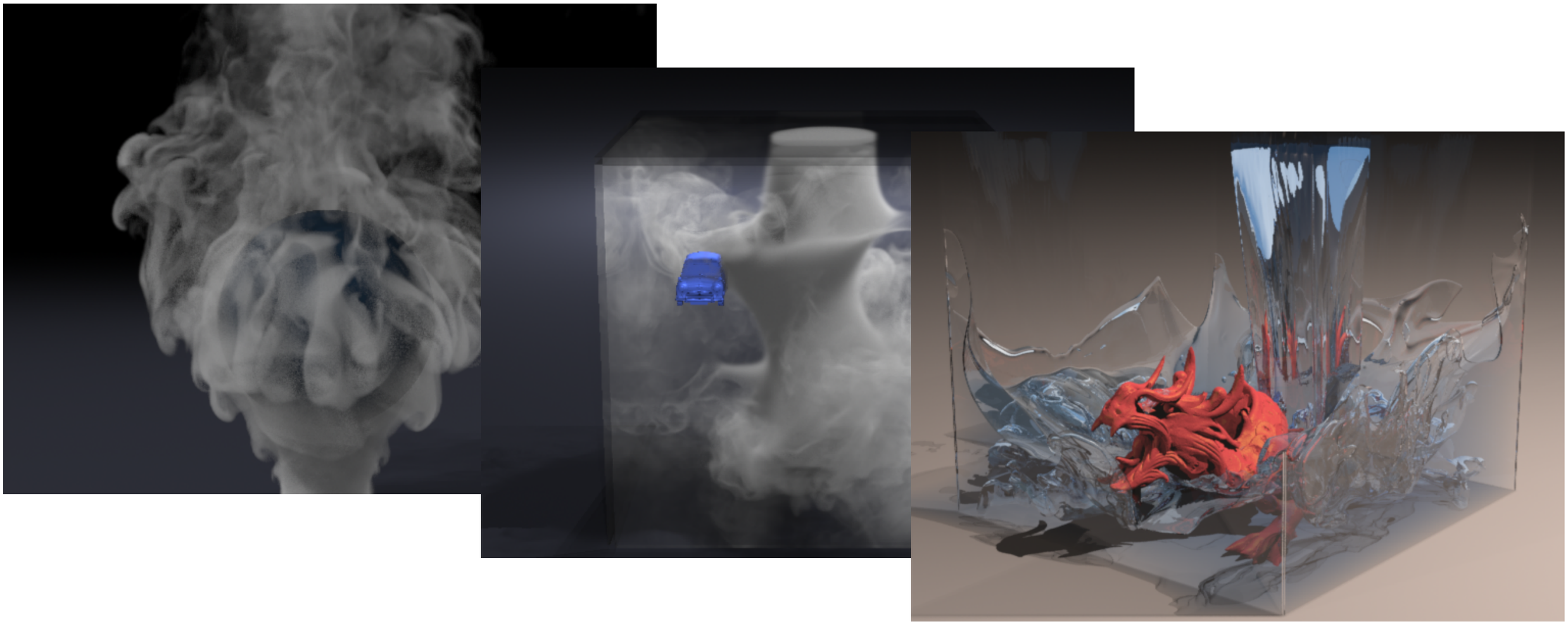

Dense Matrix Computations

Introduction to GEMM (Matrix-Matrix Multiply) Operations and Associated Optimizations

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_0

New test directory : Look at test GEMM_Test_0_[012]

```
#include "MatMatMultiply.h"
#include "Timer.h"
#include "Utilities.h"
#include <iostream>
#include <iomanip>

int main(int argc, char *argv[])
{
    float *Araw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Braw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Craw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));

    using matrix_t = float (&) [MATRIX_SIZE][MATRIX_SIZE];
    matrix_t A = reinterpret_cast<matrix_t>(*Araw);
    matrix_t B = reinterpret_cast<matrix_t>(*Braw);
    matrix_t C = reinterpret_cast<matrix_t>(*Craw);

    InitializeMatrices(A, B);
    Timer timer;

    for(int test = 1; test <= 10; test++){
        std::cout << "Running test iteration " << std::setw(2) << test << " ";
        timer.Start();
        MatMatMultiply(A, B, C);
        timer.Stop("Elapsed time : ");
    }

    return 0;
}
```

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_0

Allocates enough memory to fit a square matrix
ensuring *that the allocated memory starts at a cache line*
(i.e. at a byte address multiple of 64)

```
#include "MatMatMultiply.h"
#include "Timer.h"
#include "Utilities.h"
#include <iostream>
#include <iomanip>

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

    InitializeMatrices(A, B);
    Timer timer;

    for(int test = 1; test <= 10; test++){
        std::cout << "Running test iteration " << std::setw(2) << test << " ";
        timer.Start();
        MatMatMultiply(A, B, C);
        timer.Stop("Elapsed time : ");
    }

    return 0;
}
```

Allocate/Initialize (Utilities.cpp)

DenseAlgebra/GEMM_Test_0_0

```
#include "Utilities.h"

#include <memory>
#include <new>
#include <random>

void* AlignedAllocate(const std::size_t size, const std::size_t alignment)
{
    std::size_t capacity = size + alignment - 1;
    void *ptr = new char[capacity];
    auto result = std::align(alignment, size, ptr, capacity);
    if (result == nullptr) throw std::bad_alloc();
    if (capacity < size) throw std::bad_alloc();
    return ptr;
}

void InitializeMatrices(float (&A)[MATRIX_SIZE][MATRIX_SIZE],float (&B)[MATRIX_SIZE][MATRIX_SIZE])
{
    std::random_device rd; std::mt19937 gen(rd());
    std::uniform_real_distribution<float> uniform_dist(-1., 1.);
    for (int i = 0; i < MATRIX_SIZE; i++)
        for (int j = 0; j < MATRIX_SIZE; j++) {
            A[i][j] = uniform_dist(gen);
            B[i][j] = uniform_dist(gen);
        }
}
```

Allocate/Initialize (Utilities.cpp)

DenseAlgebra/GEMM_Test_0_0

```
#include "Utilities.h"

#include <memory>
#include <new>
#include <random>

void* AlignedAllocate(const std::size_t size, const std::size_t alignment)
{
    std::size_t capacity = size + alignment - 1;
    void *ptr = new char[capacity];
    auto result = std::align(alignment, size, ptr, capacity);
    if (result == nullptr) throw std::bad_alloc();
    if (capacity < size) throw std::bad_alloc();
    return ptr;
}

void InitializeMatrices(float (&A)[MATRI                                    X_SIZE])
{
    std::random_device rd; std::mt19937
    std::uniform_real_distribution<float
    for (int i = 0; i < MATRIX_SIZE; i++
        for (int j = 0; j < MATRIX_SIZE;
            A[i][j] = uniform_dist(gen);
            B[i][j] = uniform_dist(gen);
        }
}
```

We allocate enough space so there's enough to "trim" the beginning to make it aligned (if you needed to explicitly delete the memory, you will need to also keep a pointer to the originally allocated memory)

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_0

```
#include "MatMatMultiply.h"
#include "Timer.h"
#include "Utilities.h"
#include <iostream>
#include <iomanip>

int main(int argc, char *argv[])
{
    float *Araw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Braw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Craw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));

    using matrix_t = float (&) [MATRIX_SIZE][MATRIX_SIZE];
    matrix_t A = reinterpret_cast<matrix_t>(*Araw);
    matrix_t B = reinterpret_cast<matrix_t>(*Braw);
    matrix_t C = reinterpret_cast<matrix_t>(*Craw);

    InitializeMatrices(A, B);
    Timer timer;

    for(int test = 1; test <= 10; test++){
        std::cout << "Running test iteration
        timer.Start();
        MatMatMultiply(A, B, C);
        timer.Stop("Elapsed time : ");
    }

    return 0;
}
```

Recast the allocated memory to a "matrix" that can be indexed just like an array (e.g. A[i][j] contains element (i,j) of the array)

Note: *This is effectively a Row Major matrix*

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_0

```
#include "MatMatMultiply.h"
#include "Timer.h"
#include "Utilities.h"
#include <iostream>
#include <iomanip>

int main(int argc, char *argv[])
{
    float *Araw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Braw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Craw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));

    using matrix_t = float (&) [MATRIX_SIZE][MATRIX_SIZE];
    matrix_t A = reinterpret_cast<matrix_t>(*Araw);
    matrix_t B = reinterpret_cast<matrix_t>(*Braw);
    matrix_t C = reinterpret_cast<matrix_t>(*Craw);

    InitializeMatrices(A, B);
    Timer timer;

    for(int test = 1; test <= 10; test++){
        std::cout << "Running test iteration " << std::setw(2) << test << " ";
        timer.Start();
        MatMatMultiply(A, B, C);
        timer.Stop("Elapsed time : ");
    }

    return 0;
}
```

*Fill matrices **A & B** with random entries*

Allocate/Initialize (Utilities.cpp)

DenseAlgebra/GEMM_Test_0_0

```
#include "Utilities.h"

#include <memory>
#include <new>
#include <random>

void* AlignedAllocate(const std::size_t size, const std::size_t alignment)
{
    std::size_t capacity = size + alignment - 1;
    void *ptr = new char[capacity];
    auto result = std::align(alignment, size, ptr, capacity);
    if (result == nullptr) throw std::bad_alloc();
    if (capacity < size) throw std::bad_alloc();
    return ptr;
}

void InitializeMatrices(float (&A)[MATRIX_SIZE][MATRIX_SIZE],float (&B)[MATRIX_SIZE][MATRIX_SIZE])
{
    std::random_device rd; std::mt19937 gen(rd());
    std::uniform_real_distribution<float> uniform_dist(-1., 1.);
    for (int i = 0; i < MATRIX_SIZE; i++)
        for (int j = 0; j < MATRIX_SIZE; j++) {
            A[i][j] = uniform_dist(gen);
            B[i][j] = uniform_dist(gen);
        }
}
```

Fill each matrix with random entries between [-1, +1]

Main routine (main.cpp)

DenseAlgebra/GEMM_Test_0_0

```
#include "MatMatMultiply.h"
#include "Timer.h"
#include "Utilities.h"
#include <iostream>
#include <iomanip>

int main(int argc, char *argv[])
{
    float *Araw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Braw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));
    float *Craw = static_cast<float*>(AlignedAllocate(MATRIX_SIZE*MATRIX_SIZE*sizeof(float),64));

    using matrix_t = float (&) [MATRIX_SIZE][MATRIX_SIZE];
    matrix_t A = reinterpret_cast<matrix_t>(*Araw);
    matrix_t B = reinterpret_cast<matrix_t>(*Braw);
    matrix_t C = reinterpret_cast<matrix_t>(*Craw);

    InitializeMatrices(A, B);
    Timer timer;

    for(int test = 1; test <= 10; test++){
        std::cout << "Running test iteration " << std::setw(2) << test << " ";
        timer.Start();
        MatMatMultiply(A, B, C);
        timer.Stop("Elapsed time : ");
    }

    return 0;
}
```

Run & time the matrix-matrix multiplication operation ***C = A*B***

Kernel parameters (Parameters.h)

DenseAlgebra/GEMM_Test_0_0

```
#pragma once

#define MATRIX_SIZE 1024
```

*For simplicity, assume the size of the matrix is known at compile time, and all matrices involved are **square** with the same dimension as **MATRIX_SIZE***

GEMM routine (MatMatMultiply.h)

DenseAlgebra/GEMM_Test_0_0

```
#pragma once

#include "Parameters.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE]);
```

*This will essentially be the interface to our hand-implemented equivalent of the BLAS **GEMM** routine (general purpose Matrix-Matrix multiply)*

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_0

```
#include "MatMatMultiply.h"

void MatMatMultiply(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 i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++) {
        C[i][j] = 0.;
        for (int k = 0; k < MATRIX_SIZE; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
}
```

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_0

```
#include "MatMatMultiply.h"

void MatMatMultiply(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 i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++) {
        C[i][j] = 0.;
        for (int k = 0; k < MATRIX_SIZE; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
}
```

As we saw in theory, the triple for-loop incurs O(N^3) complexity

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_0

At matrix size = 1024

```
#include "MatMatMultiply.h"

void MatMatMultiply(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 i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++) {
        C[i][j] = 0.;
        for (int k = 0; k < MATRIX_SIZE; k
            C[i][j] += A[i][k] * B[k][j];
    }
}
```

Execution:

```
Running test iteration  1 [Elapsed time : 275.052ms]
Running test iteration  2 [Elapsed time : 245.782ms]
Running test iteration  3 [Elapsed time : 244.407ms]
Running test iteration  4 [Elapsed time : 245.818ms]
Running test iteration  5 [Elapsed time : 244.987ms]
Running test iteration  6 [Elapsed time : 244.948ms]
Running test iteration  7 [Elapsed time : 245.638ms]
Running test iteration  8 [Elapsed time : 245.293ms]
Running test iteration  9 [Elapsed time : 245.689ms]
Running test iteration 10 [Elapsed time : 245.317ms]
```

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_0

At matrix size = 512

```
#include "MatMatMultiply.h"

void MatMatMultiply(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 i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++) {
        C[i][j] = 0.;
        for (int k = 0; k < MATRIX_SIZE; k
            C[i][j] += A[i][k] * B[k][j];
    }
}
```

Execution:

```
Running test iteration  1 [Elapsed time : 30.0578ms]
Running test iteration  2 [Elapsed time : 13.7184ms]
Running test iteration  3 [Elapsed time : 13.3553ms]
Running test iteration  4 [Elapsed time : 13.4283ms]
Running test iteration  5 [Elapsed time : 13.3111ms]
Running test iteration  6 [Elapsed time : 13.4193ms]
Running test iteration  7 [Elapsed time : 13.1227ms]
Running test iteration  8 [Elapsed time : 13.5121ms]
Running test iteration  9 [Elapsed time : 13.248ms]
Running test iteration 10 [Elapsed time : 12.7863ms]
```

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_0

At matrix size = 2048

```
#include "MatMatMultiply.h"

void MatMatMultiply(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 i = 0; i < MATRIX_SIZE; i++)
    for (int j = 0; j < MATRIX_SIZE; j++) {
        C[i][j] = 0.;
        for (int k = 0; k < MATRIX_SIZE; k
            C[i][j] += A[i][k] * B[k][j];
    }
}
```

Execution:

```
Running test iteration  1 [Elapsed time : 1940.06ms]
Running test iteration  2 [Elapsed time : 1899.59ms]
Running test iteration  3 [Elapsed time : 1895.68ms]
Running test iteration  4 [Elapsed time : 1898.08ms]
Running test iteration  5 [Elapsed time : 1897.78ms]
Running test iteration  6 [Elapsed time : 1898.02ms]
Running test iteration  7 [Elapsed time : 1899.11ms]
Running test iteration  8 [Elapsed time : 1897.91ms]
Running test iteration  9 [Elapsed time : 1898.76ms]
Running test iteration 10 [Elapsed time : 1899.64ms]
```

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_1

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

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

*We have replaced our hand-implemented code with a call to the BLAS **GEMM** routine*

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_1

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

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

At matrix size = 1024

Execution:

```
Running test iteration  1 [Elapsed time : 42.4088ms]
Running test iteration  2 [Elapsed time : 3.33403ms]
Running test iteration  3 [Elapsed time : 2.29802ms]
Running test iteration  4 [Elapsed time : 2.22505ms]
Running test iteration  5 [Elapsed time : 2.21731ms]
Running test iteration  6 [Elapsed time : 1.96854ms]
Running test iteration  7 [Elapsed time : 1.87623ms]
Running test iteration  8 [Elapsed time : 1.91837ms]
Running test iteration  9 [Elapsed time : 1.91348ms]
Running test iteration 10 [Elapsed time : 1.90199ms]
```

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_1

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

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

At matrix size = 2048

Execution:

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

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_1

```
#include "MatMatMultiply.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
    static constexpr int NBLOCKS = MATRIX_SIZE / BLOCK_SIZE;

    using blocked_matrix_t = float (&) [NBLOCKS][BLOCK_SIZE][NBLOCKS][BLOCK_SIZE];
    using const_blocked_matrix_t = const float (&) [NBLOCKS][BLOCK_SIZE][NBLOCKS][BLOCK_SIZE];
    auto blockA = reinterpret_cast<const_blocked_matrix_t>(A[0][0]);
    auto blockB = reinterpret_cast<const_blocked_matrix_t>(B[0][0]);
    auto blockC = reinterpret_cast<blocked_matrix_t>(C[0][0]);
```

Adjusting our implementation to a "blocked" concept of matrix-matrix multiply

```
#pragma omp parallel for
    for (int i = 0; i < NBLOCKS; i++)
    for (int j = 0; j < NBLOCKS; j++)
        C[i][j] = 0.;

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

#pragma omp parallel for
            for (int i = 0; i < BLOCK_SIZE; i++)
            for (int j = 0; j < BLOCK_SIZE; j++)
            for (int k = 0; k < BLOCK_SIZE; k++)
                blockC[bi][i][bj][j] += blockA[bi][i][bk][k] * blockB[bk][k][bj][j];

        }
}
```

GEMM routine (MatMatMultiply.cpp)

DenseAlgebra/GEMM_Test_0_1

```
#include "MatMatMultiply.h"

void MatMatMultiply(const float (&A)[MATRIX_SIZE][MATRIX_SIZE],
    const float (&B)[MATRIX_SIZE][MATRIX_SIZE], float (&C)[MATRIX_SIZE][MATRIX_SIZE])
{
    static constexpr int NBLOCKS = MATRIX_SIZE / BLOCK_SIZE;

    using blocked_matrix_t = float (&) [NBLOCKS][BLOCK_SIZE][NBLOCKS][BLOCK_SIZE];
    using const_blocked_matrix_t = const float (&) [NBLOCKS][BLOCK_SIZE][NBLOCKS][BLOCK_SIZE];
    auto blockA = reinterpret_cast<const_blocked_matrix_t>(A[0][0]);
    auto blockB = reinterpret_cast<const_blocked_matrix_t>(B[0][0]);
    auto blockC = reinterpret_cast<blocked_matrix_t>(C[0][0]);

#pragma omp parallel for
    for (int i = 0; i < NBLOCKS; i++)
    for (int j = 0; j < NBLOCKS; j++)
        C[i][j] = 0.;

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

#pragma omp parallel for
            for (int i = 0; i < BLOCK_SIZE
            for (int j = 0; j < BLOCK_SIZE
            for (int k = 0; k < BLOCK_SIZE
                blockC[bi][i][bj][j] += bl

        }
}
```

At matrix size = 1024

Execution:

```
Running test iteration  1 [Elapsed time : 171.81ms]
Running test iteration  2 [Elapsed time : 134.102ms]
Running test iteration  3 [Elapsed time : 133.837ms]
Running test iteration  4 [Elapsed time : 134.035ms]
Running test iteration  5 [Elapsed time : 134.137ms]
Running test iteration  6 [Elapsed time : 139.447ms]
Running test iteration  7 [Elapsed time : 133.784ms]
Running test iteration  8 [Elapsed time : 134.448ms]
Running test iteration  9 [Elapsed time : 134.428ms]
Running test iteration 10 [Elapsed time : 164.302ms]
```