0]);

            for (int ii = 0; ii < nW; ii++) for (int kk = 0; kk < nW; kk++)
                vC[ii] = _mm512_fmadd_ps(_mm512_set1_ps(bbA[bi][ii][bk][kk]), vB[kk], vC[ii]); }

        for (int ii = 0; ii < nW; ii++)
            _mm512_store_ps(&bbC[bi][ii][bj][0],vC[ii]);
    }
}
```

*- Similarly, define 16 "registers" vB[0] through vB[15] which will hold the contents of the **B** block*
- Read their values just once, <u>before</u> iterating through the matrix A (the ii & kk loop)

Blocking for vectorization (once again ...)



Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_0_avx512

```
#include "MatMatMultiplyBlockHelper.h"
#include "immintrin.h"

void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of SIMD vectors

    for (int ii = 0; ii < BLOCK_SIZE; ii++)
    for (int kk = 0; kk < BLOCK_SIZE; kk++)
        for (int jj = 0; jj < BLOCK_SIZE; jj += nW) {
            __m512 vB = _mm512_load_ps(&bB[kk][jj]);
            __m512 vA = _mm512_set1_ps(bA[ii][kk]);
            __m512 vC = _mm512_load_ps(&bC[ii][jj]);
            vC = _mm512_fmadd_ps(vA, vB, vC);
            _mm512_store_ps(&bC[ii][jj], vC);
        }
}
```

Compare to prior version (B is read repeatedly)

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_2_avx512

```
void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of a SIMD vector
    static constexpr int nB = BLOCK_SIZE/nW;
    using const_blocked_matrix_t = const float (&) [nB][nW][nB][nW];
    using blocked_matrix_t = float (&) [nB][nW][nB][nW];
    auto bbA = reinterpret_cast<const_blocked_matrix_t>(bA[0][0]);
    auto bbB = reinterpret_cast<const_blocked_matrix_t>(bB[0][0]);
    auto bbC = reinterpret_cast<blocked_matrix_t>(bC[0][0]);

    for (int bi = 0; bi < nB; bi++) for (int bj = 0; bj < nB; bj++)
    {
        __m512 vC[nW];
        for (int ii = 0; ii < nW; ii++)
            vC[ii] = _mm512_load_ps(&bbC[bi][ii][b

        for (int bk = 0; bk < nB; bk++) {
            __m512 vB[nW];
            for (int kk = 0; kk < nW; kk++)
                vB[kk] = _mm512_load_ps(&bbB[bk][kk][bj][0]);

            for (int ii = 0; ii < nW; ii++) for (int kk = 0; kk < nW; kk++)
                vC[ii] = _mm512_fmadd_ps(_mm512_set1_ps(bbA[bi][ii][bk][kk]), vB[kk], vC[ii]); }

        for (int ii = 0; ii < nW; ii++)
            _mm512_store_ps(&bbC[bi][ii][bj][0],vC[ii]);
    }
}
```

*- Perform fused multiply-add operation on registers for **B & C***

*- Inline the "broadcast" operation for the corresponding entry of **A***

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_2_avx512

```
void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of a SIMD vector
    static constexpr int nB = BLOCK_SIZE/nW;
    using const_blocked_matrix_t = const float (&) [nB][nW][nB][nW];
    using blocked_matrix_t = float (&) [nB][nW][nB][nW];
    auto bbA = reinterpret_cast<const_blocked_matrix_t>(bA[0][0]);
    auto bbB = reinterpret_cast<const_blocked_matrix_t>(bB[0][0]);
    auto bbC = reinterpret_cast<blocked_matrix_t>(bC[0][0]);

    for (int bi = 0; bi < nB; bi++) for (int bj = 0; bj < nB; bj++)
    {
        __m512 vC[nW];
        for (int ii = 0; ii < nW; ii++)
            vC[ii] = _mm512_load_ps(&bbC[bi][ii][bj][0]);

        for (int bk = 0; bk < nB; bk++) {
            __m512 vB[nW];
            for (int kk = 0; kk < nW; kk++)
                vB[kk] = _mm512_load_ps(&bbB[bk][kk][bj][0]);

            for (int ii = 0; ii < nW; ii++) for (int kk = 0; kk < nW; kk++)
                vC[ii] = _mm512_fmadd_ps(_mm512_set1_ps(bbA[bi][ii][bk][kk]), vB[kk], vC[ii]); }

        for (int ii = 0; ii < nW; ii++)
            _mm512_store_ps(&bbC[bi][ii][bj][0],vC[ii]);
    }
}
```

*Store the result back to **bbC** at the end of the loops for bk, ii and kk*

Assembly code

MatMatMultiplyBlockHelper.s

DenseAlgebra/GEMM_Test_1_2_avx512

```
[...]
..B1.4:                         # Preds ..B1.6 ..B1.3
                                # Execution count [6.40e+01]
        vmovups   (%r10,%r11), %zmm15                        #30.42
        xorb      %r15b, %r15b                               #33.13
        vmovups   256(%r10,%r11), %zmm14                     #30.42
[.. .]
        vmovups   3584(%r10,%r11), %zmm1                     #30.42
        vmovups   3840(%r10,%r11), %zmm0                     #30.42
        xorl      %r14d, %r14d                               #33.13
        movq      %rdx, %r13                                 #34.26
..B1.5:                         # Preds ..B1.5 ..B1.4
                                # Execution count [1.02e+03]
        vbroadcastss (%r13), %zmm16                          #34.26
        incb      %r15b                                      #33.13
        vfmadd213ps 64(%rsp,%r14), %zmm15, %zmm16            #34.26
        vfmadd231ps 4(%r13){1to16}, %zmm14, %zmm16           #34.26
        vfmadd231ps 8(%r13){1to16}, %zmm13, %zmm16           #34.26
        vfmadd231ps 12(%r13){1to16}, %zmm12, %zmm16          #34.26
        vfmadd231ps 16(%r13){1to16}, %zmm11, %zmm16          #34.26
        vfmadd231ps 20(%r13){1to16}, %zmm10, %zmm16          #34.26
        vfmadd231ps 24(%r13){1to16}, %zmm9, %zmm16           #34.26
        vfmadd231ps 28(%r13){1to16}, %zmm8, %zmm16           #34.26
        vfmadd231ps 32(%r13){1to16}, %zmm7, %zmm16           #34.26
        vfmadd231ps 36(%r13){1to16}, %zmm6, %zmm16           #34.26
        vfmadd231ps 40(%r13){1to16}, %zmm5, %zmm16           #34.26
        vfmadd231ps 44(%r13){1to16}, %zmm4, %zmm16           #34.26
        vfmadd231ps 48(%r13){1to16}, %zmm3, %zmm16           #34.26
        vfmadd231ps 52(%r13){1to16}, %zmm2, %zmm16           #34.26
        vfmadd231ps 56(%r13){1to16}, %zmm1, %zmm16           #34.26
        vfmadd231ps 60(%r13){1to16}, %zmm0, %zmm16           #34.26
        addq      $256, %r13                                 #33.13
        vmovups   %zmm16, 64(%rsp,%r14)                      #34.17
 [...]
```

Assembly code

MatMatMultiplyBlockHelper.s

DenseAlgebra/GEMM_Test_1_2_avx512

```
[...]
..B1.4:                         # Preds ..B1.6 ..B1.3
                                # Execution count [6.40e+01]
        vmovups   (%r10,%r11), %zmm15                             #30.42
        xorb      %r15b, %r15b                                    #33.13
        vmovups   256(%r10,%r11), %zmm14                          #30.42
[.. .]
        vmovups   3584(%r10,%r11), %zmm1
        vmovups   3840(%r10,%r11), %zmm0
        xorl      %r14d, %r14d                                    #33.13
        movq      %rdx, %r13                                      #34.26
..B1.5:                         # Preds ..B1.5 ..B1.4
                                # Execution count [1.02e+03]
        vbroadcastss (%r13), %zmm16                               #34.26
        incb      %r15b                                           #33.13
        vfmadd213ps 64(%rsp,%r14), %zmm15, %zmm16                 #34.26
        vfmadd231ps 4(%r13){1to16}, %zmm14, %zmm16                #34.26
        vfmadd231ps 8(%r13){1to16}, %zmm13, %zmm16                #34.26
        vfmadd231ps 12(%r13){1to16}, %zmm12, %zmm16               #34.26
        vfmadd231ps 16(%r13){1to16}, %zmm11, %zmm16               #34.26
        vfmadd231ps 20(%r13){1to16}, %zmm10, %zmm16               #34.26
        vfmadd231ps 24(%r13){1to16}, %zmm9, %zmm16                #34.26
        vfmadd231ps 28(%r13){1to16}, %zmm8, %zmm16                #34.26
        vfmadd231ps 32(%r13){1to16}, %zmm7, %zmm16                #34.26
        vfmadd231ps 36(%r13){1to16}, %zmm6, %zmm16                #34.26
        vfmadd231ps 40(%r13){1to16}, %zmm5, %zmm16                #34.26
        vfmadd231ps 44(%r13){1to16}, %zmm4, %zmm16                #34.26
        vfmadd231ps 48(%r13){1to16}, %zmm3, %zmm16                #34.26
        vfmadd231ps 52(%r13){1to16}, %zmm2, %zmm16                #34.26
        vfmadd231ps 56(%r13){1to16}, %zmm1, %zmm16                #34.26
        vfmadd231ps 60(%r13){1to16}, %zmm0, %zmm16                #34.26
        addq      $256, %r13                                      #33.13
        vmovups   %zmm16, 64(%rsp,%r14)                           #34.17
 [...]
```

- All of B pre-loaded into registers (%zmm0 through %zmm15)

Assembly code

MatMatMultiplyBlockHelper.s

DenseAlgebra/GEMM_Test_1_2_avx512

```
[...]
..B1.4:                         # Preds ..B1.6 ..B1.3
                                # Execution count [6.40e+01]
        vmovups   (%r10,%r11), %zmm15                          #30.42
        xorb      %r15b, %r15b                                 #33.13
        vmovups   256(%r10,%r11), %zmm14                       #30.42
[.. .]
        vmovups   3584(%r10,%r11), %zmm1                       #30.42
        vmovups   3840(%r10,%r11), %zmm0                       #30.42
        xorl      %r14d, %r14d                                 #33.13
        movq      %rdx, %r13                                   #34.26
..B1.5:                         # Preds ..B1.5 ..B1.4
                                # Execution count [1.02e+03]
        vbroadcastss (%r13), %zmm16                            #34.26
        incb      %r15b
        vfmadd213ps 64(%rsp,%r14), %zmm15, %zmm16
        vfmadd231ps 4(%r13){1to16}, %zmm14, %zmm16
        vfmadd231ps 8(%r13){1to16}, %zmm13, %zmm16
        vfmadd231ps 12(%r13){1to16}, %zmm12, %zmm16
        vfmadd231ps 16(%r13){1to16}, %zmm11, %zmm16
        vfmadd231ps 20(%r13){1to16}, %zmm10, %zmm16
        vfmadd231ps 24(%r13){1to16}, %zmm9, %zmm16
        vfmadd231ps 28(%r13){1to16}, %zmm8, %zmm16            #34.26
        vfmadd231ps 32(%r13){1to16}, %zmm7, %zmm16            #34.26
        vfmadd231ps 36(%r13){1to16}, %zmm6, %zmm16            #34.26
        vfmadd231ps 40(%r13){1to16}, %zmm5, %zmm16            #34.26
        vfmadd231ps 44(%r13){1to16}, %zmm4, %zmm16            #34.26
        vfmadd231ps 48(%r13){1to16}, %zmm3, %zmm16            #34.26
        vfmadd231ps 52(%r13){1to16}, %zmm2, %zmm16            #34.26
        vfmadd231ps 56(%r13){1to16}, %zmm1, %zmm16            #34.26
        vfmadd231ps 60(%r13){1to16}, %zmm0, %zmm16            #34.26
        addq      $256, %r13                                   #33.13
        vmovups   %zmm16, 64(%rsp,%r14)                        #34.17
 [...]
```

- Even higher density of fused multiply-adds
- Broadcast operation embedded into the arithmetic operation!
- Better absorption of load latency (B's have been loaded much earlier)

Assembly code

MatMatMultiplyBlockHelper.s

DenseAlgebra/GEMM_Test_1_0_avx512

```
[...]
        vmovups   192(%rax,%rdx), %zmm3                         #14.41
        xorl      %r8d, %r8d                                    #10.5
        vmovups   128(%rax,%rdx), %zmm2                         #14.41
        vmovups   64(%rax,%rdx), %zmm1                          #14.41
        vmovups   (%rax,%rdx), %zmm0                            #14.41
                                # LOE rax rdx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 cl zmm0
zmm1 zmm2 zmm3
..B1.3:                         # Preds ..B1.3 ..B1.2
                                # Execution count [4.10e+03]
        vbroadcastss (%r9,%r10,4), %zmm4                        #13.40
        incq      %r10                                          #10.5
        vfmadd231ps (%r8,%rsi), %zmm4, %zmm0                    #15.18
        vfmadd231ps 64(%r8,%rsi), %zmm4, %zmm1                  #15.18
        vfmadd231ps 128(%r8,%rsi), %zmm4, %zmm2                 #15.18
        vfmadd231ps 192(%r8,%rsi), %zmm4, %zmm3                 #15.18
        addq      $256, %r8                                     #10.5
        cmpq      $64, %r10                                     #10.5
        jb        ..B1.3        # Prob 98%                      #10.5
                                # LOE rax rdx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 cl zmm0
zmm1 zmm2 zmm3
..B1.4:                         # Preds ..B1.3
                                # Execution count [6.40e+01]
        incb      %cl                                           #9.5
        vmovups   %zmm3, 192(%rax,%rdx)                         #16.30
        vmovups   %zmm2, 128(%rax,%rdx)                         #16.30
        vmovups   %zmm1, 64(%rax,%rdx)                          #16.30
        vmovups   %zmm0, (%rax,%rdx)                            #16.30
 [...]
```

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

DenseAlgebra/GEMM_Test_1_2_avx512

```
void MatMatMultiplyBlockHelper(const float (&bA)[BLOCK_SIZE][BLOCK_SIZE],
    const float (&bB)[BLOCK_SIZE][BLOCK_SIZE], float (&bC)[BLOCK_SIZE][BLOCK_SIZE])
{
    static constexpr int nW = 16; // Width of a SIMD vector
    static constexpr int nB = BLOCK_SIZE/nW;
    using const_blocked_matrix_t = const float (&) [nB][nW][nB][nW];
    using blocked_matrix_t = float (&) [nB][nW][nB][nW];
    auto bbA = reinterpret_cast<const_blocked_matrix_t>(bA[0][0]);
    auto bbB = reinterpret_cast<const_blocked_matrix_t>(bB[0][0]);
    auto bbC = reinterpret_cast<blocked_matrix_t>(bC[0][0]);

    for (int bi = 0; bi < nB; bi++) for (int bj = 0; bj < nB; bj++)
    {
        __m512 vC[nW];
        for (int ii = 0; ii < nW; ii++)
            vC[ii] =

        for (int bk
            __m512 \
            for (int
                vB[k

            for (int
                vC[i

        for (int ii
            _mm512_s
    }
}
```

1.35x the runtime of MKL code!

Execution:

```
Running candidate kernel for correctness test ... [Elapsed time : 36.8076ms]
Running reference kernel for correctness test ... [Elapsed time : 40.5675ms]
Discrepancy between two methods : 0.000156403
Running kernel for performance run # 1 ... [Elapsed time : 19.7365ms]
Running kernel for performance run # 2 ... [Elapsed time : 17.6981ms]
Running kernel for performance run # 3 ... [Elapsed time : 16.658ms]
Running kernel for performance run # 4 ... [Elapsed time : 16.4186ms]
Running kernel for performance run # 5 ... [Elapsed time : 17.454ms]
Running kernel for performance run # 6 ... [Elapsed time : 17.6172ms]
Running kernel for performance run # 7 ... [Elapsed time : 16.8232ms]
Running kernel for performance run # 8 ... [Elapsed time : 17.4193ms]
[...]
```