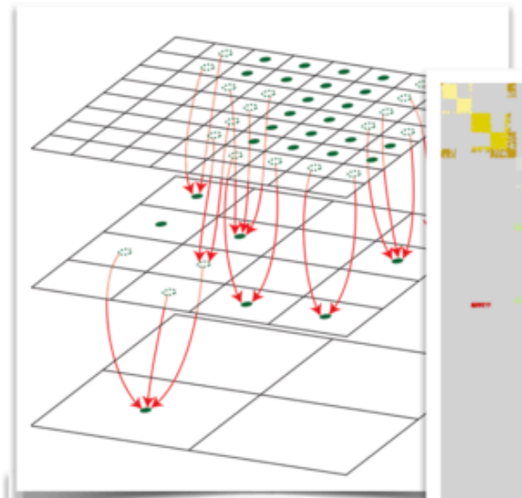

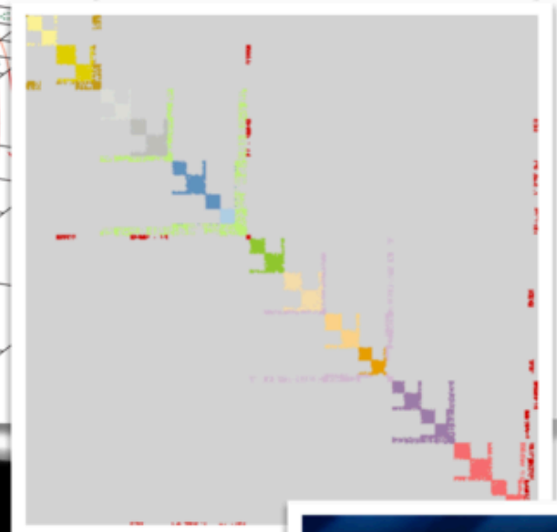

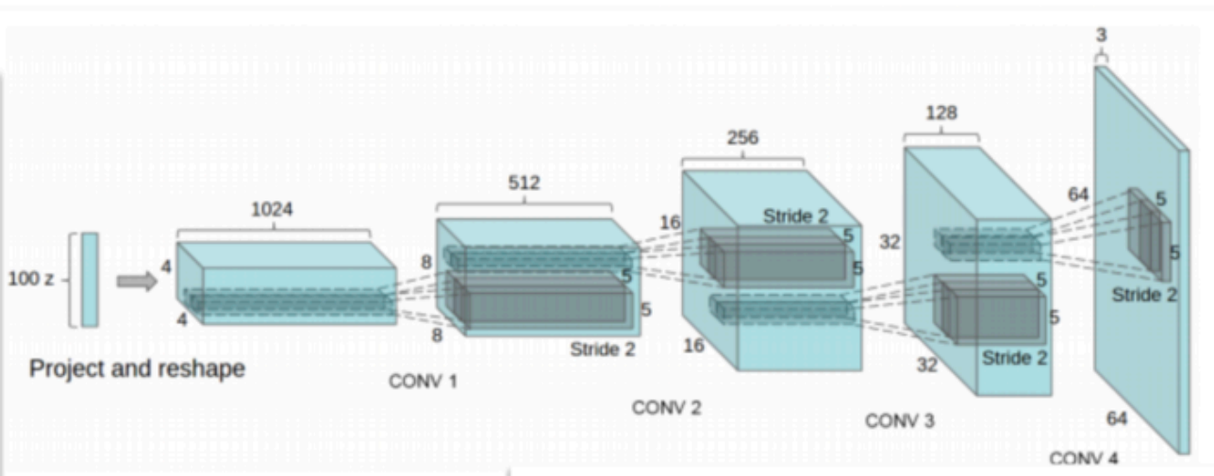

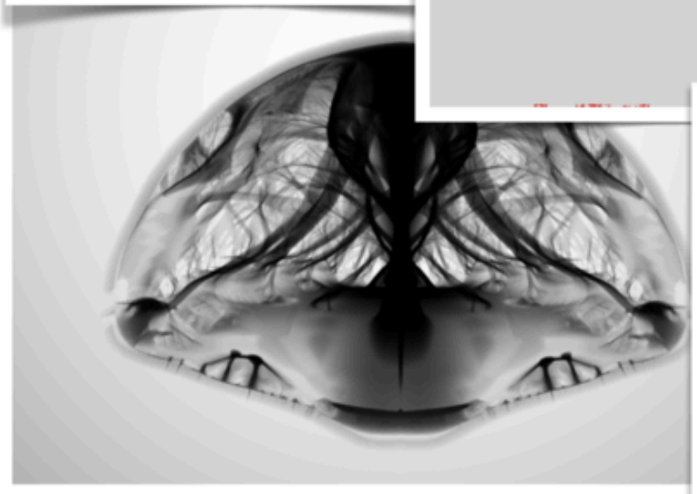

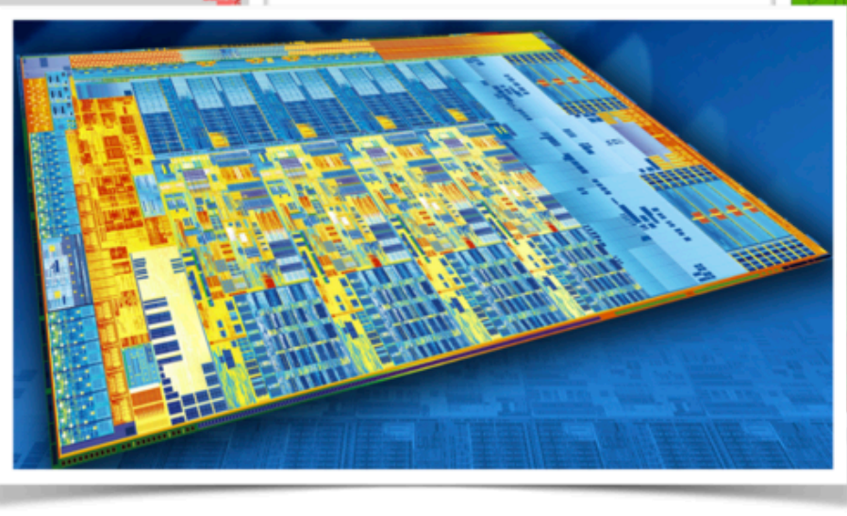

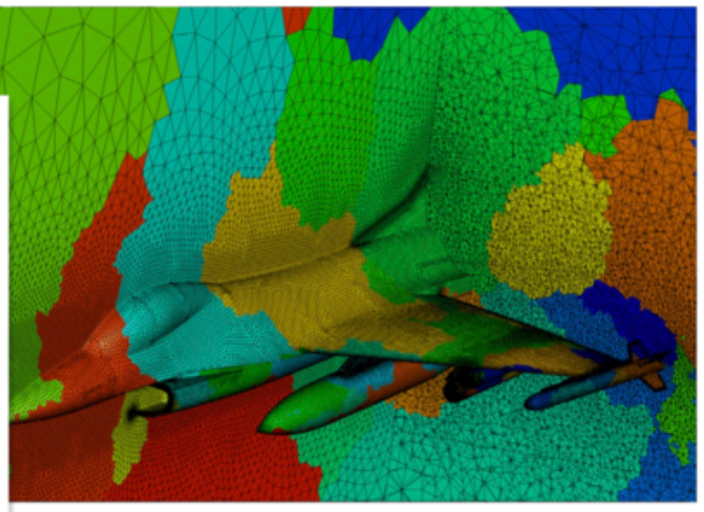

Lecture 8: Introduction to sparse matrix formats. Compressed Sparse Row storage.

Thursday February 16th 2023

Logistics

- Assignment #1 due tomorrow
- New programming assignment (#2) to be released next week
  - Available by Wednesay, maybe earlier
  - At least a week to complete, as before

Today's lecture

- One last optimization for our Conjugate Gradients prototype

- An introductory discussion of *sparse matrix formats* (where sparse matrices are explicitly stored)

- A deeper look at the *Compressed Sparse Row* format (the basis for some of our upcoming examples)

- Sample implementation of the Laplacian Solver (using Conjugate Gradients) using CSR matrix storage.

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: **if** $(\nu < \nu_{\max})$ **then return**
4: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: **for** $k = 0$ **to** k_{max} **do**
6: $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\alpha \leftarrow \rho/\sigma$
8: $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: **return**
12: **end if**
13: $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\rho \leftarrow \rho^{\text{new}}$
16: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: **end for**
18: **end procedure**

Development Plan

Design

- Define your objectives
- Choose a parallel-friendly theoretical formulation
- Set performance expectations
- Choose a promising algorithm

Implement

- Implement a prototype
- Organize code into reusable kernels

Accelerate

- Reorder/combine/pipeline operations
- Reduce resource utilization (try harder ...)
- Parallelize component kernels

Test case: Preconditioned Conjugate Gradients

Operations in red boxes can be executed in single streaming pass through memory

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\quad$ $\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: $\quad$ **if** $(\nu < \nu_{\max})$ **then return**
4: $\quad$ $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$ `Assume identity in assignment p <-r`
5: $\quad$ **for** $k = 0$ **to** k_{max} **do**
6: $\quad\quad$ $\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\quad\quad$ $\alpha \leftarrow \rho/\sigma$
8: $\quad\quad$ $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: $\quad\quad$ **if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\quad\quad\quad$ $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: $\quad\quad\quad$ **return**
12: $\quad\quad$ **end if**
13: $\quad\quad$ $\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$ `Assume identity z <- r`
14: $\quad\quad$ $\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\quad\quad$ $\rho \leftarrow \rho^{\text{new}}$
16: $\quad\quad$ $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: $\quad$ **end for**
18: **end procedure**

Test case: Preconditioned Conjugate Gradients

Operations in red boxes can be executed in single streaming pass through memory

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\quad\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: $\quad$**if** $(\nu < \nu_{\max})$ **then return**
4: $\quad\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: $\quad$**for** $k = 0$ **to** k_{max} **do**
6: $\quad\quad\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\quad\quad\alpha \leftarrow \rho/\sigma$
8: $\quad\quad\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: $\quad\quad$**if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\quad\quad\quad\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: $\quad\quad\quad$**return**
12: $\quad\quad$**end if**
13: $\quad\quad\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\quad\quad\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\quad\quad\rho \leftarrow \rho^{\text{new}}$
16: $\quad\quad\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: $\quad$**end for**
18: **end procedure**

Test case: Preconditioned Conjugate Gradients

Operations in red boxes can be executed in single streaming pass through memory

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\quad\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: $\quad$**if** $(\nu < \nu_{\max})$ **then return**
4: $\quad\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: $\quad$**for** $k = 0$ **to** k_{max} **do**
6: $\quad\quad\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\quad\quad\alpha \leftarrow \rho/\sigma$
8: $\quad\quad\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: $\quad\quad$**if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\quad\quad\quad\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: $\quad\quad\quad$**return**
12: $\quad\quad$**end if**
13: $\quad\quad\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\quad\quad\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\quad\quad\rho \leftarrow \rho^{\text{new}}$
16: $\quad\quad\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: $\quad$**end for**
18: **end procedure**

Test case: Preconditioned Conjugate Gradients

Operations in red boxes can be executed in single streaming pass through memory

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\quad\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: $\quad$**if** $(\nu < \nu_{\max})$ **then return**
4: $\quad\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: $\quad$**for** $k = 0$ **to** k_{max} **do**
6: $\quad\quad\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\quad\quad\alpha \leftarrow \rho/\sigma$
8: $\quad\quad\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: $\quad\quad$**if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\quad\quad\quad\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: $\quad\quad\quad$**return**
12: $\quad\quad$**end if**
13: $\quad\quad\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\quad\quad\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\quad\quad\rho \leftarrow \rho^{\text{new}}$
16: $\quad\quad\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: $\quad$**end for**
18: **end procedure**

Test case: Preconditioned Conjugate Gradients

Operations in red boxes can be executed in single streaming pass through memory

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\quad\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: $\quad$**if** $(\nu < \nu_{\max})$ **then return**
4: $\quad\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: $\quad$**for** $k = 0$ **to** k_{max} **do**
6: $\quad\quad\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\quad\quad\alpha \leftarrow \rho/\sigma$
8: $\quad\quad\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: $\quad\quad$**if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\quad\quad\quad\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: $\quad\quad\quad$**return**
12: $\quad\quad$**end if**
13: $\quad\quad\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\quad\quad\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\quad\quad\rho \leftarrow \rho^{\text{new}}$
16: $\quad\quad\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: $\quad$**end for**
18: **end procedure**

Test case: Preconditioned Conjugate Gradients

Operations in red boxes can be executed in single streaming pass through memory

1: **procedure** MGPCG($\mathbf{f}$, $\mathbf{x}$)
2: $\quad\mathbf{r} \leftarrow \mathbf{f} - \mathcal{L}\mathbf{x}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
3: $\quad$**if** $(\nu < \nu_{\max})$ **then return**
4: $\quad\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{p} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho \leftarrow \mathbf{p}^T\mathbf{r}$
5: $\quad$**for** $k = 0$ **to** k_{max} **do**
6: $\quad\quad\mathbf{z} \leftarrow \mathcal{L}\mathbf{p}$, $\sigma \leftarrow \mathbf{p}^T\mathbf{z}$
7: $\quad\quad\alpha \leftarrow \rho/\sigma$
8: $\quad\quad\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{z}$, $\mu \leftarrow \bar{\mathbf{r}}$, $\nu \leftarrow \|\mathbf{r} - \mu\|_\infty$
9: $\quad\quad$**if** $(\nu < \nu_{\max}$ **or** $k = k_{\max})$ **then**
10: $\quad\quad\quad\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$
11: $\quad\quad\quad$**return**
12: $\quad\quad$**end if**
13: $\quad\quad\mathbf{r} \leftarrow \mathbf{r} - \mu$, $\mathbf{z} \leftarrow \mathcal{M}^{-1}\mathbf{r}^{(\dagger)}$, $\rho^{\text{new}} \leftarrow \mathbf{z}^T\mathbf{r}$
14: $\quad\quad\beta \leftarrow \rho^{\text{new}}/\rho$
15: $\quad\quad\rho \leftarrow \rho^{\text{new}}$
16: $\quad\quad\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$, $\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$
17: $\quad$**end for**
18: **end procedure**

kernel coalescing

To amortize memory access cost and to improve the operations per byte.

Today's lecture

- One last optimization for our Conjugate Gradients prototype

  CG was an alternative to MatrixVectorMUL without the overhead of storing the matrices that made it compute bound. This was possible because of the fixed 6 locations in Laplacian.

- An introductory discussion of *sparse matrix formats* (where sparse matrices are <u>explicitly</u> stored)

- A deeper look at the *Compressed Sparse Row* format (the basis for some of our upcoming examples)

- Sample implementation of the Laplacian Solver (using Conjugate Gradients) using CSR matrix storage.

Pointwise Ops (PointwiseOps.cpp)

LaplaceSolver/LaplaceSolver_0_0

```
#include "Reductions.h"

#include <algorithm>

float Norm(const float (&x)[XDIM][YDIM][ZDIM])
{
    float result = 0.;

#pragma omp parallel for reduction(max:result)
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        result = std::max(result, std::abs(x[i][j][k]));

    return result;
}

float InnerProduct(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM])
{
    double result = 0.;

#pragma omp parallel for reduction(+:result)
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
        result += (double) x[i][j][k] * (double) y[i][j][k];

    return (float) result;
}
```

Recap

Example : The 3D Poisson equation

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

Recap

Example : The 3D Poisson equation

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

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

What about x & b?
How are they stored?

Recap

Example : The 3D Poisson equation

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

What about x & b?
How are they stored?

$$\mathbf{x} = \mathbf{b}$$

Recap

Example : The 3D Poisson equation

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

*Previously : Matrix-Vector Multiplication was **implicitly** encoded in a function call* ***ComputeLaplacian(u,Lu)***

Recap

Example : The 3D Poisson equation

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

*Now : We will investigate **explicitly** storing this matrix, making sure we leverage sparsity!*

Sparse Matrix Representations

Coordinate *Format (COO)*

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

Note zero-based numbering!
(a matter of convention)

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

Sparse Matrix Representations

Coordinate *Format (COO)*

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

Note zero-based numbering!
(a matter of convention)

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

Length of these is the number of nnz in the matrix. 3nnz

Sparse Matrix Representations

Coordinate *Format (COO)*

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

Note zero-based numbering!
(a matter of convention)

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

Sparse Matrix Representations

Coordinate *Format (COO)*

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

Note zero-based numbering!
(a matter of convention)

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

Sparse Matrix Representations

Coordinate *Format (COO)*

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

Note zero-based numbering!
(a matter of convention)

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

Sparse Matrix Representations

Compressed Sparse Row *(CSR)*

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

last entry of offsets the total nnz count

```
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}
int value[] = {10,11,12,13,14,15,16,17,18,19}
```

nth entry tells kth row

Storage cost: 2nnz+m

Sparse Matrix Representations

Compressed Sparse Row *(CSR)*

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

```
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}
int value[] = {10,11,12,13,14,15,16,17,18,19}
```

Sparse Matrix Representations

Compressed Sparse Row *(CSR)*

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

```
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}
int value[] = {10,11,12,13,14,15,16,17,18,19}
```

Sparse Matrix Representations

Compressed Sparse Row *(CSR)*

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

```
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}
int value[] = {10,11,12,13,14,15,16,17,18,19}
```

Sparse Matrix Representations

Compressed Sparse Row *(CSR)*

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

```
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}
int value[] = {10,11,12,13,14,15,16,17,18,19}
```

Sparse Matrix Representations

Compressed Sparse Row *(CSR)*

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

```
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}
int value[] = {10,11,12,13,14,15,16,17,18,19}
```

CSR matrix format

CSR = Compressed Sparse Row

- Components - ***N*** is the matrix size (assume #rows=#columns)
- Components - ***values*** is a flat array listing all (say, ***k***) non-zero elements of the matrix (in order, scanning left-to-right on columns, and top-to-bottom for rows; all elements of a row are listed consecutively, each row follows the previous one)
- Components - ***rowOffsets*** is an int-array of size (N+1)
  - ***rowOffsets[i]*** indicates where values of the i-th row start
  - ***rowOffsets[N]*** is the number of all non-zeros (i.e., ***k***)
- Components - ***columnIndices*** is in 1-to-1 correspondence with the values array, but lists the column index of each value
- Compressed sparse column (CSC) format is very similar, but it traverses matrices by columns, instead of rows

CSR Matrix structure (CSRMatrix.h)

LaplaceSolver_1_0

```
#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(); }
};
```

Prevents creating a copy of the pointer. Use CPPreference

CSR Matrix structure (CSRMatrix.h)

LaplaceSolver_1_0

```
#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)

CSR Matrix structure (CSRMatrix.h)

LaplaceSolver_1_0

```
#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(); }
};
```

Accessor functions (use these after initial allocation)

CSR Matrix structure (CSRMatrix.h)

LaplaceSolver_1_0

```
#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(); }
};
```

Data structure is very lean!
*Optimized for **using** vs. **building/updating** the matrix*

Matrix-Vector multiply (MatVecMultiply.h)

LaplaceSolver_1_0

Typical example of use : Multiplying matrix by a vector

```
#pragma once

#include "CSRMatrix.h"

void MatVecMultiply(CSRMatrix& mat, const float *x, float *y);
```

Matrix-Vector multiply (MatVecMultiply.h)

LaplaceSolver_1_0

```
#pragma once

#include "CSRMatrix.h"

void MatVecMultiply(CSRMatrix& mat, const float *x, float *y);
```

x[] and y[] are presumed to have arrays allocated on them, of size mat.mSize

Matrix-Vector multiply (MatVecMultiply.cpp)

LaplaceSolver_1_0

```
#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];
        }
    }
}
```

x[] and y[] are presumed to have arrays allocated on them, of size mat.mSize

Matrix-Vector multiply (MatVecMultiply.cpp)

LaplaceSolver_1_0

```
#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

```
#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

```
#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     Parallelization over rows, so it depends on row fitting in cache
    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];
        }                                   Allows sequential access to the matrix elements
    }                                       x[j] represents irregular accesses of the dense vector
}
```

*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***

CSR helper (CSRMatrixHelper.h)

LaplaceSolver_1_0

```
#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 ...]
    }
};
```

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

CSR helper (CSRMatrixHelper.h)

LaplaceSolver_1_0

```
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;
    }
};
```

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++) {
        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 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);
```