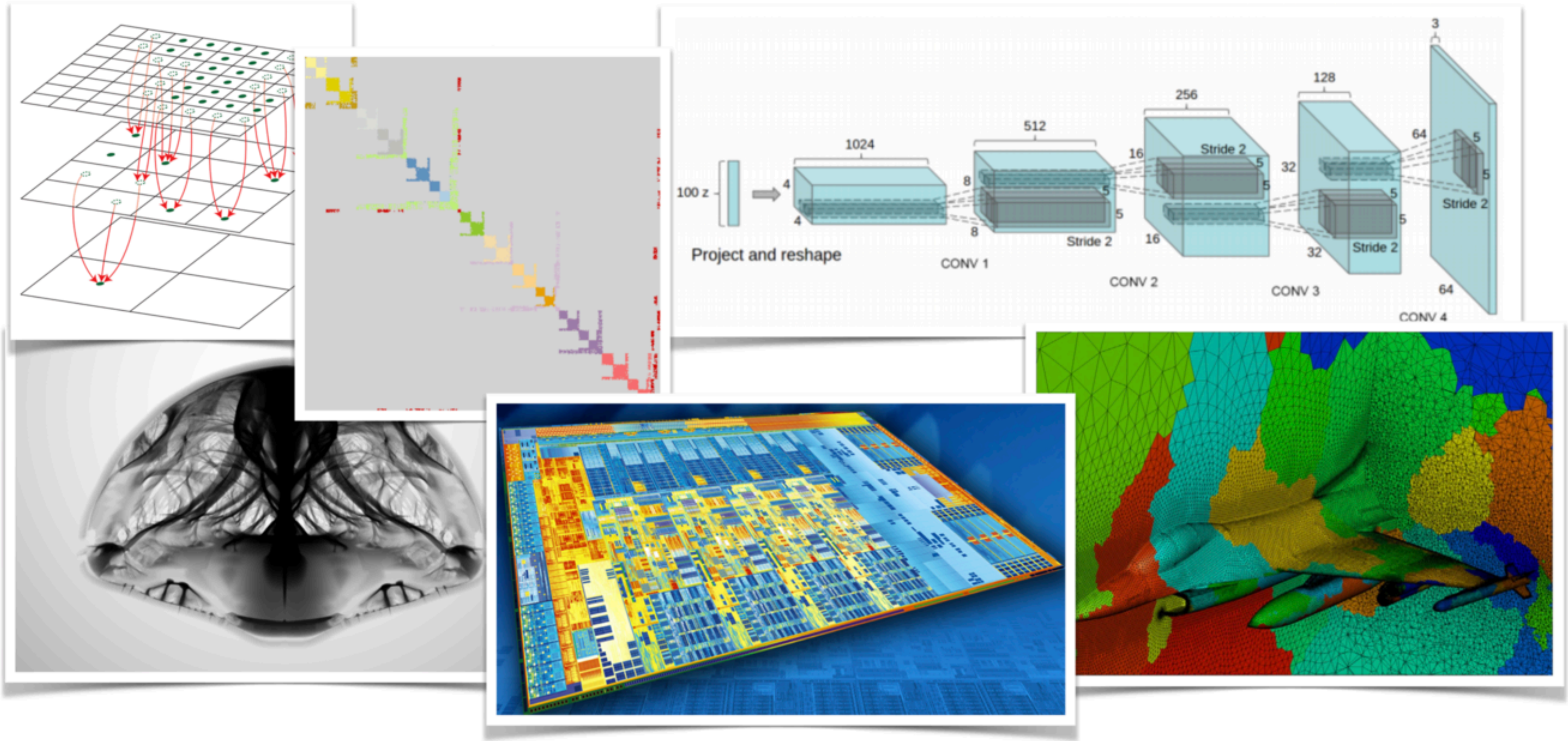


Lecture 2 : Introduction to Grid/Stencil Computations. A first case study - Application of a Laplacian Stencil

Thursday January 23rd 2023

Today’s lecture

- An introductory comment:
    - We’re jumping into action almost right away … this is just to give you a taste of problems/solutions/approach that we will track.
    - Might be a lot to absorb at first, but we’ll revisit multiple times!
    - Ask questions! We’ll go as slow as needed to make sure everyone is onboard!

Today's lecture

- Wrapping up our discussion from previous lecture

- Getting started with a concrete problem to study : Computations using grids and stencils (and a *specific* stencil; the Laplacian)

- Pointers to where to get sample code for the class (and how you get started with building it)

- Timing program executions (and some nontrivial reasons why variations might be observed)

- A first walk-through of how different approaches to parallelization could lead to different performance profiles (and a first mention of some reasons why)

Are we pursuing the right efficiency?

RANT ALERT!

Well-intended evaluation practices ...

> *"My serial implementation of algorithm X on machine Y ran in Z seconds. When I parallelized my code, I got a speedup of 15x on 16 cores ..."*

... are sometimes abused like this:

> *"... when I ported my implementation to CUDA, this numerical solver ran 200 times faster than my original MATLAB code ..."*

(frequent culprit: flawed understanding of how the computing platform works @ low level)

Are we pursuing the right efficiency?

RANT ALERT!

A different perspective ...

> *" ... after optimizing my code, the runtime is about 5x slower than **the best possible performance** that I could expect from this machine ..."*

*... i.e. 20% of maximum theoretical **efficiency!***

***Challenge :** How can we tell how fast the <u>best</u> implementation could have been? (without implementing it ...)*

Are we pursuing the right efficiency?

RANT ALERT!

Example : Solving the quadratic equation

$$ax^2 + bx + c = 0$$

What is the *minimum* amount of time needed to solve this?

Data access cost bound

"We cannot solve this faster than the time needed to read ***a,b,c*** *and write* ***x****"*

"We cannot solve this faster than the time needed evaluate the polynomial, for given values of a,b,c and x"
(i.e. 2 ADDs, 2 MULTs plus data access)

Solution verification bound

Equivalent operation bound

"We cannot solve this faster than the time it takes to compute a square root"

Are we pursuing the right efficiency?

RANT ALERT!

What about linear systems of equations?

$$\mathbf{Ax} = \mathbf{b}$$

"Textbook Efficiency"
(for certain types of problems)

*It is **theoretically possible** to compute the solution to a linear system (with certain properties) with a cost comparable to **10x the cost of verifying** that a given value **x** is an actual solution*

... or ...

*It is **theoretically possible** to compute the solution to a linear system at **10x the cost of computing*** the ***r=b-Ax*** and verifying that ***r=0***

Scope of Class

- At least for this semester (and likely in future years), focus will be on CPU-hosted parallel programming paradigms
  - Single-chassis multiprocessors (but substantial similarity to GPU programming)
  - Will not focus on distributed or highly heterogeneous programming (e.g. MPI)

Scope of Class

- Technical topics
  - Multithreaded programming; Synchronization; Using the OpenMP API
  - Instruction Level Parallelism; Vectorization and challenges; SIMD intrinsics
  - Memory hierarchy and its implications; Caches; Virtual Memory
  - Assessing efficiency, predicting parallel potential, and benchmarking performance
  - Understanding the role of compute and/or memory throughput as a limiting factor of performance
  - Optimizing data structures for target architecture; Memory allocation and management

Scope of Class

- Tentative application focus (may adjust slightly)
  - Sparse linear algebra; Matrix representations; Iterative solvers for sparse systems
  - Dense linear algebra; Matrix/Vector operations; Matrix Factorizations; Using the MKL library
  - Grid and stencil computations; Convolutions and their use in neural networks and/or image processing applications.
  - Sparse data structures (OpenVDB/NanoVDB) and their implications to bandwidth-optimized parallel programming.

Today's lecture

- Wrapping up our discussion from previous lecture

- Getting started with a concrete problem to study :
  Computations using grids and stencils
  (and a *specific* stencil; the Laplacian)

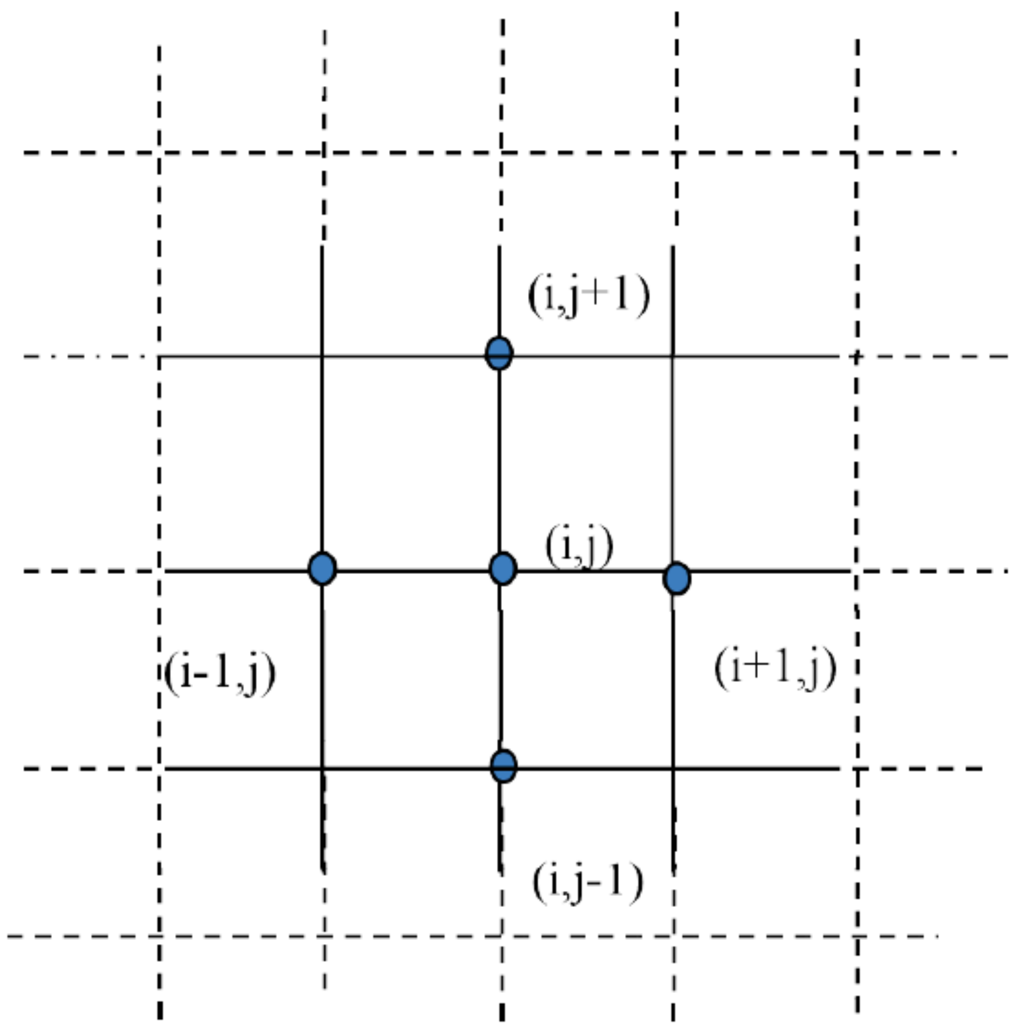
- Pointers to where to get sample code for the class
  (and how you get started with building it)

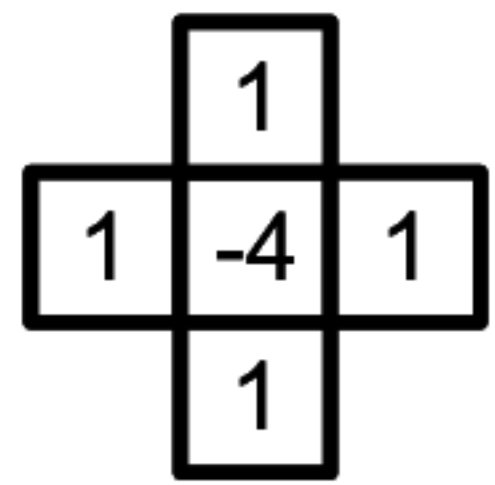
- Timing program executions (and some nontrivial reasons why variations might be observed)

- A first walk-through of how different approaches to parallelization could lead to different performance profiles
  (and a first mention of some reasons why)

A first case study …

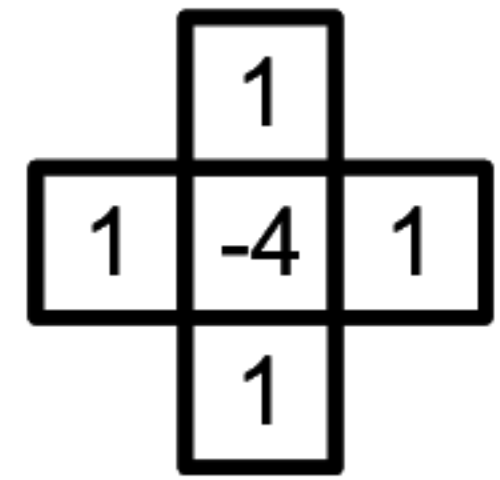
Laplacian Stencil Application
(Today : on 2D grid)

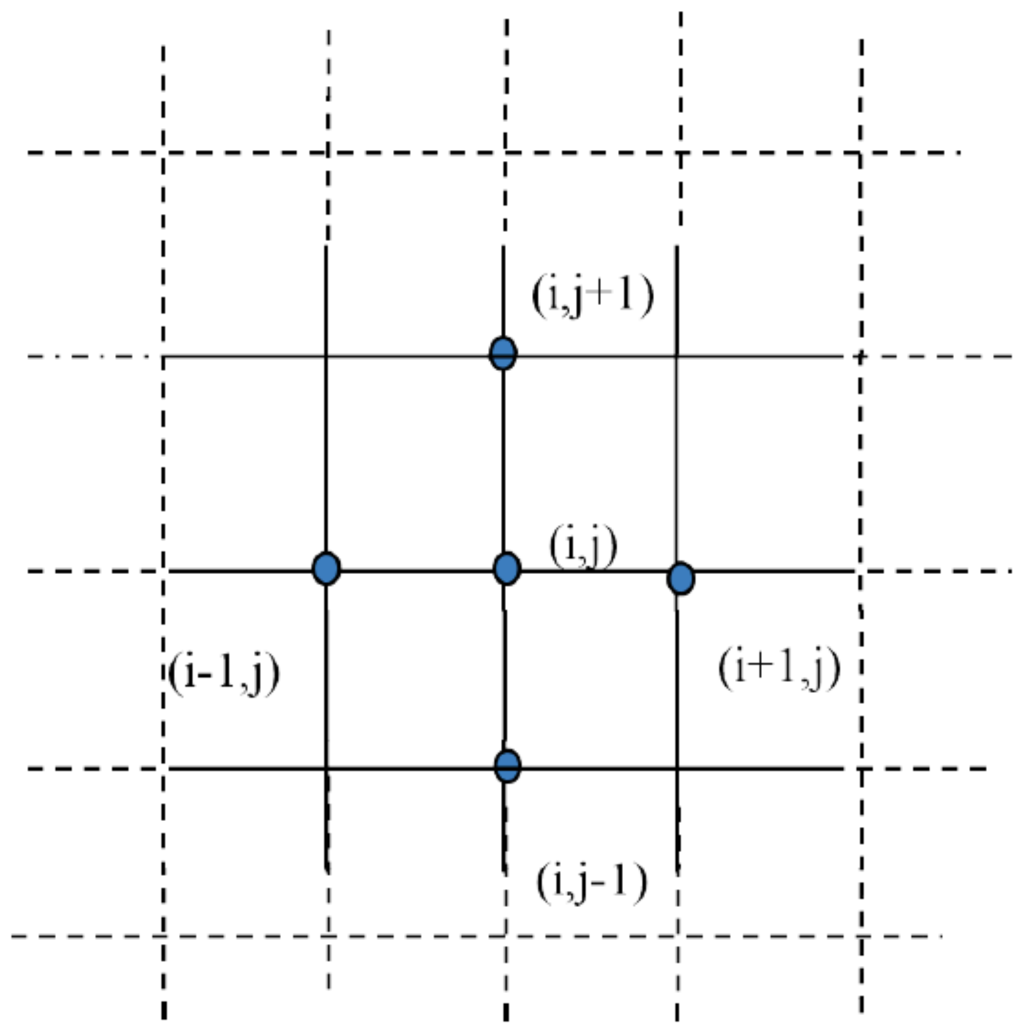




A first case study …

Laplacian Stencil Application
(Today : on 2D grid)



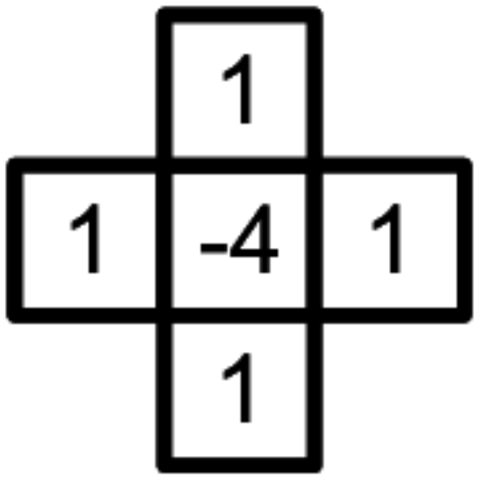


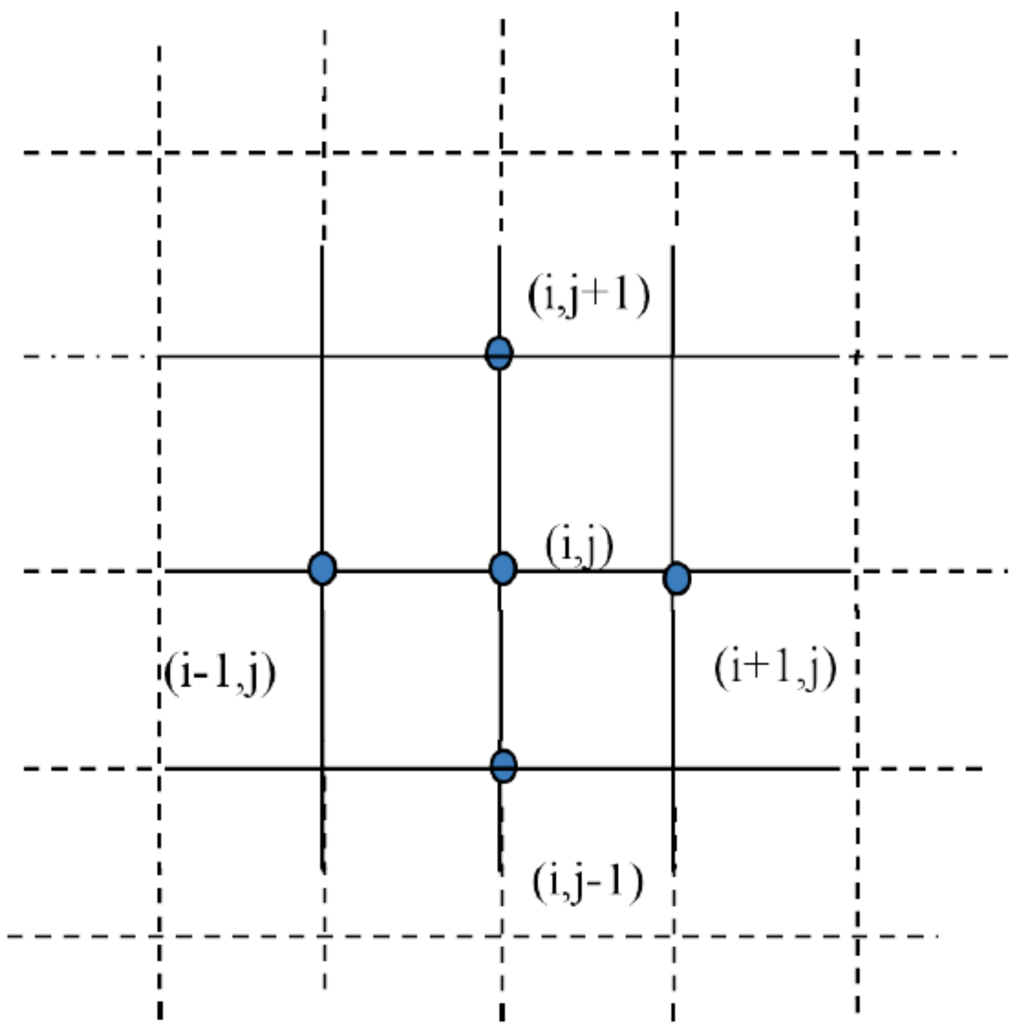
General idea (<u>highly</u> conceptual):

```
float u[N][N], Lu[N][N];
for i = 0,...,N-1
  for j = 0,…,N-1
    Lu[i][j] = -4u[i][j] + u[i+1][j]
      +u[i-1][j] + u[i][j+1] + u[i][j-1]
```

A first case study …

Laplacian Stencil Application
(Today : on 2D grid)





Applications:
- *Image processing*
- *Convolutional neural networks*
- *Computational physics*

… and many more

Accessing code examples

- All benchmarks discussed in class can be downloaded from the GitHub public repository

  https://github.com/sifakis/CS639S23_Demos

  (please report any access issues)

- Today's examples in Folder "LaplacianStencil"

- Subfolder of specific example listed on upper-right of each slide

Accessing code examples

- If you need a quick walk-through to familiarize yourselves with the “Git” version control system, the following guide is helpful : https://guides.github.com/activities/hello-world

- If you just want to get up-and-running, simply download a .zip archive from GitHub!

- Examples are set-up to easily compile without needing (yet) external libraries. Parallelization uses the OpenMP API.

OpenMP? What about OpenMP?

- OpenMP is an API to support parallel programming on shared-memory multiprocessors
- Support incorporated on most C++ compilers
    - For icc, compile with the -qopenmp option
    - For g++, compile with the -fopenmp option
    - For VS/Win, compile with the /openmp option
- Offline reading (but we'll review here, too!) :
    - https://www.openmp.org/wp-content/uploads/ntu-vanderpas.pdf

Accessing code examples

- Execution times reported on a Single CPU Workstation with an Intel Xeon 6210U Processor
  (20 cores @ 2.5Ghz)

- Peak Memory Bandwidth on this platform ~138GB/sec

- Peak Compute Bandwidth on this platform ~2.7TFLOPS

- How to find the specifications for your own machine?
  For Intel : http://ark.intel.com
  For AMD : https://www.amd.com/en/products/specifications/processors

Timer Module (timer.h)

LaplacianStencil_XX_YY (all versions)

```
#pragma once

#include <chrono>
#include <cstring>
#include <iostream>

struct Timer
{
    using clock_t = std::chrono::high_resolution_clock;
    using time_point_t = std::chrono::time_point<clock_t>;

    time_point_t mStartTime;
    time_point_t mStopTime;

    void Start()
    {
        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;
    }

};
```

Timer Module (timer.h)

LaplacianStencil_XX_YY (all versions)

```
#pragma once

#include <chrono>
#include <cstring>
#include <iostream>

struct Timer
{
    using clock_t = std::chrono::high_resolution_clock;
    using time_point_t = std::chrono::time_point<clock_t>;

    time_point_t mStartTime;
    time_point_t mStopTime;

    void Start()
    {
        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;
    }

};
```

A "Timer" object will be used via the start()/stop() functions;
start() "starts" a stopwatch
stop() "halts" the timer, and reports the elapsed time, with a message

Benchmark launcher (main.cpp)

LaplacianStencil_0_[0-6]

```
#include "Timer.h"
#include "Laplacian.h"

#include <iomanip>

int main(int argc, char *argv[])
{
    using array_t = float (&) [XDIM][YDIM];

    float *uRaw = new float [XDIM*YDIM];
    float *LuRaw = new float [XDIM*YDIM];
    array_t u = reinterpret_cast<array_t>(*uRaw);
    array_t Lu = reinterpret_cast<array_t>(*LuRaw);

    Timer timer;

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

    return 0;
}
```

Benchmark launcher (main.cpp)

LaplacianStencil_0_[0-6]

```
#include "Timer.h"
#include "Laplacian.h"

#include <iomanip>

int main(int argc, char *argv[])
{
    using array_t = float (&) [XDIM][YDIM];

    float *uRaw = new float [XDIM*YDIM];
    float *LuRaw = new float [XDIM*YDIM];
    array_t u = reinterpret_cast<array_t>(*uRaw);
    array_t Lu = reinterpret_cast<array_t>(*LuRaw);

    Timer timer;

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

    return 0;
}
```

Allocate u & Lu so that they can be used as 2-dimensional arrays (i.e. u[56][67])

Benchmark launcher (main.cpp)

LaplacianStencil_0_[0-6]

```
#include "Timer.h"
#include "Laplacian.h"

#include <iomanip>

int main(int argc, char *argv[])
{
    using array_t = float (&) [XDIM][YDIM];

    float *uRaw = new float [XDIM*YDIM];
    float *LuRaw = new float [XDIM*YDIM];
    array_t u = reinterpret_cast<array_t>(*uRaw);
    array_t Lu = reinterpret_cast<array_t>(*LuRaw);

    Timer timer;

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

    return 0;
}
```

Use “Timer” class to time execution of your test(s)

Benchmark launcher (main.cpp)

LaplacianStencil_0_[0-6]

```
#include "Timer.h"
#include "Laplacian.h"

#include <iomanip>

int main(int argc, char *argv[])
{
    using array_t = float (&) [XDIM][YDIM];

    float *uRaw = new float [XDIM*YDIM];
    float *LuRaw = new float [XDIM*YDIM];
    array_t u = reinterpret_cast<array_t>(*uRaw);
    array_t Lu = reinterpret_cast<array_t>(*LuRaw);

    Timer timer;

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

    return 0;
}
```

This is the actual call to our “benchmark” (executed and measured several times)

Kernel header (Laplacian.h)

LaplacianStencil_0_0

```
#pragma once

#define XDIM 16384
#define YDIM 16384

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

Kernel header (Laplacian.h)

LaplacianStencil_0_0

Size of grid presumed constant and known at time of compilation

```
#pragma once

#define XDIM 16384
#define YDIM 16384

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

Kernel Body (Laplacian.cpp)

LaplacianStencil_0_0

```
#include "Laplacian.h"

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM])
{

#pragma omp parallel for
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
        Lu[i][j] =
            -4 * u[i][j]
            + u[i+1][j]
            + u[i-1][j]
            + u[i][j+1]
            + u[i][j-1];

}
```

Kernel Body (Laplacian.cpp)

LaplacianStencil_0_0

```
#include "Laplacian.h"

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM])
{

#pragma omp parallel for
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
        Lu[i][j] =
            -4 * u[i][j]
            + u[i+1][j]
            + u[i-1][j]
            + u[i][j+1]
            + u[i][j-1];

}
```

OpenMP used to parallelize outer loop

Kernel Body (Laplacian.cpp)

LaplacianStencil_0_0

```
#include "Laplacian.h"

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM])
{

#pragma omp parallel for
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
        Lu[i][j] =
            -4 * u[i][j]
            + u[i+1][j]
            + u[i-1][j]
            + u[i][j+1]
            + u[i][j-1];

}
```

Execution:

```
Running test iteration  1 [Elapsed time : 120.472ms]
Running test iteration  2 [Elapsed time : 25.3752ms]
Running test iteration  3 [Elapsed time : 24.3025ms]
Running test iteration  4 [Elapsed time : 23.0271ms]
Running test iteration  5 [Elapsed time : 22.6208ms]
Running test iteration  6 [Elapsed time : 22.9576ms]
Running test iteration  7 [Elapsed time : 22.5305ms]
Running test iteration  8 [Elapsed time : 23.7184ms]
Running test iteration  9 [Elapsed time : 22.691ms]
Running test iteration 10 [Elapsed time : 24.7188ms]
```

Kernel Body (Laplacian.cpp)

LaplacianStencil_0_0

```
#include "Laplacian.h"

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM])
{

#pragma omp parallel for
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
        Lu[i][j] =
            -4 * u[i][j]
            + u[i+1][j]
            + u[i-1][j]
            + u[i][j+1]
            + u[i][j-1];

}
```

Why are these times not all equal?
Is it "reasonable" that this execution takes this long?

Execution:

```
Running test iteration  1 [Elapsed time : 120.472ms]
Running test iteration  2 [Elapsed time : 25.3752ms]
Running test iteration  3 [Elapsed time : 24.3025ms]
Running test iteration  4 [Elapsed time : 23.0271ms]
Running test iteration  5 [Elapsed time : 22.6208ms]
Running test iteration  6 [Elapsed time : 22.9576ms]
Running test iteration  7 [Elapsed time : 22.5305ms]
Running test iteration  8 [Elapsed time : 23.7184ms]
Running test iteration  9 [Elapsed time : 22.691ms]
Running test iteration 10 [Elapsed time : 24.7188ms]
```

Kernel Body (Laplacian.cpp)

LaplacianStencil_0_1

```
#include "Laplacian.h"

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM])
{

    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
        Lu[i][j] =
            -4 * u[i][j]
            + u[i+1][j]
            + u[i-1][j]
            + u[i][j+1]
            + u[i][j-1];

}
```

Without OpenMP parallelization

Execution:

```
Running test iteration  1 [Elapsed time : 678.226ms]
Running test iteration  2 [Elapsed time : 244.218ms]
Running test iteration  3 [Elapsed time : 244.315ms]
Running test iteration  4 [Elapsed time : 246.056ms]
Running test iteration  5 [Elapsed time : 244.506ms]
Running test iteration  6 [Elapsed time : 243.8ms]
Running test iteration  7 [Elapsed time : 243.287ms]
Running test iteration  8 [Elapsed time : 245.844ms]
Running test iteration  9 [Elapsed time : 244.315ms]
Running test iteration 10 [Elapsed time : 245.566ms]
```

Kernel Body (Laplacian.cpp)

LaplacianStencil_0_0

```
#include "Laplacian.h"

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM])
{

#pragma omp parallel for
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
        Lu[i][j] =
            -4 * u[i][j]
            + u[i+1][j]
            + u[i-1][j]
            + u[i][j+1]
            + u[i][j-1];

}
```

Why are these times not all equal?
Is it "reasonable" that this execution takes this long?

Execution:

```
Running test iteration  1 [Elapsed time : 120.472ms]
Running test iteration  2 [Elapsed time : 25.3752ms]
Running test iteration  3 [Elapsed time : 24.3025ms]
Running test iteration  4 [Elapsed time : 23.0271ms]
Running test iteration  5 [Elapsed time : 22.6208ms]
Running test iteration  6 [Elapsed time : 22.9576ms]
Running test iteration  7 [Elapsed time : 22.5305ms]
Running test iteration  8 [Elapsed time : 23.7184ms]
Running test iteration  9 [Elapsed time : 22.691ms]
Running test iteration 10 [Elapsed time : 24.7188ms]
```

Establishing boundaries of performance

- Remember: The computing platform has limited capacity for (a) Moving data between the CPU and Main Memory, and (b) Executing calculations on data

- Most of the examples we'll see here are constrained by *memory bandwidth* (we typically call such algorithms "memory bound") rather than *computing bandwidth.*

- Let's start by exploring memory constraints in our examples …

Platform Specifications (theoretical)

- Execution times reported on a Single CPU Workstation with an Intel Xeon 6210U Processor
  (20 cores @ 2.5Ghz)

- Peak Memory Bandwidth on this platform ~138GB/sec

- Peak Compute Bandwidth on this platform ~2.7TFLOPS

- How to find the specifications for your own machine?
  For Intel : http://ark.intel.com
  For AMD : https://www.amd.com/en/products/specifications/processors

Platform Specifications (practical; memory bandwidth)

- The STREAM benchmark is a well-established metric of "practical" memory bandwidth capability. https://www.cs.virginia.edu/stream/

- Runs 4 tests
  (a) Copy array a[] to b[] - "Copy"
  (b) Scale array a[] by constant value - "Scale"
  (c) Add respective entries in a[] and b[] - "Add"
  (d) Multiply entries in a[] & b[] and add to c[] - "Triad"

Platform Specifications (practical; memory bandwidth)

- The STREAM benchmark is a well-established metric of "practical" memory bandwidth capability. *https://www.cs.virginia.edu/stream/*

- Runs 4 tests
  (a) Copy array a[] to b[] - "Copy"
  (b) Scale array a[] by constant value - "Scale"
  (c) Add respective entries in a[] and b[] - "Add"
  (d) Multiply entries in a[] & b[] and add to c[] - "Triad"

Practical bandwidth: 80-90GB/s

Execution:

```
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function       Rate (MB/s)   Avg time     Min time     Max time
Copy:      111476.5183     0.0004       0.0003       0.0005
Scale:      93271.5274     0.0004       0.0003       0.0004
Add:        70271.0618     0.0008       0.0007       0.0008
Triad:      96559.5165     0.0006       0.0005       0.0006
```

Kernel Body (Laplacian.cpp)

LaplacianStencil_0_0

```
#include "Laplacian.h"

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM])
{

#pragma omp parallel for
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
        Lu[i][j] =
            -4 * u[i][j]
            + u[i+1][j]
            + u[i-1][j]
            + u[i][j+1]
            + u[i][j-1];

}
```

"At-a-minimum" cost : We need to read each entry u[i][j], and write each entry Lu[i][j]

Two arrays of size 16K x 16K floats 2GB total

*At 80-90GB/s : Should take **22-25ms***

Kernel Body (Laplacian.cpp)

LaplacianStencil_0_0

```
#include "Laplacian.h"

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM])
{

#pragma omp parallel for
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
        Lu[i][j] =
            -4 * u[i][j]
            + u[i+1][j]
            + u[i-1][j]
            + u[i][j+1]
            + u[i][j-1];

}
```

Execution:

```
Running test iteration  1 [Elapsed time : 120.472ms]
Running test iteration  2 [Elapsed time : 25.3752ms]
Running test iteration  3 [Elapsed time : 24.3025ms]
Running test iteration  4 [Elapsed time : 23.0271ms]
Running test iteration  5 [Elapsed time : 22.6208ms]
Running test iteration  6 [Elapsed time : 22.9576ms]
Running test iteration  7 [Elapsed time : 22.5305ms]
Running test iteration  8 [Elapsed time : 23.7184ms]
Running test iteration  9 [Elapsed time : 22.691ms]
Running test iteration 10 [Elapsed time : 24.7188ms]
```

In light of this, the efficiency of this execution is very high!

(not frequent for workloads to exceed 80-90% of peak efficiency …)

Kernel Body (Laplacian.cpp)

LaplacianStencil_0_0

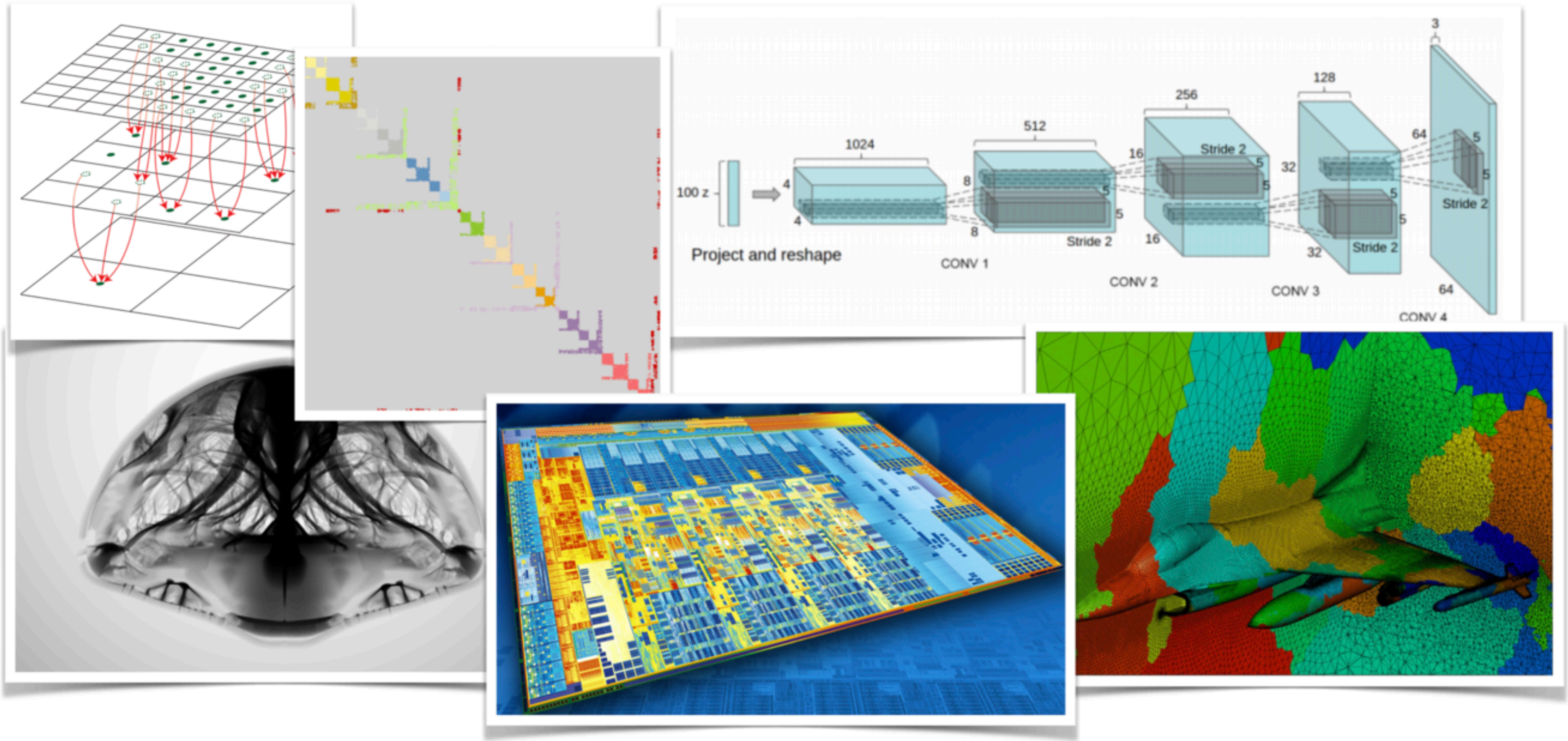
```
#include "Laplacian.h"

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM])
{

#pragma omp parallel for
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
        Lu[i][j] =
            -4 * u[i][j]
            + u[i+1][j]
            + u[i-1][j]
            + u[i][j+1]
            + u[i][j-1];

}
```

Certainly "memory bound" - we'll access computational burden later …

Execution:

```
Running test iteration  1 [Elapsed time : 120.472ms]
Running test iteration  2 [Elapsed time : 25.3752ms]
Running test iteration  3 [Elapsed time : 24.3025ms]
Running test iteration  4 [Elapsed time : 23.0271ms]
Running test iteration  5 [Elapsed time : 22.6208ms]
Running test iteration  6 [Elapsed time : 22.9576ms]
Running test iteration  7 [Elapsed time : 22.5305ms]
Running test iteration  8 [Elapsed time : 23.7184ms]
Running test iteration  9 [Elapsed time : 22.691ms]
Running test iteration 10 [Elapsed time : 24.7188ms]
```

Note that "neighbor accesses" didn't quite count multiple times! (Pretty close to having been "perfectly cached")



Lecture 2 : Introduction to Grid/Stencil Computations.
A first case study - Application of a Laplacian Stencil

Thursday January 23rd 2023

(overflow/preview)

Kernel header (Laplacian.h)

LaplacianStencil_0_2

```
#pragma once

#define XDIM 4096
#define YDIM 4096

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

Size reduced 16K -> 4K

Execution:

```
Running test iteration  1 [Elapsed time : 21.3287ms]
Running test iteration  2 [Elapsed time : 2.81527ms]
Running test iteration  3 [Elapsed time : 1.66752ms]
Running test iteration  4 [Elapsed time : 1.57543ms]
Running test iteration  5 [Elapsed time : 1.50367ms]
Running test iteration  6 [Elapsed time : 1.48125ms]
Running test iteration  7 [Elapsed time : 1.52556ms]
Running test iteration  8 [Elapsed time : 1.33879ms]
Running test iteration  9 [Elapsed time : 1.3976ms]
Running test iteration 10 [Elapsed time : 1.40909ms]
```

Kernel header (Laplacian.h)

LaplacianStencil_0_3

Size reduced 16K -> 2K

```
#pragma once

#define XDIM 2048
#define YDIM 2048

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

Execution:

```
Running test iteration  1 [Elapsed time : 25.4213ms]
Running test iteration  2 [Elapsed time : 10.8833ms]
Running test iteration  3 [Elapsed time : 0.807804ms]
Running test iteration  4 [Elapsed time : 0.325908ms]
Running test iteration  5 [Elapsed time : 0.307869ms]
Running test iteration  6 [Elapsed time : 0.29541ms]
Running test iteration  7 [Elapsed time : 0.298488ms]
Running test iteration  8 [Elapsed time : 0.298959ms]
Running test iteration  9 [Elapsed time : 0.298472ms]
Running test iteration 10 [Elapsed time : 0.299072ms]
```

Kernel Body (Laplacian.cpp)

LaplacianStencil_0_4

```
#include "Laplacian.h"

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM])
{

#pragma omp parallel for
   for (int j = 1; j < YDIM-1; j++)
   for (int i = 1; i < XDIM-1; i++)
        Lu[i][j] =
            -4 * u[i][j]
            + u[i+1][j]
            + u[i-1][j]
            + u[i][j+1]
            + u[i][j-1];

}
```

Size reduced 16K -> 4K
Loop Order Swapped

Execution:

```
Running test iteration  1 [Elapsed time : 88.9032ms]
Running test iteration  2 [Elapsed time : 50.2971ms]
Running test iteration  3 [Elapsed time : 50.5499ms]
Running test iteration  4 [Elapsed time : 50.2705ms]
Running test iteration  5 [Elapsed time : 51.0571ms]
Running test iteration  6 [Elapsed time : 51.5478ms]
Running test iteration  7 [Elapsed time : 51.4321ms]
Running test iteration  8 [Elapsed time : 50.3991ms]
Running test iteration  9 [Elapsed time : 50.4688ms]
Running test iteration 10 [Elapsed time : 52.8201ms]
```

Kernel Body (Laplacian.cpp)

LaplacianStencil_0_4

```
#include "Laplacian.h"

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM])
{

#pragma omp parallel for
    for (int j = 1; j < YDIM-1; j++)
    for (int i = 1; i < XDIM-1; i++)
        Lu[i][j] =
            -4 * u[i][j]
            + u[i+1][j]
            + u[i-1][j]
            + u[i][j+1]
            + u[i][j-1];

}
```

Size reduced 16K -> 4K
Loop Order Swapped

Execution:

```
Running test iteration  1 [Elapsed time : 88.9032ms]
Running test iteration  2 [Elapsed time : 50.2971ms]
Running test iteration  3 [Elapsed time : 50.5499ms]
Running test iteration  4 [Elapsed time : 50.2705ms]
Running test iteration  5 [Elapsed time : 51.0571ms]
Running test iteration  6 [Elapsed time : 51.5478ms]
Running test iteration  7 [Elapsed time : 51.4321ms]
Running test iteration  8 [Elapsed time : 50.3991ms]
Running test iteration  9 [Elapsed time : 50.4688ms]
Running test iteration 10 [Elapsed time : 52.8201ms]
```

Kernel Body (Laplacian.cpp)

LaplacianStencil_0_5

```
#include "Laplacian.h"

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM])
{

#pragma omp parallel for
    for (int j = 1; j < YDIM-1; j++)
    for (int i = 1; i < XDIM-1; i++)
        Lu[i][j] =
            -4 * u[i][j]
            + u[i+1][j]
            + u[i-1][j]
            + u[i][j+1]
            + u[i][j-1];

}
```

Size reduced 16K -> 2K
Loop Order Swapped

Execution:

```
Running test iteration  1 [Elapsed time : 53.1412ms]
Running test iteration  2 [Elapsed time : 2.73531ms]
Running test iteration  3 [Elapsed time : 2.6788ms]
Running test iteration  4 [Elapsed time : 2.66177ms]
Running test iteration  5 [Elapsed time : 2.66733ms]
Running test iteration  6 [Elapsed time : 2.6668ms]
Running test iteration  7 [Elapsed time : 2.63204ms]
Running test iteration  8 [Elapsed time : 2.67448ms]
Running test iteration  9 [Elapsed time : 2.6665ms]
Running test iteration 10 [Elapsed time : 2.66042ms]
```

Kernel Body (Laplacian.cpp)

LaplacianStencil_0_6

```
#include "Laplacian.h"

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM])
{

#pragma omp parallel for
    for (int j = 1; j < YDIM-1; j++)
    for (int i = 1; i < XDIM-1; i++)
        Lu[i][j] =
            -4 * u[i][j]
            + u[i+1][j]
            + u[i-1][j]
            + u[i][j+1]
            + u[i][j-1];

}
```

Original Size
Loop Order Swapped

Execution:

```
Running test iteration  1 [Elapsed time : 2034.53ms]
Running test iteration  2 [Elapsed time : 1814.3ms]
Running test iteration  3 [Elapsed time : 1873.85ms]
Running test iteration  4 [Elapsed time : 1779.44ms]
Running test iteration  5 [Elapsed time : 1731.12ms]
Running test iteration  6 [Elapsed time : 1809.28ms]
Running test iteration  7 [Elapsed time : 1825.35ms]
Running test iteration  8 [Elapsed time : 1725.44ms]
Running test iteration  9 [Elapsed time : 1806.62ms]
Running test iteration 10 [Elapsed time : 1882.4ms]
```

Benchmark launcher (main.cpp)

LaplacianStencil_0_7

```
#include "Timer.h"
#include "Laplacian.h"

#include <iomanip>

int main(int argc, char *argv[])
{
    float **u = new float *[XDIM];
    float **Lu = new float *[XDIM];
    for (int i = 0; i < XDIM; i++){
        u[i] = new float [YDIM];
        Lu[i] = new float [YDIM];
    }

    Timer timer;

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

    return 0;
}
```

Arrays (u,Lu) allocated as “arrays of pointers to allocated arrays”

Kernel header (Laplacian.h)

LaplacianStencil_0_3

```
#pragma once

#define XDIM 2048
#define YDIM 2048

void ComputeLaplacian(const float **u, float **Lu);
```

Arguments passed as double pointers (Laplacian.cpp is largely unchanged)

Execution:

```
Running test iteration  1 [Elapsed time : 23.4823ms]
Running test iteration  2 [Elapsed time : 9.35612ms]
Running test iteration  3 [Elapsed time : 3.60061ms]
Running test iteration  4 [Elapsed time : 7.08704ms]
Running test iteration  5 [Elapsed time : 0.438221ms]
Running test iteration  6 [Elapsed time : 8.44043ms]
Running test iteration  7 [Elapsed time : 4.80748ms]
Running test iteration  8 [Elapsed time : 6.9574ms]
Running test iteration  9 [Elapsed time : 8.16184ms]
Running test iteration 10 [Elapsed time : 0.285378ms]
```

Kernel header (Laplacian.h)

LaplacianStencil_0_3

Arguments passed as double pointers (Laplacian.cpp is largely unchanged)

```
#pragma once

#define XDIM 2048
#define YDIM 2048

void ComputeLaplacian(const float **u, float **Lu);
```

Why are run times so volatile? (and worse than before, generally)

Execution:

```
Running test iteration  1 [Elapsed time : 23.4823ms]
Running test iteration  2 [Elapsed time : 9.35612ms]
Running test iteration  3 [Elapsed time : 3.60061ms]
Running test iteration  4 [Elapsed time : 7.08704ms]
Running test iteration  5 [Elapsed time : 0.438221ms]
Running test iteration  6 [Elapsed time : 8.44043ms]
Running test iteration  7 [Elapsed time : 4.80748ms]
Running test iteration  8 [Elapsed time : 6.9574ms]
Running test iteration  9 [Elapsed time : 8.16184ms]
Running test iteration 10 [Elapsed time : 0.285378ms]
```