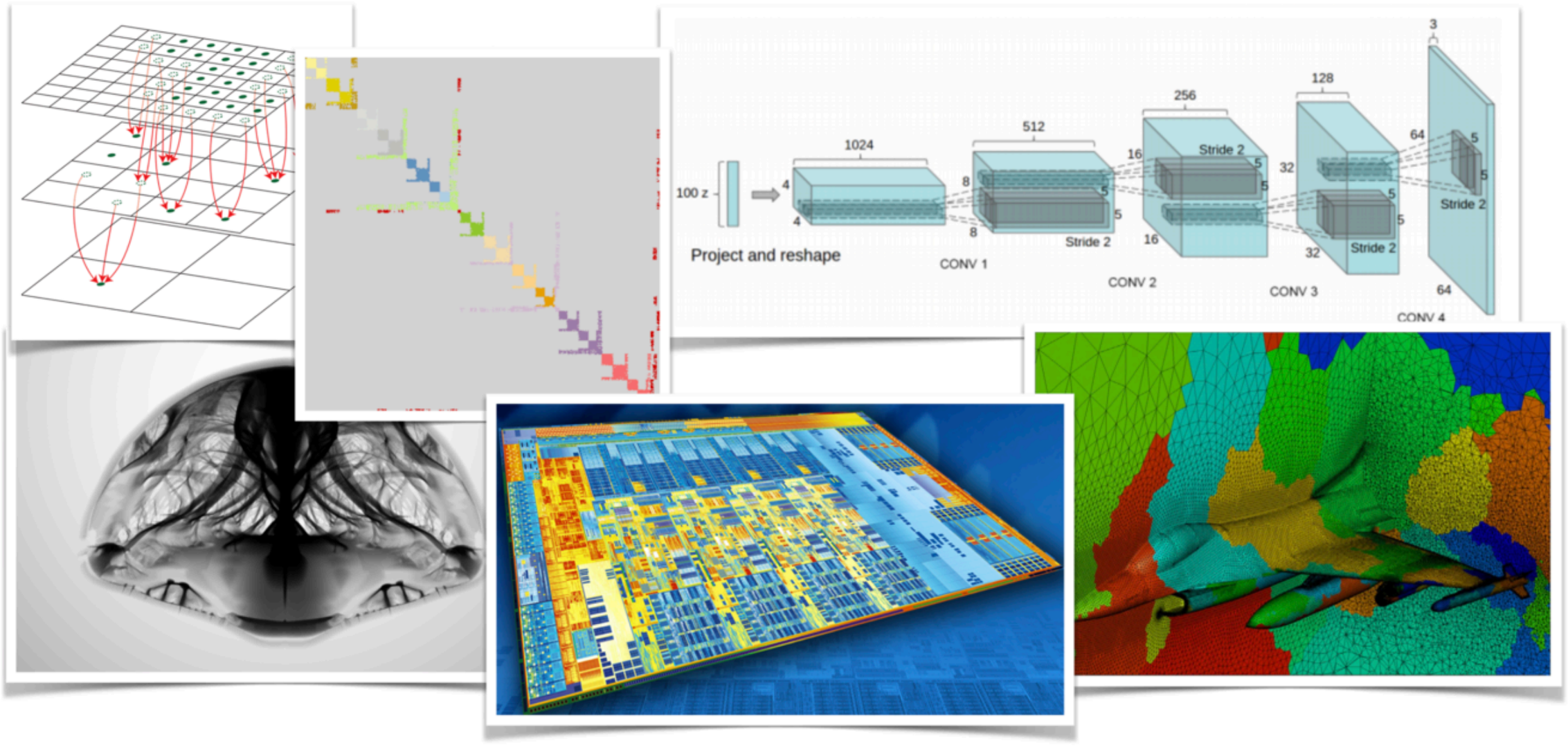


Lecture 3 : More on Grid/Stencil Computations (using the Laplacian example).Tips on benchmarking

Tuesday January 31st 2023

Today's lecture

- Continuing with the Laplacian Stencil example; we will see more issues that influence parallelization efficiency.

- Discussion of performance ceilings based on architectural specifications

- First taste of: effects of caching, prefetching, cache line utilization

- Brief introduction to vectorization and instruction-level parallelism

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/

Not constrained by caches unlike stencil that relies on neighboring memory locations in all directions

- 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"

Add is slower because a is being read+written

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];

}
```

Assume caching on subsequent access for simplification

"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];

}
```

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

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

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];

}
```

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

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

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

This is because of reduced OPs/byte when using fewer cores

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 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]);
```

For memory bound applications,
the openMP core limit
(OMP_NUM_THREADS)
can be lower
than maximum to reach
peak performance. There is also
the overhead of switching between
cores with more complicated kernels

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_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

Up to 9X Slower!

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 header (Laplacian.h)

LaplacianStencil_0_2

Size reduced 16K -> 4K

```
#pragma once

#define XDIM 4096
#define YDIM 4096

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

Working set fits in the cache

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 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

Up to 38X Slower!

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_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

Up to 80X Slower!

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]
```

Reasons for slowing-down?

- Arrays stored in memory according to order:
  a[0][0], a[0][1], … a[0][N-1], a[1][0], a[1][1], a[1][2], …
  `row major by default`
- Data (2GB) too large to fit in cache (~25MB) … needs to be brought in from Main Memory
- When data is fetched from RAM -> Cache, a minimum unit of transfer is a *cache line (64bytes; 16floats).*
- If the loop order follows the shape of the cache lines (i.e. aligned with the 2nd array dimension), the entire cache line is soon reused

Reasons for slowing-down?

- If the loop order follows the shape of the cache lines (i.e. aligned with the 2nd array dimension), the entire cache line is soon reused

- If the loop order is **not** aligned with the orientation of the cache lines, we waste performance in at least 2 ways:

- Cache lines are brought in, but only partly utilized (wasted memory bandwidth)

  This can be avoided by instrumenting code to measure effective bandwidth instead of using hardware monitors of used bandwidth

- Cache runs out of capacity sooner (which is why things get worse with larger arrays)

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”

reinterpret_cast (in original implementation)
allows using [] with pointer to pointer

Kernel header (Laplacian.h)

LaplacianStencil_0_7

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);
```

Execution:

```
Running test iteration  1 [Elapsed time : 20.1705ms]
Running test iteration  2 [Elapsed time : 1.51735ms]
Running test iteration  3 [Elapsed time : 1.51338ms]
Running test iteration  4 [Elapsed time : 0.668702ms]
Running test iteration  5 [Elapsed time : 0.621804ms]
Running test iteration  6 [Elapsed time : 0.62804ms]
Running test iteration  7 [Elapsed time : 0.623426ms]
Running test iteration  8 [Elapsed time : 0.623373ms]
Running test iteration  9 [Elapsed time : 0.624101ms]
Running test iteration 10 [Elapsed time : 0.61673ms]
```

Kernel header (Laplacian.h)

LaplacianStencil_0_8

Rectangular size, 16K x 256
(same overall size as 2K x 2K)

```
#pragma once

#define XDIM 16384
#define YDIM 256

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

Execution:

```
Running test iteration  1 [Elapsed time : 19.4975ms]
Running test iteration  2 [Elapsed time : 0.695738ms]
Running test iteration  3 [Elapsed time : 0.692519ms]
Running test iteration  4 [Elapsed time : 0.692588ms]
Running test iteration  5 [Elapsed time : 0.693134ms]
Running test iteration  6 [Elapsed time : 0.752835ms]
Running test iteration  7 [Elapsed time : 0.348585ms]
Running test iteration  8 [Elapsed time : 0.299074ms]
Running test iteration  9 [Elapsed time : 0.32255ms]
Running test iteration 10 [Elapsed time : 0.299462ms]
```

Benchmark launcher (main.cpp)

LaplacianStencil_0_9

Rows are not in consecutive memory locations

Arrays (u,Lu) allocated as
"arrays of pointers to allocated arrays"
(and allocation randomized)

```
#include "Timer.h"
#include "Laplacian.h"
#include <iomanip>
#include <random>

int main(int argc, char *argv[])
{
    float **u = new float *[XDIM];
    float **Lu = new float *[XDIM];

    // Randomize allocation of minor array dimension
    std::vector<int> reorderMap;
    std::vector<int> tempMap;
    for (int i = 0; i < XDIM; i++) tempMap.push_back(i);
    std::random_device r; std::default_random_engine e(r());
    while (!tempMap.empty()) {
        std::uniform_int_distribution<int> uniform_dist(0, tempMap.size()-1);
        int j = uniform_dist(e);
        reorderMap.push_back(tempMap[j]); tempMap[j] = tempMap.back(); tempMap.pop_back(); }

    for (int i = 0; i < XDIM; i++){
        u[reorderMap[i]] = new float [YDIM];
        Lu[reorderMap[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;
}
```

Kernel header (Laplacian.h)

LaplacianStencil_0_9

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

```
#pragma once

#define XDIM 16384
#define YDIM 256

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

Execution:

```
Running test iteration  1 [Elapsed time : 10.0235ms]
Running test iteration  2 [Elapsed time : 0.750141ms]
Running test iteration  3 [Elapsed time : 0.725621ms]
Running test iteration  4 [Elapsed time : 0.830286ms]
Running test iteration  5 [Elapsed time : 0.801024ms]
Running test iteration  6 [Elapsed time : 0.78661ms]
Running test iteration  7 [Elapsed time : 0.714213ms]
Running test iteration  8 [Elapsed time : 0.71165ms]
Running test iteration  9 [Elapsed time : 0.713606ms]
Running test iteration 10 [Elapsed time : 0.771579ms]
```

Possible culprits?

- *Latency* in fetching a cache line from RAM->Cache is significantly higher than the cost of bringing in a cache line in a *continuous stream* (by 1-2 orders of magnitude)

- We rely on *prefetching* to pick up access patterns, and preemptively bring likely-to-be-used data into cache. (Prefetching is done transparently by the CPU/MMU, or with compiler/programmer assistance and hints)

- Non-sequentially allocated memory compromises the ability of the prefetcher to look ahead and make sure data is brought into cache early enough.

(Check whiteboard notes for further elaboration …)

Compute bounds reached?

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];

}
```

The memory indices get translated to offsets in the compiled code

Every iteration does about 6 (float) arithmetic operations (some more for index algebra, maybe)

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]
```

Compute bounds reached?

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];

}
```

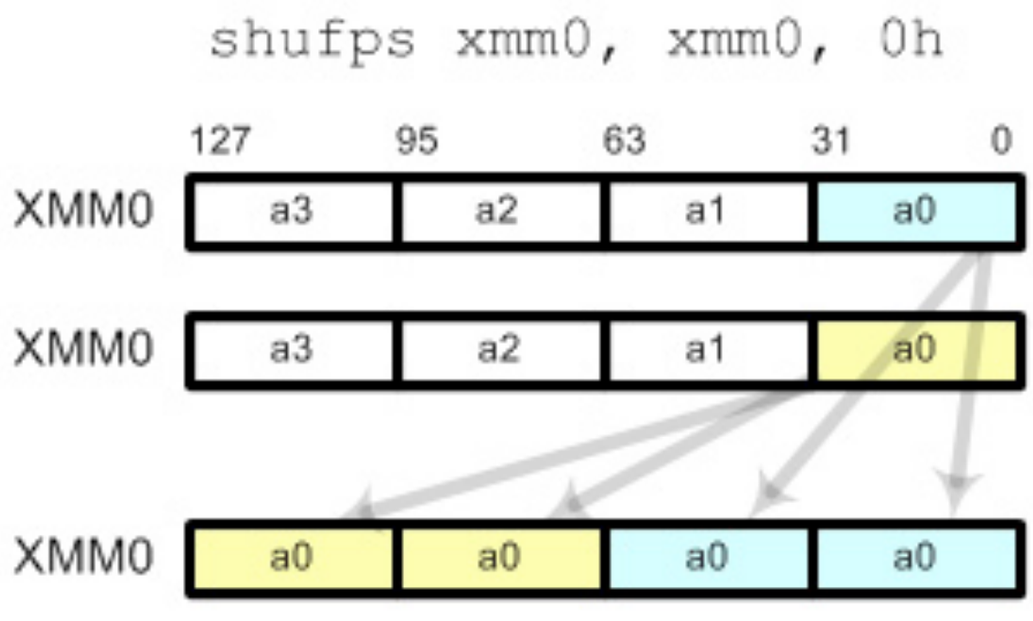
Even at 20ops/iterations, we would be performing about 5Gops/sec (we stated availability of 2.7TFLOPS!)

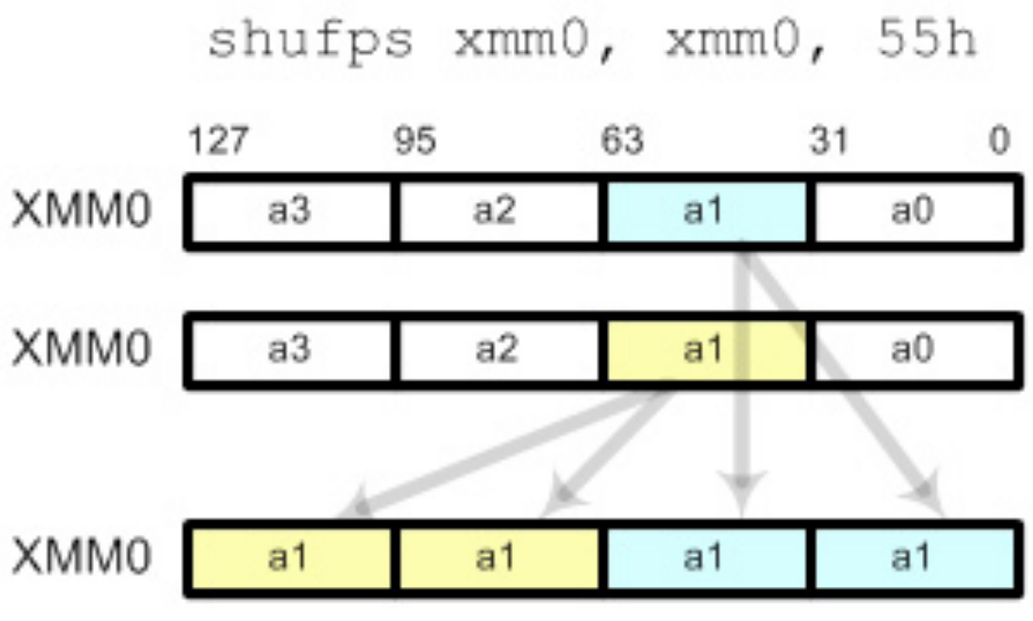
Execution:

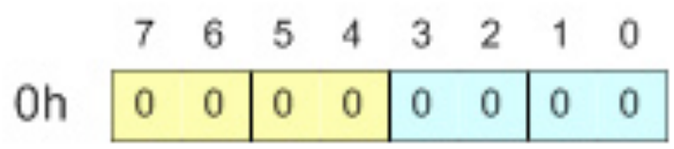
```
Running test iteration  1 [Elapsed time : 120.472ms]
Running test iteration  2 [Elapsed time : 25.3752ms]
Running test iteration  3 [Elapsed time : 24.3025ms]
Running test iteration  4 [Elapsed time : 23.0271ms]
Running test iteration  5 [Elapsed time : 22.6208ms]
Running test iteration  6 [Elapsed 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]
```

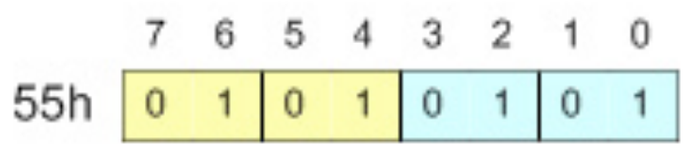
Peak compute performance?

- Peak compute performance presumes full utilization of SIMD Processing or *vectorization*
- The compiler often helps with that, but not at all a trivial proposition!



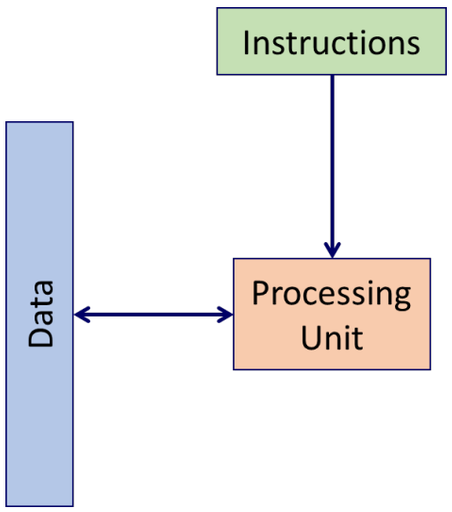




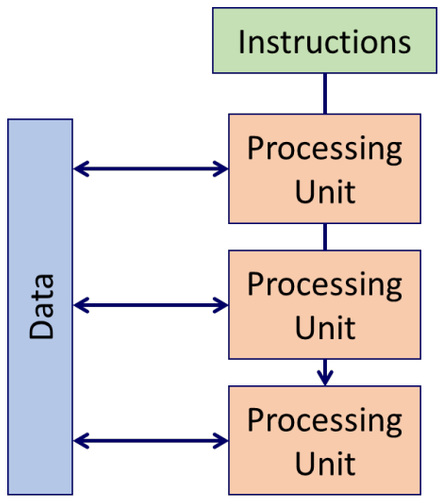


Practical use of SIMD in code

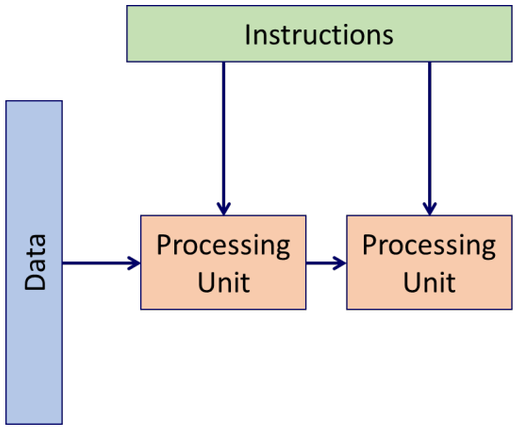
		Data Stream	
		Single	Multi
Instruction Stream	Single	SISD (Single-Core Processors)	SIMD (GPUs, Intel SSE/AVX extensions, …)
	Multi	MISD (Systolic Arrays, …)	MIMD (VLIW, Parallel Computers)



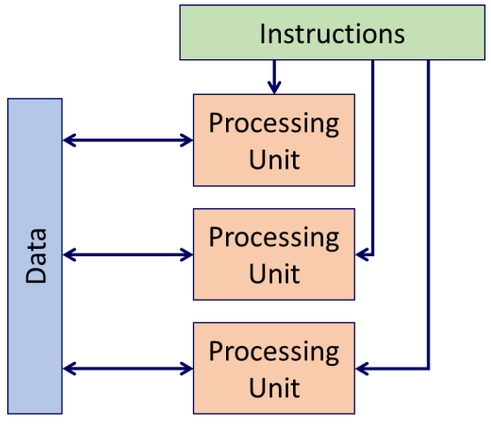
Single-Instruction
Single-Data
(Single-Core Processors)



Single-Instruction
Multi-Data
(GPUs, Intel SIMD)

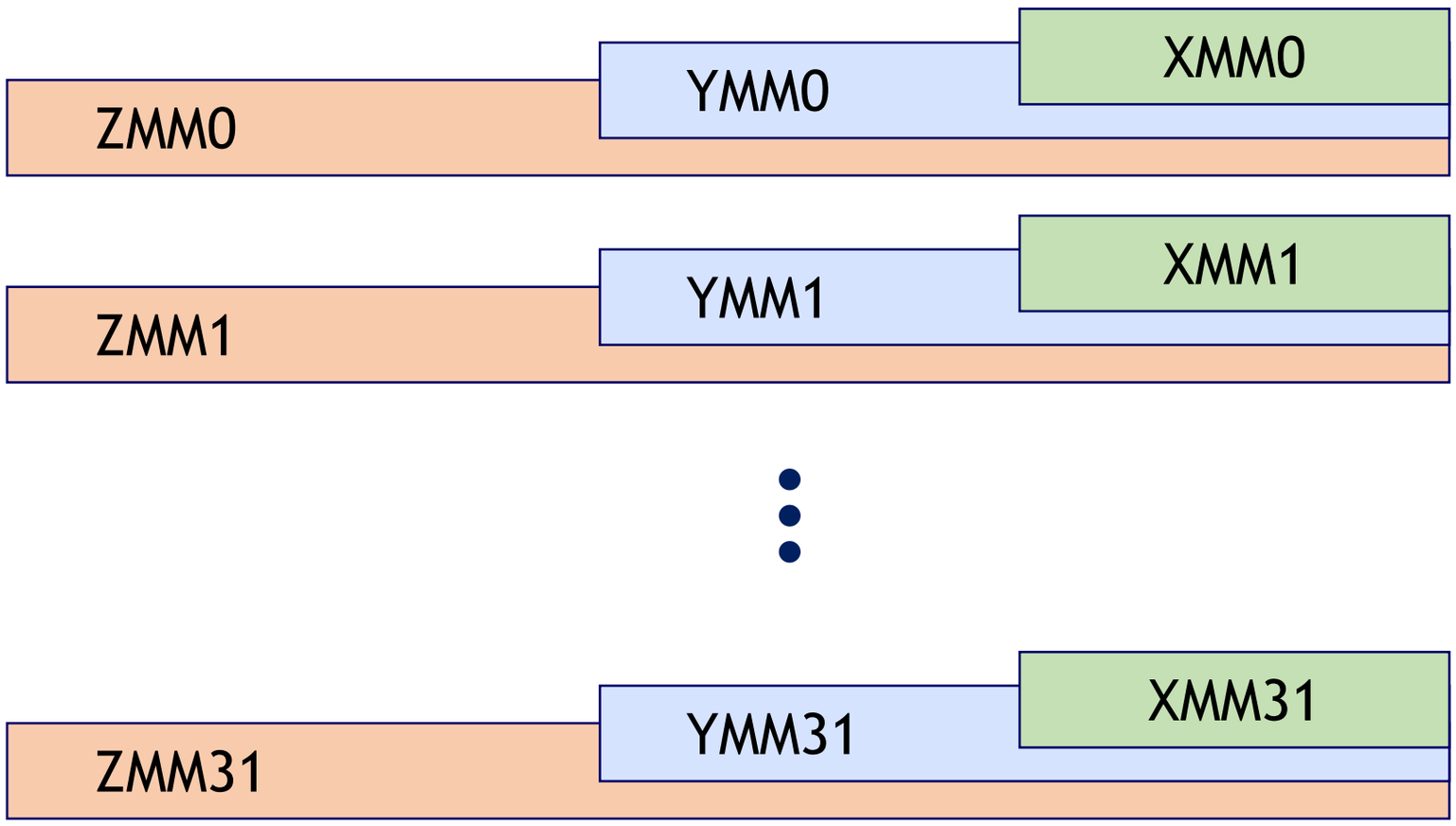


Multi-Instruction
Single-Data
(Systolic Arrays,…)



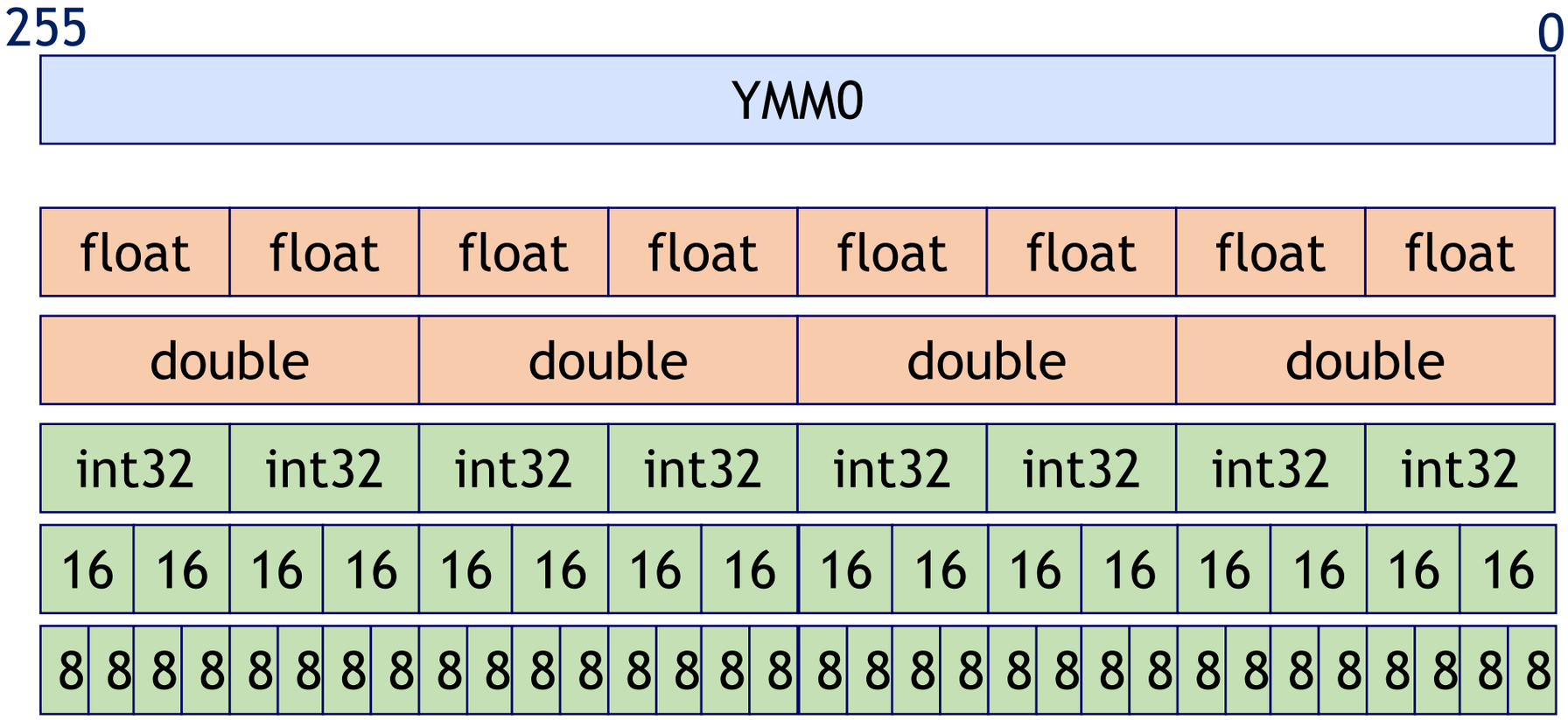
Multi-Instruction
Multi-Data
(Parallel Computers)

Intel SIMD Registers (AVX-512)



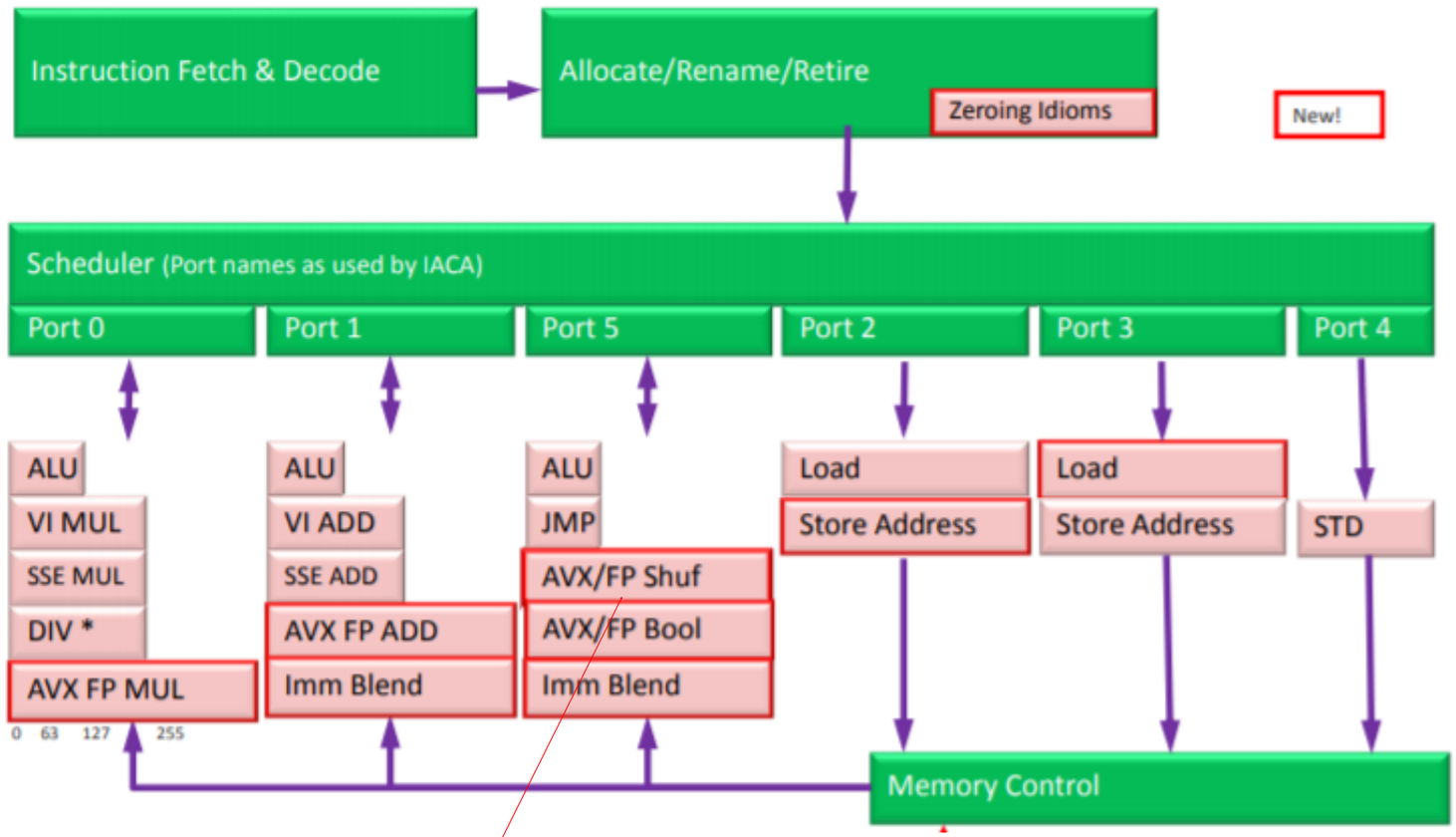
- XMM0 – XMM15
  - 128-bit registers
  - SSE
- YMM0 – YMM15
  - 256-bit registers
  - AVX, AVX2
- ZMM0 – ZMM31
  - 512-bit registers
  - AVX-512

SSE/AVX Data Types



Operation on 32 8-bit values in one instruction!

Sandy Bridge Microarchitecture



e.g., “Port 5 pressure” when code uses too much shuffle operations

Example 4-13. Simple Four-Iteration Loop

```
void add(float *a, float *b, float *c)
{
int i;
for (i = 0; i < 4; i++) {
    c[i] = a[i] + b[i];
  }
}
```

Example 4-14. Streaming SIMD Extensions Using Inlined Assembly Encoding

```
void add(float *a, float *b, float *c)
{
  __asm {
    mov     eax, a
    mov     edx, b
    mov     ecx, c
    movaps  xmm0, XMMWORD PTR [eax]
    addps   xmm0, XMMWORD PTR [edx]
    movaps  XMMWORD PTR [ecx], xmm0
  }
}
```

Example 4-14. Streaming SIMD Extensions Using Inlined Assembly Encoding

```
void add(float *a, float *b, float *c)
{
  __asm {
    mov     eax, a
    mov     edx, b
    mov     ecx, c
    movaps  xmm0, XMMWORD PTR [eax]
    addps   xmm0, XMMWORD PTR [edx]
    movaps  XMMWORD PTR [ecx], xmm0
  }
}
```

✓ Anything that ***can*** be done, can be coded up as inline assembly

✓ Maximum ***potential*** for performance accelerations

✓ Direct control over the code being generated

Example 4-14. Streaming SIMD Extensions Using Inlined Assembly Encoding

```
void add(float *a, float *b, float *c)
{
  __asm {
    mov     eax, a
    mov     edx, b
    mov     ecx, c
    movaps  xmm0, XMMWORD PTR [eax]
    addps   xmm0, XMMWORD PTR [edx]
    movaps  XMMWORD PTR [ecx], xmm0
  }
}
```

✓ Anything that ***can*** be done, can be coded up as inline assembly

✓ Maximum ***potential*** for performance accelerations

✓ Direct control over the code being generated

✗ Impractical for all but the smallest of kernels

✗ Not portable

✗ User needs to perform register allocation (and save old registers)

✗ User needs to (expertly) schedule instructions to hide latencies

Intrinsics

- A framework for generating assembly-level code without many of the drawbacks of inline assembly
  - Compiler (not programmer) takes care of register allocation
  - Compiler is able to schedule instructions to hide latencies
- Data types
  - Scalar : `float, double, unsigned int ...`
  - Vector : `__mm128, __m128d, __m256, __m256i ...`
- Intrinsic functions
  - Instruction wrappers : `_mm_add_pd, _mm256_mult_pd, _mm_xor_ps, _mm_sub_ss ...`
  - Macros : `_mm_set1_ps, _mm256_setzero_ps ...`
  - Math Wrappers : `_mm_log_ps, _mm256_pow_pd ...`

Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

Example 4-14. Streaming SIMD Extensions Using Inlined Assembly Encoding

```
void add(float *a, float *b, float *c)
{
  __asm {
    mov     eax, a
    mov     edx, b
    mov     ecx, c
    movaps  xmm0, XMMWORD PTR [eax]
    addps   xmm0, XMMWORD PTR [edx]
    movaps  XMMWORD PTR [ecx], xmm0
  }
}
```

Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

✓ Almost as flexible as inline assembly

✓ Somewhat portable

✓ Compiler takes care of register allocation (and spill, if needed)

✓ Compiler will shuffle & schedule instructions to best hide latencies

✓ Relatively easy migration from SSE -> AVX -> MIC ....

Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

✓ Almost as flexible as inline assembly

✓ Somewhat portable

✓ Compiler takes care of register allocation (and spill, if needed)

✓ Compiler will shuffle & schedule instructions to best hide latencies

✓ Relatively easy migration from SSE -> AVX
SSE -> AVX -> MIC ….

✗ Coding large kernels is still challenging and bug-prone

✗ Un-natural notation (vs. C++ expressions and operators)

✗ SSE code is ***similar*** to AVX code, but different enough so that 2 distinct versions must be written

✗ Vector code looks very different than scalar code

Example 4-17. Automatic Vectorization for a Simple Loop

```
void add (float *restrict a,
                float *restrict b,
                float *restrict c)
{
        int i;
        for (i = 0; i < 4; i++) {
                c[i] = a[i] + b[i];
        }
}
```

Example 4-17. Automatic Vectorization for a Simple Loop

```
void add (float *restrict a,
          float *restrict b,
          float *restrict c)
{
    int i;
    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```

✓Minimal effort required (assuming it works ...)

✓Development of SIMD code is no different than scalar code

✓Ability to use complex C++ expressions

✓Larger kernels are easier to tackle

Example 4-17. Automatic Vectorization for a Simple Loop

```
void add (float *restrict a,
          float *restrict b,
          float *restrict c)
{
    int i;
    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```

✓ Minimal effort required (assuming it works ...)

✓ Development of SIMD code is no different than scalar code

✓ Ability to use complex C++ expressions

✓ Larger kernels are easier to tackle

✗ In practice it can be ***very*** challenging to achieve efficiency comparable to assembly/intrinsics

✗ Compilers are ***very*** conservative when vectorizing, for the risk of jeopardizing scalar equivalence

✗ The no-aliasing restriction might run contrary to the spirit of certain kernels

Example 4-16. C++ Code Using the Vector Classes

```
#include <fvec.h>
void add(float *a, float *b, float *c)
{
        F32vec4 *av=(F32vec4 *) a;
        F32vec4 *bv=(F32vec4 *) b;
        F32vec4 *cv=(F32vec4 *) c;
                *cv=*av + *bv;
}
```

Example 4-16. C++ Code Using the Vector Classes

```
#include <fvec.h>
void add(float *a, float *b, float *c)
{
        F32vec4 *av=(F32vec4 *) a;
        F32vec4 *bv=(F32vec4 *) b;
        F32vec4 *cv=(F32vec4 *) c;
                *cv=*av + *bv;
}
```

✓Fewer visual differences between vector and scalar code

✓Ability to use complex C++ expressions (assuming wrapper types have been overloaded)

✓Easy transition to different vector widths

Example 4-16. C++ Code Using the Vector Classes

```
#include <fvec.h>
void add(float *a, float *b, float *c)
{
        F32vec4 *av=(F32vec4 *) a;
        F32vec4 *bv=(F32vec4 *) b;
        F32vec4 *cv=(F32vec4 *) c;
                *cv=*av + *bv;
}
```

✓Fewer visual differences between vector and scalar code

✓Ability to use complex C++ expressions (assuming wrapper types have been overloaded)

✓Easy transition to different vector widths

✗ Heavy dependence on the compiler for eliminating temporaries (but it typically does a really good job at it)

✗ Limited to the semantics of the built-in vector wrapper classes (but we are free to extend those)

✗ Risk of more bloated executable code than by using intrinsics