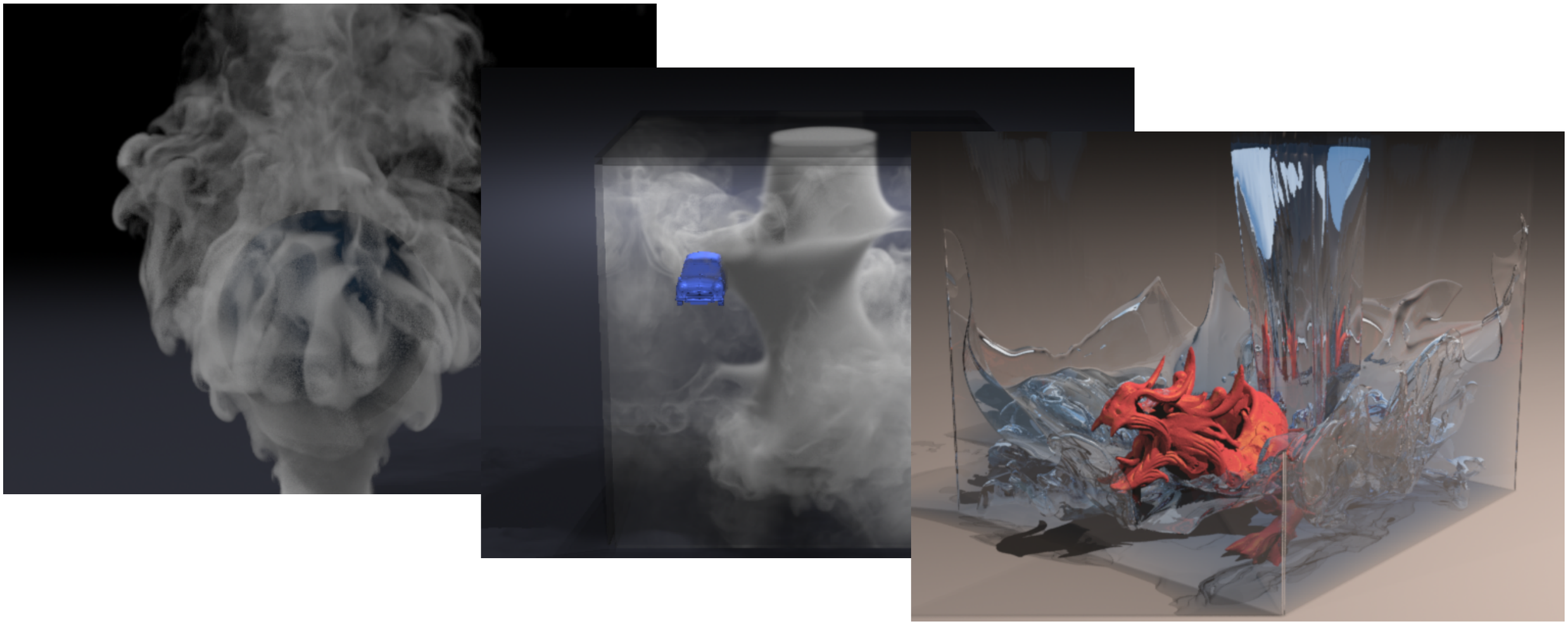


Dense Matrix Computations
Optimizing GEMM Operations in OpenMP
(Part#4 - Assembly-Level Optimizations)

Progress so far

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

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

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

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

The next (and last) step

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

Recommendation:

- Try to isolate the code "hotspot" that has the most dramatic impact on runtime performance (the more you can localize it, the better).

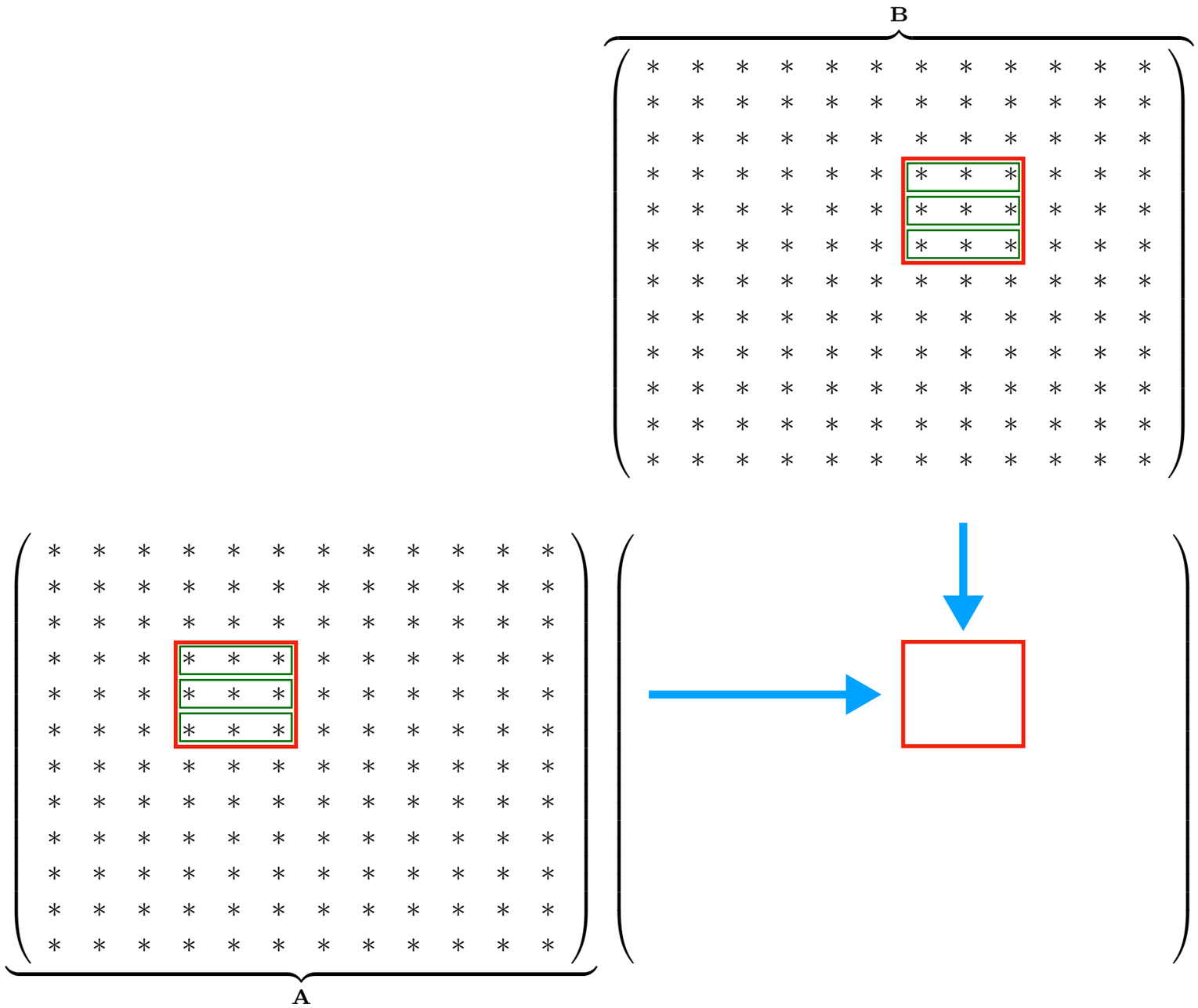
- Not a simple recipe for doing that (although profiling tools help …) but if you have a hunch, you can try to experimentally validate it

DenseAlgebra/GEMM_Test_0_10

Matrix Multiplication (MatMatMultiply.cpp)

```
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




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

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

DenseAlgebra/GEMM_Test_0_10

Matrix Multiplication (MatMatMultiply.cpp)

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

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

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

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

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

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

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

DenseAlgebra/GEMM_Test_0_10a

Matrix Multiplication (MatMatMultiply.cpp)

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

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

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

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

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

DenseAlgebra/GEMM_Test_0_10a

Matrix Multiplication (MatMatMultiply.cpp)

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

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

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

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

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

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

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

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

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

Matrix Multiplication (MatMatMultiply.cpp)

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

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

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

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

            MatMatMultiplyBlockHelper(localA, localB, localC);
        }

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

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

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#pragma once

#include "Parameters.h"

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

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

Costly inner-matrix multiply factored into separate .cpp file

```
#include "MatMatMultiplyBlockHelper.h"

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

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#include "MatMatMultiplyBlockHelper.h"

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

What happened …?

Execution:

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

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#include "MatMatMultiplyBlockHelper.h"

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

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];
    }
}
```

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
```

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];
    }
}
```

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
```

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
        vfmadd231ps 192(%r8,%rsi), %zmm7, %zmm3                 #10.13
        vmovups   %zmm3, 192(%rax,%rdx)                         #10.13
        cmpq      $64, %r10                                     #7.9
        jb        ..B1.3        # Prob 98%                      #7.9
```

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

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
[...]
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
```

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

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
# 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
```

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]

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];
    }
}
```

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!!)

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
```

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                                          #7.9
        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];
    }
}
```

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
        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)                                  #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
```

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

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#include "MatMatMultiplyBlockHelper.h"

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

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
        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)

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#include "MatMatMultiplyBlockHelper.h"

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

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

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

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

DenseAlgebra/GEMM_Test_0_11

Inner multiplication (MatMatMultiplyBlockHelper.cpp)

```
#include "MatMatMultiplyBlockHelper.h"

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

1.8x the runtime of MKL code!

Execution:

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