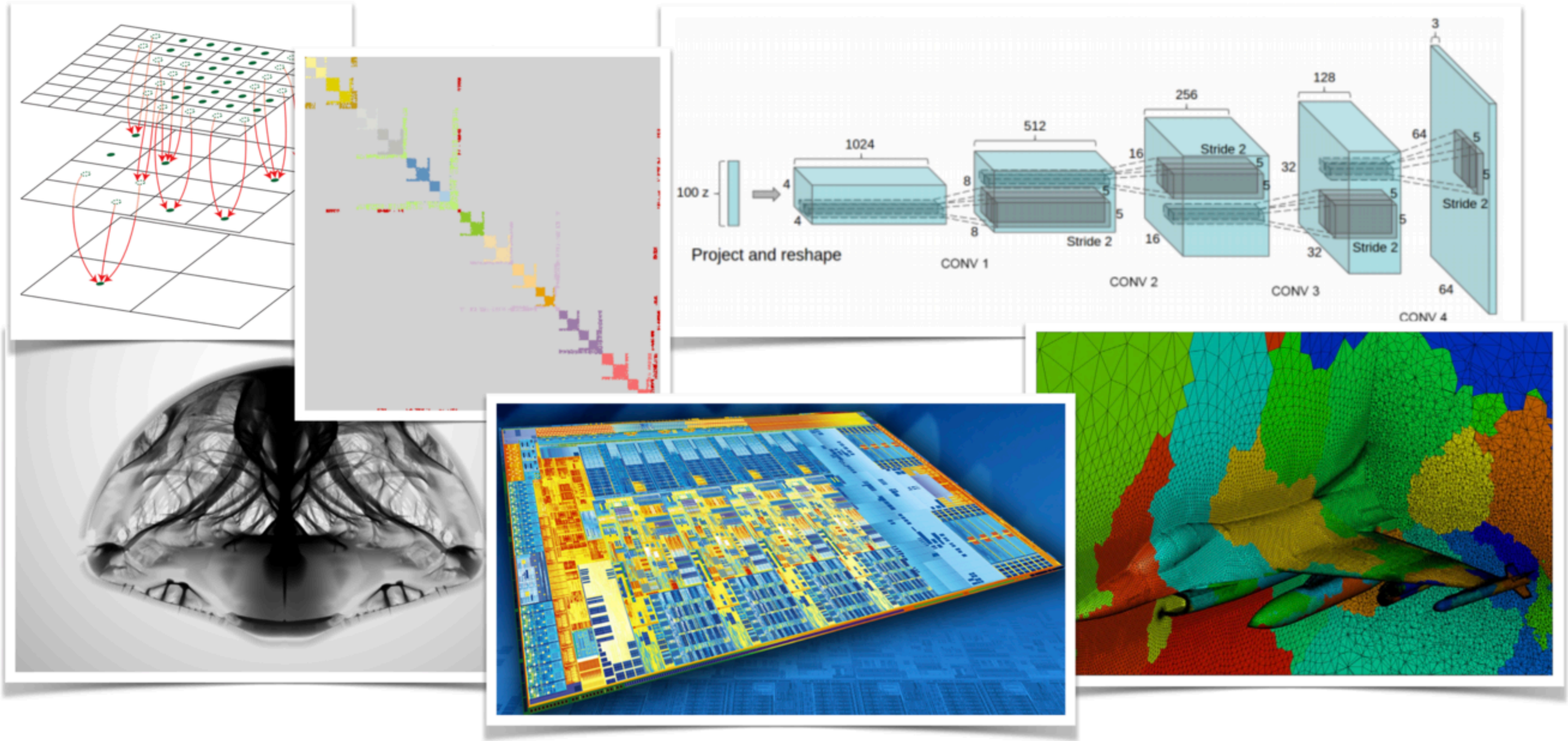

Lecture 9: More on Sparse Matrices, their associated operations, and performance. Symmetry and traversal considerations. A teaser of Intel MKL

Tuesday February 21st 2023

Logistics

- Watch out for programming assignment #2 later this week
- Will be due 1 week later
- Next week: you will be provided a practice midterm, and we will also do an in-class review ahead of your exam.

Today’s lecture

- More on the use of sparse matrices and the CSR (or associated) format(s). Focus on matrix-vector multiply.

- Performance considerations and storage footprint.

- An introduction (really, a teaser) to the Intel MKL (Math Kernel Library), and its use in sparse matrix operations.

Sparse Matrix Representations

Recap

Compressed Sparse Row *(CSR)*

$$\mathbf{A} = \begin{pmatrix} 10 & 11 & & 12 \\ & 13 & 14 & \\ 15 & & 16 & 17 \\ & 18 & & 19 \end{pmatrix}$$

the number of elements of offset array gives the numRows+1

kth entry of offset array indicates the starting position of a row in the value/col array

```
int row[]      = { 0, 0, 0, 1, 1, 2, 2, 2, 3, 3}
int offsets[] = { 0, 3, 5, 8, 10}
int col[]      = { 0, 1, 3, 1, 2, 0, 2, 3, 1, 3}
float value[] = {10,11,12,13,14,15,16,17,18,19}
```

CSR Matrix structure (CSRMatrix.h)

LaplaceSolver_1_0

Recap

```
#pragma once

#include <memory>

struct CSRMatrix
{
    int mSize;
    std::unique_ptr<int> mRowOffsets;
    std::unique_ptr<int> mColumnIndices;
    std::unique_ptr<float> mValues;

    int* GetRowOffsets() { return mRowOffsets.get(); }
    int* GetColumnIndices() { return mColumnIndices.get(); }
    float* GetValues() { return mValues.get(); }
};
```

Smart-pointer wrappers for ***rowOffsets, columnIndices, and Values*** *(just treat as arrays)*

Accessor functions (use these after initial allocation)

Entire interface is optimized for the ***recurring, performance-sensitive*** *operation of matrix-vector multiplication.*
(For example: No functionality to access/insert any individual entry!)

CSR helper (CSRMatrixHelper.h)

LaplaceSolver_1_0

This helper assists us in constructing the matrix in a more intuitive way (i.e. by accessing/setting individual entries)

Recap

```
#include "CSRMatrix.h"

struct CSRMatrixHelper
{
    std::vector<std::map<int,float> > mSparseRows;

    CSRMatrixHelper(const int size) { mSparseRows.resize(size); }

    float& operator() (const int i, const int j)
    {
        if (i < 0 || i >= mSparseRows.size() || j < 0 || j >= mSparseRows.size())
            throw std::logic_error("Matrix index out of bounds");
        return mSparseRows[i].insert( {j, 0.} ).first->second;
    }

    CSRMatrix ConvertToCSRMatrix()
    {
       [ ... omitted ...]
    }
};
```

CSR helper (CSRMatrixHelper.h)

LaplaceSolver_1_0

Recap

```
#include "CSRMatrix.h"

struct CSRMatrixHelper
{
    std::vector<std::map<int,float> > mSparseRows;

    CSRMatrixHelper(const int size) { mSparseRows.resize(size); }

    float& operator() (const int i, const int j)
    {
        if (i < 0 || i >= mSparseRows.size() || j < 0 || j >= mSparseRows.size())
            throw std::logic_error("Matrix index out of bounds");
        return mSparseRows[i].insert( {j, 0.} ).first->second;
    }

    CSRMatrix ConvertToCSRMatrix
    {
      [ ... omitted ...]
    }
};
```

"Sparse rows" stored as a list of hashtables!
(would be very slow to use in multiplication operations)

Initialization requires just the size to be specified

CSR helper (CSRMatrixHelper.h)

LaplaceSolver_1_0

Convenient accessor that uses matrix indices to access (or insert) a matrix entry!
This can be very slow (but convenient!)

Recap

```
#include "CSRMatrix.h"

struct CSRMatrixHelper
{
    std::vector<std::map<int,float> > mSparseRows;

    CSRMatrixHelper(const int size) { mSparseRows.resize(size); }

    float& operator() (const int i, const int j)
    {
        if (i < 0 || i >= mSparseRows.size() || j < 0 || j >= mSparseRows.size())
            throw std::logic_error("Matrix index out of bounds");
        return mSparseRows[i].insert( {j, 0.} ).first->second;
    }

    CSRMatrix ConvertToCSRMatrix()
    {
       [ ... omitted ...]
    }
};
```

CSR helper (CSRMatrixHelper.h)

LaplaceSolver_1_0

Recap

```
#include "CSRMatrix.h"

struct CSRMatrixHelper
{
    std::vector<std::map<int,float> > mSparseRows;

    CSRMatrixHelper(const int size) { mSparseRows.resize(size); }

    float& operator() (const int i, const int j)
    {
        if (i < 0 || i >= mSparseRows.size() || j < 0 || j >= mSparseRows.size())
            throw std::logic_error("Matrix index out of bounds");
        return mSparseRows[i].insert( {j, 0.} ).first->second;
    }

    CSRMatrix ConvertToCSRMatrix()
    {
       [ ... omitted ...]
    }
};
```

Once we've constructed the matrix (entry-by-entry) we can convert it to the performance-optimized format

CSR helper (CSRMatrixHelper.h)

LaplaceSolver_1_0

Recap

```
struct CSRMatrixHelper
{
    std::vector<std::map<int,float> > mSparseRows;
    [ ... omitted ...]
    CSRMatrix ConvertToCSRMatrix()
    {
        int N = mSparseRows.size(); // Size of matrix
        int NNZ = 0; // Number of non-zero entries
        for (int i = 0; i < N; i++) NNZ += mSparseRows[i].size();

        CSRMatrix matrix { N }; // Initialize just matrix.mSize
        matrix.mRowOffsets.reset(new int [N + 1]); // Need a sentinel value in the end
        matrix.mColumnIndices.reset(new int [NNZ]);
        matrix.mValues.reset(new float [NNZ]);

        auto rowOffsets = matrix.GetRowOffsets();
        auto columnIndices = matrix.GetColumnIndices();
        auto values = matrix.GetValues();

        rowOffsets[0] = 0;
        for (int i = 0, k = 0; i < N; i++) {
            rowOffsets[i + 1] = rowOffsets[i] + mSparseRows[i].size(); // Mark where this row ends
            for (auto it = mSparseRows[i].begin(); it != mSparseRows[i].end(); it++) {
                columnIndices[k] = it->first;
                values[k] = it->second;
                k++;}}
       return matrix;
    }
};
```

CSR helper (CSRMatrixHelper.h)

LaplaceSolver_1_0

Recap

```
struct CSRMatrixHelper
{
    std::vector<std::map<int,float> > mSparseRows;
    [ ... omitted ...]
    CSRMatrix ConvertToCSRMatrix()
    {
        int N = mSparseRows.size(); // Size of matrix
        int NNZ = 0; // Number of non-zero entries
        for (int i = 0; i < N; i++) NNZ += mSparseRows[i].size();

        CSRMatrix matrix { N }; // Initialize just matrix.mSize
        matrix.mRowOffsets.reset(new int [N + 1]); // Need a sentinel value in the end
        matrix.mColumnIndices.reset(new int [NNZ]);
        matrix.mValues.reset(new float [NNZ]);

        auto rowOffsets = matrix.GetRowOffsets();
        auto columnIndices = matrix.GetColumnIndices();
        auto values = matrix.GetValues();

        rowOffsets[0] = 0;
        for (int i = 0, k = 0; i < N; i++) {
            rowOffsets[i + 1] = rowOffsets[i] + mSparseRows[i].size(); // Mark where this row ends
            for (auto it = mSparseRows[i].begin(); it != mSparseRows[i].end(); it++) {
                columnIndices[k] = it->first;
                values[k] = it->second;
                k++;}}
       return matrix;
    }
};
```

Count how many non-zero entries have been inserted in the matrix …

CSR helper (CSRMatrixHelper.h)

LaplaceSolver_1_0

Recap

```
struct CSRMatrixHelper
{
    std::vector<std::map<int,float> > mSparseRows;
    [ ... omitted ...]
    CSRMatrix ConvertToCSRMatrix()
    {
        int N = mSparseRows.size(); // Size (
        int NNZ = 0; // Number of non-zero en
        for (int i = 0; i < N; i++) NNZ += mSparseRows[i].size();

        CSRMatrix matrix { N }; // Initialize just matrix.mSize
        matrix.mRowOffsets.reset(new int [N + 1]); // Need a sentinel value in the end
        matrix.mColumnIndices.reset(new int [NNZ]);
        matrix.mValues.reset(new float [NNZ]);

        auto rowOffsets = matrix.GetRowOffsets();
        auto columnIndices = matrix.GetColumnIndices();
        auto values = matrix.GetValues();

        rowOffsets[0] = 0;
        for (int i = 0, k = 0; i < N; i++) {
            rowOffsets[i + 1] = rowOffsets[i] + mSparseRows[i].size(); // Mark where this row ends
            for (auto it = mSparseRows[i].begin(); it != mSparseRows[i].end(); it++) {
                columnIndices[k] = it->first;
                values[k] = it->second;
                k++;}}
       return matrix;
    }
};
```

… allocate all arrays to the appropriate size …

CSR helper (CSRMatrixHelper.h)

LaplaceSolver_1_0

Recap

```
struct CSRMatrixHelper
{
    std::vector<std::map<int,float> > mSparseRows;
    [ ... omitted ...]
    CSRMatrix ConvertToCSRMatrix()
    {
        int N = mSparseRows.size(); // Size of matrix
        int NNZ = 0; // Number of non-zero entries
        for (int i = 0; i < N; i++) NNZ += mSparseRows[i].size();

        CSRMatrix matrix { N }; // Initialize just matrix.mSize
        matrix.mRowOffsets.reset(new int [N + 1]); // Need a sentinel value in the end
        matrix.mColumnIndices.reset(new int [NNZ]);
        matrix.mValues.reset(new float [NNZ])

        auto rowOffsets = matrix.GetRowOffset
        auto columnIndices = matrix.GetColumn
        auto values = matrix.GetValues();

        rowOffsets[0] = 0;
        for (int i = 0, k = 0; i < N; i++) {
            rowOffsets[i + 1] = rowOffsets[i] + mSparseRows[i].size(); // Mark where this row ends
            for (auto it = mSparseRows[i].begin(); it != mSparseRows[i].end(); it++) {
                columnIndices[k] = it->first;
                values[k] = it->second;
                k++;}}
       return matrix;
    }
};
```

… and insert all entries to the flattened arrays, sorting them along the way, and building the "offsets" list as well

Matrix-Vector multiply (MatVecMultiply.cpp)

LaplaceSolver_1_0

Recap

```
#include "MatVecMultiply.h"

void MatVecMultiply(CSRMatrix& mat, const float *x, float *y)
{
    int N = mat.mSize;
    const auto rowOffsets = mat.GetRowOffsets();
    const auto columnIndices = mat.GetColumnIndices();
    const auto values = mat.GetValues();

#pragma omp parallel for
    for (int i = 0; i < N; i++)
    {
        y[i] = 0.;
        for (int k = rowOffsets[i]; k < rowOffsets[i+1]; k++) {
            const int j = columnIndices[k];
            y[i] += values[k] * x[j];
        }
    }
}
```

Matrix-Vector multiply (MatVecMultiply.cpp)

LaplaceSolver_1_0

Recap

```
#include "MatVecMultiply.h"

void MatVecMultiply(CSRMatrix& mat, const float *x, float *y)
{
    int N = mat.mSize;
    const auto rowOffsets = mat.GetRowOffsets();
    const auto columnIndices = mat.GetColumnIndices();
    const auto values = mat.GetValues();

#pragma omp parallel for
    for (int i = 0; i < N; i++)
    {
        y[i] = 0.;
        for (int k = rowOffsets[i]; k < rowOffsets[i+1]; k++) {
            const int j = columnIndices[k];
            y[i] += values[k] * x[j];
        }
    }
}
```

Unpack components of CSR Matrix

Matrix-Vector multiply (MatVecMultiply.cpp)

LaplaceSolver_1_0

Recap

```
#include "MatVecMultiply.h"

void MatVecMultiply(CSRMatrix& mat, const float *x, float *y)
{
    int N = mat.mSize;
    const auto rowOffsets = mat.GetRowOffsets();
    const auto columnIndices = mat.GetColumnIndices();
    const auto values = mat.GetValues();

#pragma omp parallel for
    for (int i = 0; i < N; i++)
    {
        y[i] = 0.;
        for (int k = rowOffsets[i]; k < rowOffsets[i+1]; k++) {
            const int j = columnIndices[k];
            y[i] += values[k] * x[j];
        }
    }
}
```

Build matrix-vector product row-by-row (similar to how we applied one stencil at a time, to compute a single value of Lu(i,j,k))

Matrix-Vector multiply (MatVecMultiply.cpp)

LaplaceSolver_1_0

Recap

```
#include "MatVecMultiply.h"

void MatVecMultiply(CSRMatrix& mat, const float *x, float *y)
{
    int N = mat.mSize;
    const auto rowOffsets = mat.GetRowOffsets();
    const auto columnIndices = mat.GetColumnIndices();
    const auto values = mat.GetValues();

#pragma omp parallel for
    for (int i = 0; i < N; i++)
    {
        y[i] = 0.;
        for (int k = rowOffsets[i]; k < rowOffsets[i+1]; k++) {
            const int j = columnIndices[k];
            y[i] += values[k] * x[j];
        }
    }
}
```

*The flattened indices for the elements of sparse row **i** can be found between entries **rowOffsets[i]** and **rowOffsets[i+1]-1** of arrays **columnIndices** and **values***

New Laplacian (Laplacian.h)

LaplaceSolver_1_0

New Laplacian : Build and use CSR Matrix

```
#pragma once

#include "CSRMatrix.h"
#include "Parameters.h"

CSRMatrix BuildLaplacianMatrix();

void ComputeLaplacian(CSRMatrix& laplacianMatrix,
    const float (&u)[XDIM][YDIM][ZDIM], float (&Lu)[XDIM][YDIM][ZDIM]);
```

New Laplacian (Laplacian.cpp)

LaplaceSolver_1_0

New Laplacian : Build and use CSR Matrix

```
#include "CSRMatrixHelper.h"
#include "Laplacian.h"
#include "MatVecMultiply.h"

inline int LinearIndex(const int i, const int j, const int k)
{ return ((i * YDIM) + j) * ZDIM + k; }

CSRMatrix BuildLaplacianMatrix() {
    static constexpr int matSize = XDIM * YDIM * ZDIM;
    CSRMatrixHelper matrixHelper(matSize);

    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++) {   (row, column), -6 is on the diagonal
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k) ) = -6.;
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i+1, j, k) ) = 1.;
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i-1, j, k) ) = 1.;
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j+1, k) ) = 1.;
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j-1, k) ) = 1.;
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k+1) ) = 1.;
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k-1) ) = 1.;}
    return matrixHelper.ConvertToCSRMatrix();
}

void ComputeLaplacian(CSRMatrix& laplacianMatrix,
    const float (&u)[XDIM][YDIM][ZDIM], float (&Lu)[XDIM][YDIM][ZDIM]) {
    // Treat the arrays u & Lu as flattened vectors, and apply matrix-vector multiplication
    MatVecMultiply(laplacianMatrix, &u[0][0][0], &Lu[0][0][0]);
}
```

New Laplacian (Laplacian.cpp)

LaplaceSolver_1_0

```
#include "CSRMatrixHelper.h"
#include "Laplacian.h"
#include "MatVecMultiply.h"
```

[0][1][0] -> [i][j][k] yields 512 which is the expcted location

```
inline int LinearIndex(const int i, const int j, const int k)
{ return ((i * YDIM) + j) * ZDIM + k; }

CSRMatrix BuildLaplacianMatrix() {
    static constexpr int matSize = XDIM * YDIM
    CSRMatrixHelper matrixHelper(matSize);

    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++) {
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k) ) = -6.;
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i+1, j, k) ) = 1.;
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i-1, j, k) ) = 1.;
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j+1, k) ) = 1.;
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j-1, k) ) = 1.;
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k+1) ) = 1.;
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k-1) ) = 1.;}
    return matrixHelper.ConvertToCSRMatrix();
}

void ComputeLaplacian(CSRMatrix& laplacianMatrix,
    const float (&u)[XDIM][YDIM][ZDIM], float (&Lu)[XDIM][YDIM][ZDIM]) {
    // Treat the arrays u & Lu as flattened vectors, and apply matrix-vector multiplication
    MatVecMultiply(laplacianMatrix, &u[0][0][0], &Lu[0][0][0]);
}
```

Simple remapping from a triple of grid indices (i,j,k) to the corresponding “flattened” index when stored in lexicographical order

Recap

Example : The 3D Poisson equation

$$
\begin{pmatrix}
-6 & 1 & & & 1 & & & & 1 & & & \\
1 & -6 & 1 & & & 1 & & & & \ddots & & \\
& 1 & -6 & 1 & & & \ddots & & & & \ddots & \\
& & \ddots & \ddots & \ddots & & & \ddots & & & & 1 \\
1 & & & \ddots & \ddots & \ddots & & & \ddots & & & \\
& 1 & & & \ddots & \ddots & \ddots & & & \ddots & & \\
& & \ddots & & & \ddots & \ddots & \ddots & & & 1 & \\
& & & \ddots & & & \ddots & \ddots & \ddots & & & 1 \\
1 & & & & \ddots & & & \ddots & \ddots & \ddots & & \\
& \ddots & & & & \ddots & & & 1 & -6 & 1 & \\
& & \ddots & & & & 1 & & & 1 & -6 & 1 \\
& & & 1 & & & & 1 & & & 1 & -6
\end{pmatrix}
\mathbf{x} = \mathbf{b}
$$

What about x & b?
How are they stored?

Recap

Example : The 3D Poisson equation

$$\mathbf{x} = \begin{pmatrix} \mathrm{u[0][0][0]} \\ \mathrm{u[0][0][1]} \\ \vdots \\ \mathrm{u[0][0][511]} \\ \mathrm{u[0][1][0]} \\ \mathrm{u[0][1][1]} \\ \vdots \\ \mathrm{u[0][1][511]} \\ \vdots \\ \mathrm{u[511][511][511]} \end{pmatrix}$$

$$\begin{pmatrix} \ddots & & \\ & \ddots & \\ & & 1 \\ \ddots & & \\ & 1 & \\ & & 1 \\ \ddots & & \\ -6 & 1 & \\ 1 & -6 & 1 \\ & 1 & -6 \end{pmatrix} \mathbf{x} = \mathbf{b}$$

What about x & b?
How are they stored?

New CG routine (ConjugateGradients.cpp)

LaplaceSolver_1_0

Two ways of Compute Laplacian operation that is used in the Conjugate Gradient algorithm: Stencils or Matrices

```
void ConjugateGradients(
    CSRMatrix& matrix,
    float (&x)[XDIM][YDIM][ZDIM],
    const float (&f)[XDIM][YDIM][ZDIM],
    float (&p)[XDIM][YDIM][ZDIM],
    float (&r)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM],
    const bool writeIterations)
{
    // Algorithm : Line 2
    timerLaplacian.Restart(); ComputeLaplacian(matrix, x, z); timerLaplacian.Pause();
    Saxpy(z, f, r, -1);
    float nu = Norm(r);

    // Algorithm : Line 3
    if (nu < nuMax) return;

    // Algorithm : Line 4
    Copy(r, p);
    float rho=InnerProduct(p, r);

    // Beginning of loop from Line 5
    for(int k=0;;k++)
    {
        std::cout << "Residual norm (nu) after " << k << " iterations = " << nu << std::endl;

        // Algorithm : Line 6
        timerLaplacian.Restart(); ComputeLaplacian(matrix, p, z); timerLaplacian.Pause();
        float sigma=InnerProduct(p, z);
```

New timer (Timer.h)

LaplaceSolver_0_3

LaplaceSolver_1_0

0_x are matrix free
1_x are explicit CSR matrices

```
[...]

struct Timer
{
    using clock_t = std::chrono::high_resolution_clock;
    using time_point_t = std::chrono::time_point<clock_t>;
    using elapsed_time_t = std::chrono::duration<double, std::milli>;

    time_point_t mStartTime;
    time_point_t mStopTime;
    elapsed_time_t mElapsedTime;

    void Start()
    { mElapsedTime = elapsed_time_t::zero(); mStartTime = clock_t::now(); }

    void Stop(const std::string& msg)
    {
        mStopTime = clock_t::now();
        std::chrono::duration<double, std::milli> elapsedTime = mStopTime - mStartTime;
        std::cout << "[" << msg << elapsedTime.count() << "ms]" << std::endl;
    }

    void Reset()
    { mElapsedTime = elapsed_time_t::zero(); }

    void Restart()
    { mStartTime = clock_t::now(); }

    void Pause()
    { mStopTime = clock_t::now(); mElapsedTime += mStopTime-mStartTime; }

    void Print(const std::string& msg)
    { std::cout << "[" << msg << mElapsedTime.count() << "ms]" << std::endl; }
}
```

New main routine (main.cpp)

LaplaceSolver_1_0

```
[.. .]
Timer timerLaplacian;

int main(int argc, char *argv[])
{
    using array_t = float (&) [XDIM][YDIM][ZDIM];
    float *xRaw = new float [XDIM*YDIM*ZDIM];
    [.. .]
    array_t z = reinterpret_cast<array_t>(*zRaw);

    CSRMatrix matrix;

    // Initialization
    {
        Timer timer;
        timer.Start();
        InitializeProblem(x, f);
        matrix = BuildLaplacianMatrix(); // This takes a while ...
        timer.Stop("Initialization : ");
    }

    // Call Conjugate Gradients algorithm
    timerLaplacian.Reset();
    ConjugateGradients(matrix, x, f, p, r, z, false);
    timerLaplacian.Print("Total Laplacian Time : ");

    return 0;
}
```

New main routine (main.cpp)

LaplaceSolver_1_0

```
[.. .]
Timer timerLaplacian;

int main(int argc, char *argv[])
{
    using array_t = float (&) [XDIM][YDIM][ZDIM];
    float *xRaw = new float [XDIM*YDIM*ZDIM];
    [.. .]
    array_t z = reinterpret_cast<array_t>(*zRaw);

    CSRMatrix matrix;

    // Initialization
    {
        Timer timer;
        timer.Start();
        InitializeProblem(x, f);
        matrix = BuildLaplacianMatrix(); // This takes a while ...
        timer.Stop("Initialization : ");
    }

    // Call Conjugate Gradients algorithm
    timerLaplacian.Reset();
    ConjugateGradients(matrix, x, f, p, r, z, false);
    timerLaplacian.Print("Total Laplacian Time : ");

    return 0;
}
```

The explicitly constructed matrix now gets passed as an argument to the CG algorithm

New CG routine (ConjugateGradients.cpp)

LaplaceSolver_1_0

```
void ConjugateGradients(
    CSRMatrix& matrix,
    float (&x)[XDIM][YDIM][ZDIM],
    const float (&f)[XDIM][YDIM][ZDIM],
    float (&p)[XDIM][YDIM][ZDIM],
    float (&r)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM],
    const bool writeIterations)
{
    // Algorithm : Line 2
    timerLaplacian.Restart(); ComputeLaplacian(matrix, x, z); timerLaplacian.Pause();
    Saxpy(z, f, r, -1);
    float nu = Norm(r);

    // Algorithm : Line 3
    if (nu < nuMax) return;

    // Algorithm : Line 4
    Copy(r, p);
    float rho=InnerProduct(p, r);

    // Beginning of loop from Line 5
    for(int k=0;;k++)
    {
        std::cout << "Residual norm (nu) after " << k << " iterations = " << nu << std::endl;

        // Algorithm : Line 6
        timerLaplacian.Restart(); ComputeLaplacian(matrix, p, z); timerLaplacian.Pause();
        float sigma=InnerProduct(p, z);
```

Note how we capture the execution cost of the matrix-based Laplacian matrix-vector multiply across several calls …

New CG routine (ConjugateGradients.cpp)

LaplaceSolver_1_0

```
void ConjugateGradients(
    CSRMatrix& matrix,
    float (&x)[XDIM][YDIM][ZDIM],
    const float (&f)[XDIM][YDIM][ZDIM],
    float (&p)[XDIM][YDIM][ZDIM],
    float (&r)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM],
    const bool writeIterations)
{
    // Algorithm : Line 2
    timerLaplacian.Restart(); ComputeLaplacian(matrix, x, z); timerLaplacian.Pause();
    Saxpy(z, f, r, -1);
    float nu = Norm(r);

    // Algorithm : Line 3
    if (nu < nuMax) return;

    // Algorithm : Line 4
    Copy(r, p);
    float rho=InnerProduct(p, r);

    // Beginning of loop from Line 5
    for(int k=0;;k++)
    {
        std::cout << "Residual norm (nu) after " << k << " iterations = " << nu << std::endl;

        // Algorithm : Line 6
        timerLaplacian.Restart(); ComputeLaplacian(matrix, p, z); timerLaplacian.Pause();
        float sigma=InnerProduct(p, z);
```

(On our benchmark system: ~3921ms for 257 iterations
Approximately 15.2ms per call)

New CG routine (ConjugateGradients.cpp)

LaplaceSolver_0_3

```
void ConjugateGradients(
    float (&x)[XDIM][YDIM][ZDIM],
    const float (&f)[XDIM][YDIM][ZDIM],
    float (&p)[XDIM][YDIM][ZDIM],
    float (&r)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM],
    const bool writeIterations)
{
    // Algorithm : Line 2
    timerLaplacian.Restart(); ComputeLaplacian(x, z); timerLaplacian.Pause();
    Saxpy(z, f, r, -1);
    float nu = Norm(r);

    // Algorithm : Line 3
    if (nu < nuMax) return;

    // Algorithm : Line 4
    Copy(r, p);
    float rho=InnerProduct(p, r);

    // Beginning of loop from Line 5
    for(int k=0;;k++)
    {
        std::cout << "Residual norm (nu) after " << k << " iterations = " << nu << std::endl;

        // Algorithm : Line 6
        timerLaplacian.Restart(); ComputeLaplacian(p, z); timerLaplacian.Pause();
```

(Matrix-free version: ~800ms for 257 iterations Approximately 3.1ms per call)

Benefit of not using the matrix

The Intel Math Kernel Library (MKL)

```
Level 1: Vector - Vector (for example SAXPY, scaling)
Level 2: Matrix - Vector
Level 3: Matrix - Matrix

These levels across libraries are based on complexity defined in historic
libraries such as BLAS and LAPACK
```

Getting started :

- Download from : https://software.intel.com/en-us/mkl
- Documentation : https://software.intel.com/en-us/mkl/documentation/view-all
- Compilation options https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor/
- Easiest with Intel Compiler, but supported on all platforms!

```
On CSL, icc -mkl willinlcude the library
```

```
Ax=b
ATAx = ATb, we use this to equate the two values when numUnknowns < numEquations
Conjugate gradients always works with ATA which is positive definite
```

Normal Equation

Given a matrix equation

$$\mathsf{A}\,\mathbf{x} = \mathbf{b},$$

the normal equation is that which minimizes the sum of the square differences between the left and right sides:

$$\mathsf{A}^{\mathrm{T}}\,\mathsf{A}\,\mathbf{x} = \mathsf{A}^{\mathrm{T}}\,\mathbf{b}.$$

It is called a normal equation because $\mathbf{b} - \mathsf{A}\,\mathbf{x}$ is normal to the range of A.

Here, $\mathsf{A}^{\mathrm{T}}\,\mathsf{A}$ is a normal matrix.

Normal Matrix

A square matrix A is a normal matrix if

$$[\mathsf{A}, \mathsf{A}^{\mathrm{H}}] = \mathsf{A}\,\mathsf{A}^{\mathrm{H}} - \mathsf{A}^{\mathrm{H}}\,\mathsf{A} = 0,$$

where $[a, b]$ is the commutator and A^{H} denotes the conjugate transpose. For example, the matrix

$$\begin{bmatrix} i & 0 \\ 0 & 3 - 5\,i \end{bmatrix}$$

is a normal matrix, but is *not* a Hermitian matrix.

A matrix m can be tested to see if it is normal in the Wolfram Language using NormalMatrixQ[m].

The opposite of a Normal Matrix is a Transposed matrix

Matrix-Vector multiply (MatVecMultiply.cpp)

LaplaceSolver_1_4

```
[...]

void MatVecMultiply(CSRMatrix& mat, const float *x, float *y)
{
    int N = mat.mSize; const auto rowOffsets = mat.GetRowOffsets();
    const auto columnIndices = mat.GetColumnIndices(); const auto values = mat.GetValues();

#ifdef DO_NOT_USE_MKL
#pragma omp parallel for
    for (int i = 0; i < N; i++) {
        y[i] = 0.;
        for (int k = rowOffsets[i]; k < rowOffsets[i+1]; k++) {
            const int j = columnIndices[k];
            y[i] += values[k] * x[j];
        }
    }
#else
    mkl_cspblas_scsrgemv( // (S)parse (CSR) (Ge)neral matrix (M)atrix-(V)ector product
        "N",              // Use the normal matrix, not its transpose
        &N,               // Size of the matrix
        values,           // values array (MKL denotes this as "a")
        rowOffsets,       // rowOffsets array (MKL denotes this as "ia")
        columnIndices,    // columnIndices array (MKL denotes this as "ja")
        x,                // Vector getting multiplied
        y                 // Vector where the product gets stored
    );
#endif
}
```

Matrix-Vector multiply (MatVecMultiply.cpp)

LaplaceSolver_1_4

```
[...]

void MatVecMultiply(CSRMatrix& mat, const float *x, float *y)
{
    int N = mat.mSize; const auto rowOffsets = mat.GetRowOffsets();
    const auto columnIndices = mat.GetColumnIndices(); const auto values = mat.GetValues();

#ifdef DO_NOT_USE_MKL
#pragma omp parallel for
    for (int i = 0; i < N; i++) {
        y[i] = 0.;
        for (int k = rowOffsets[i]; k < rowOffsets[i+1]; k++) {
            const int j = columnIndices[k];
            y[i] += values[k] * x[j];
        }
    }
#else
    mkl_cspblas_scsrgemv( // (S)parse (CSR) (Ge)neral matrix (M)atrix-(V)ector product
        "N",              // Use the normal matrix, not its transpose
        &N,               // Size of the matrix
        values,           // values array (MKL denotes this as "a")
        rowOffsets,       // rowOffsets array (MKL denotes this as "ia")
        columnIndices,    // columnIndices array (MKL denotes this as "ja")
        x,                // Vector getting multiplied
        y                 // Vector where the product gets stored
    );
#endif
}
```

Matrix-Vector multiply (MatVecMultiply.cpp)

LaplaceSolver_1_4

```
[...]

void MatVecMultiply(CSRMatrix& mat, const float *x, float *y)
{
    int N = mat.mSize; const auto rowOffsets = mat.GetRowOffsets();
    const auto columnIndices = mat.GetColumnIndices(); const auto values = mat.GetValues();

#ifdef DO_NOT_USE_MKL
#pragma omp parallel for
    for (int i = 0; i < N; i++) {
        y[i] = 0.;
        for (int k = rowOffsets[i]; k < rowOffsets[i+1]; k++) {
            const int j = columnIndices[k];
            y[i] += values[k] * x[j];
        }
    }
#else
    mkl_cspblas_scsrgemv( // (S)parse (CSR) (Ge)neral matrix (M)atrix-(V)ector product
        "N",              // Use the normal matrix, not its transpose
        &N,               // Size of the matrix
        values,           // values array (MKL denotes this as "a")
        rowOffsets,       // rowOffsets array (MKL denotes this as "ia")
        columnIndices,    // columnIndices array (MKL denotes this as "ja")
        x,                // Vector getting multiplied
        y                 // Vector where the product gets stored
    );
#endif
}
```

This parallelization becomes a load balancing issue based on the pattern of nnz. For example, a single row can house a large number of nnz and other processors wait for it

MKL-based vector ops (PointwiseOps.h)

LaplaceSolver_1_4

```
#pragma once

#include "Parameters.h"

// Copy array x into y
void Copy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM]);

// Scale array x by given number, add y, and write result into z
void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM], const float scale);

// Scale array x by given number, add y, and write result into y (specialization of call above)
void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM],
    const float scale);
```

MKL-based vector ops (PointwiseOps.h)

LaplaceSolver_1_4

```
#pragma once

#include "Parameters.h"

// Copy array x into y
void Copy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM]);

// Scale array x by given number, add y, and write result into z
void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM], const float scale);

// Scale array x by given number, add y, and write result into y (specialization of call above)
void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM],
    const float scale);
```

Special case, when the output vector (z) is the same as the one we add to (y)

MKL-based vector ops (PointwiseOps.cpp)

LaplaceSolver_1_4

```
[...]

void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM], const float scale)
{
    [...]
}

void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM],
    const float scale)
{
#ifdef DO_NOT_USE_MKL
    // Just for reference -- implementation without MKL
#pragma omp parallel for
    for (int i = 0; i < XDIM; i++)
    for (int j = 0; j < YDIM; j++)
    for (int k = 0; k < ZDIM; k++)
        y[i][j][k] += x[i][j][k] * scale;
#else
    cblas_saxpy(
        XDIM * YDIM * ZDIM, // Length of vectors
        scale,              // Scale factor
        &x[0][0][0],        // Input vector x, in operation y := x * scale + y
        1,                  // Use step 1 for x
        &y[0][0][0],        // Input/output vector y, in operation y := x * scale + y
        1                   // Use step 2 for y
    );
#endif
}
```

MKL-based vector ops (PointwiseOps.cpp)

LaplaceSolver_1_4

```
[...]

void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM], const float scale)
{
    [...]
}
```

Special case, when the output vector (z) is the same as the one we add to (y)

```
void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM],
    const float scale)
{
#ifdef DO_NOT_USE_MKL
    // Just for reference -- implementation without MKL
#pragma omp parallel for
    for (int i = 0; i < XDIM; i++)
    for (int j = 0; j < YDIM; j++)
    for (int k = 0; k < ZDIM; k++)
        y[i][j][k] += x[i][j][k] * scale;
#else
    cblas_saxpy(
        XDIM * YDIM * ZDIM, // Length of vectors
        scale,              // Scale factor
        &x[0][0][0],        // Input vector x, in operation y := x * scale + y
        1,                  // Use step 1 for x
        &y[0][0][0],        // Input/output vector y, in operation y := x * scale + y
        1                   // Use step 2 for y
    );
#endif
}
```

MKL-based vector ops (PointwiseOps.cpp)

LaplaceSolver_1_4

```
[...]

void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM], const float scale)
{
    [...]
}

void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM],
    const float scale)
{
#ifdef DO_NOT_USE_MKL
    // Just for reference -- implementation without MKL
#pragma omp parallel for
    for (int i = 0; i < XDIM; i++)
    for (int j = 0; j < YDIM; j++)
    for (int k = 0; k < ZDIM; k++)
        y[i][j][k] += x[i][j][k] * scale;
#else
    cblas_saxpy(
        XDIM * YDIM * ZDIM, // Length of vectors
        scale,              // Scale factor
        &x[0][0][0],        // Input vector x, in operation y := x * scale + y
        1,                  // Use step 1 for x
        &y[0][0][0],        // Input/output vector y, in operation y := x * scale + y
        1                   // Use step 2 for y
    );
#endif
}
```

MKL-based vector ops (PointwiseOps.cpp)

LaplaceSolver_1_4

```
[...]

void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],
    float (&z)[XDIM][YDIM][ZDIM], const float scale)
{
    [...]
}

void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM],
    const float scale)
{
#ifdef DO_NOT_USE_MKL
    // Just for reference -- implementation without MKL
#pragma omp parallel for
    for (int i = 0; i < XDIM; i++)
    for (int j = 0; j < YDIM; j++)
    for (int k = 0; k < ZDIM; k++)
        y[i][j][k] += x[i][j][k] * scale;
#else
    cblas_saxpy(
        XDIM * YDIM * ZDIM, // Length of vectors
        scale,              // Scale factor
        &x[0][0][0],        // Input vector x, in operation y := x * scale + y
        1,                  // Use step 1 for x
        &y[0][0][0],        // Input/output vector y, in operation y := x * scale + y
        1                   // Use step 2 for y
    );
#endif
}
```

CG edits (ConjugateGradients.cpp)

LaplaceSolver_1_4

```
void ConjugateGradients( ... ){
[...]
    // Algorithm : Line 2
    timerLaplacian.Restart(); ComputeLaplacian(matrix, x, z); timerLaplacian.Pause();
    Saxpy(z, f, r, -1);
    float nu = Norm(r);
[...]
        // Algorithm : Line 8
        timerSaxpy.Restart(); Saxpy(z, r, -alpha); timerSaxpy.Pause();
        nu=Norm(r);

        // Algorithm : Lines 9-12
        if (nu < nuMax || k == kMax) {
            timerSaxpy.Restart(); Saxpy(p, x, alpha); timerSaxpy.Pause();
[...]
        // Algorithm : Line 16
        timerSaxpy.Restart(); Saxpy(p, x, alpha); timerSaxpy.Pause();
        Saxpy(p, z, p, beta);
[...]
}
```

CG edits (ConjugateGradients.cpp)

LaplaceSolver_1_4

Original version of Saxpy (output not the same as an argument)

```
void ConjugateGradients( ... ){
[...]
    // Algorithm : Line 2
    timerLaplacian.Restart(); ComputeLaplacian(matrix, x, z); timerLaplacian.Pause();
    Saxpy(z, f, r, -1);
    float nu = Norm(r);
[...]
        // Algorithm : Line 8
        timerSaxpy.Restart(); Saxpy(z, r, -alpha); timerSaxpy.Pause();
        nu=Norm(r);

        // Algorithm : Lines 9-12
        if (nu < nuMax || k == kMax) {
            timerSaxpy.Restart(); Saxpy(p, x, alpha); timerSaxpy.Pause();
[...]
        // Algorithm : Line 16
        timerSaxpy.Restart(); Saxpy(p, x, alpha); timerSaxpy.Pause();
        Saxpy(p, z, p, beta);
[...]
}
```

CG edits (ConjugateGradients.cpp)

LaplaceSolver_1_4

Special case implementation applies here!

```
void ConjugateGradients( ... ){
[...]
    // Algorithm : Line 2
    timerLaplacian.Restart(); ComputeLaplacian(matrix, x, z); timerLaplacian.Pause();
    Saxpy(z, f, r, -1);
    float nu = Norm(r);
[...]
        // Algorithm : Line 8
        timerSaxpy.Restart(); Saxpy(z, r, -alpha); timerSaxpy.Pause();
        nu=Norm(r);

        // Algorithm : Lines 9-12
        if (nu < nuMax || k == kMax) {
            timerSaxpy.Restart(); Saxpy(p, x, alpha); timerSaxpy.Pause();
[...]
        // Algorithm : Line 16
        timerSaxpy.Restart(); Saxpy(p, x, alpha); timerSaxpy.Pause();
        Saxpy(p, z, p, beta);
[...]
}
```

Matrix-Vector multiply (MatVecMultiply.cpp)

LaplaceSolver_1_4

```
[...]

void MatVecMultiply(CSRMatrix& mat, const float *x, float *y)
{
    int N = mat.mSize; const auto rowOffsets = mat.GetRowOffsets();
    const auto columnIndices = mat.GetColumnIndices(); const auto values = mat.GetValues();

#ifdef DO_NOT_USE_MKL
#pragma omp parallel for
    for (int i = 0; i < N; i++) {
        y[i] = 0.;
        for (int k = rowOffsets[i]; k < rowOffsets[i+1]; k++) {
            const int j = columnIndices[k];
            y[i] += values[k] * x[j];
        }
    }
#else
    mkl_cspblas_scsrgemv( // (S)parse (CSR) (Ge)neral matrix (M)atrix-(V)ector product
        "N",              // Use the normal matrix, not its transpose
        &N,               // Size of the matrix
        values,           // values array (MKL denotes this as "a")
        rowOffsets,       // rowOffsets array (MKL denotes this as "ia")
        columnIndices,    // columnIndices array (MKL denotes this as "ja")
        x,                // Vector getting multiplied
        y                 // Vector where the product gets stored
    );
#endif
}
```

Approximately 17.3ms/call (not a very clean comparison)
You'll do more of this benchmarking as homework!