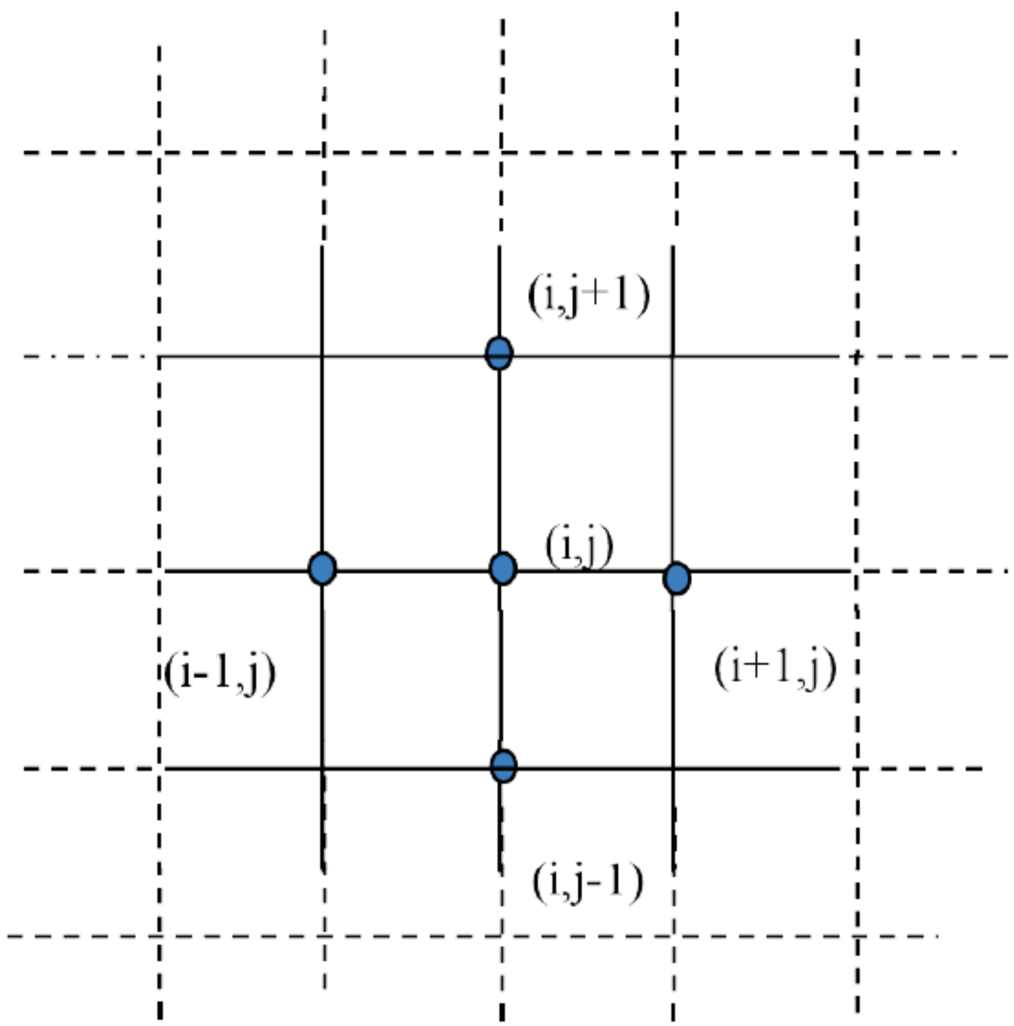
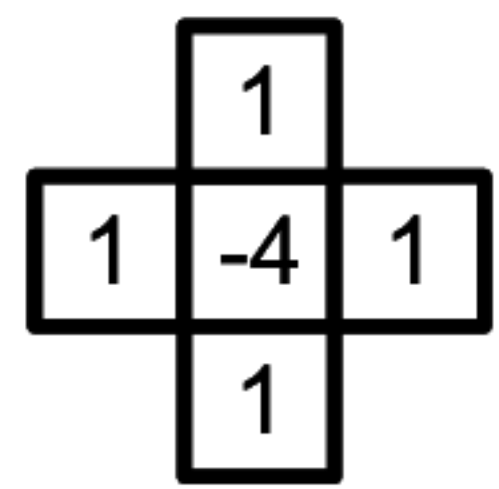
# Case study #0 (part II)

Laplacian Stencil Application
(Today : on 2D grid)



# 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_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_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

```
#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;
}
```

*Arrays (u,Lu) allocated as “arrays of pointers to allocated arrays”* ***(and allocation randomized)***

# Kernel header (Laplacian.h)

*LaplacianStencil_0_9*

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

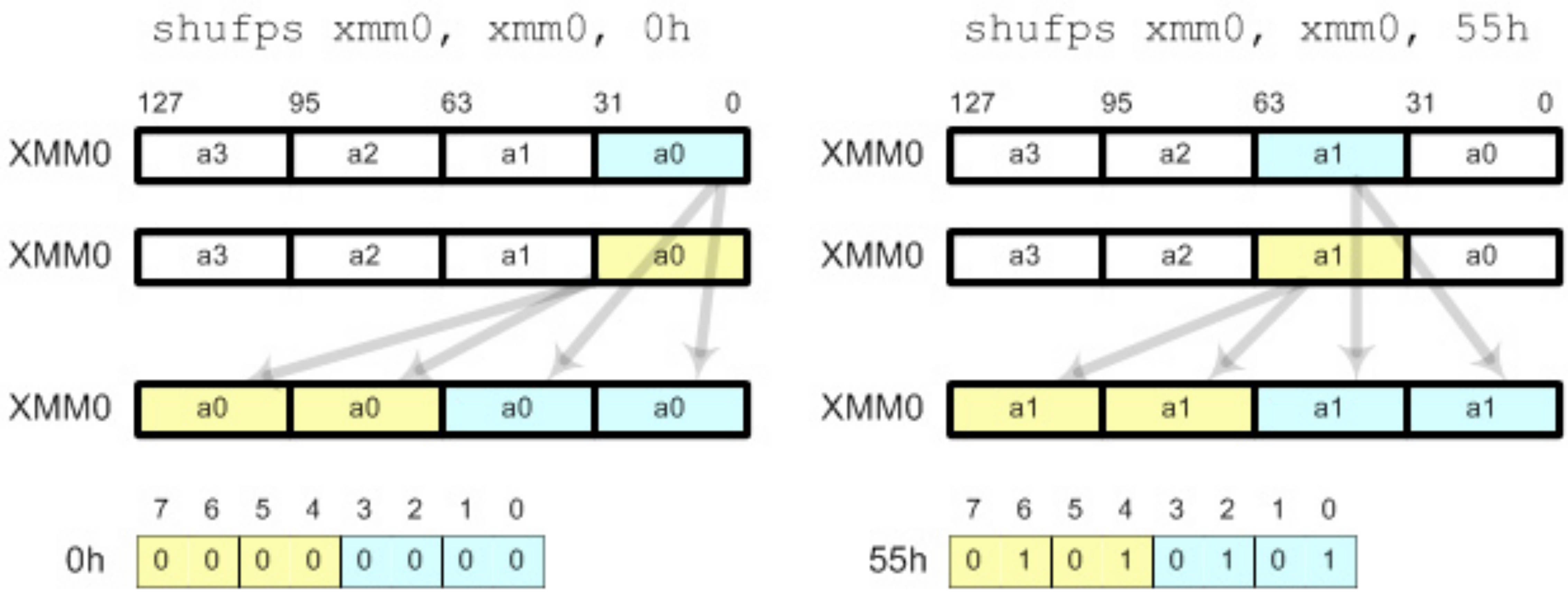
```
#pragma once

#define XDIM 16384
#define YDIM 256

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

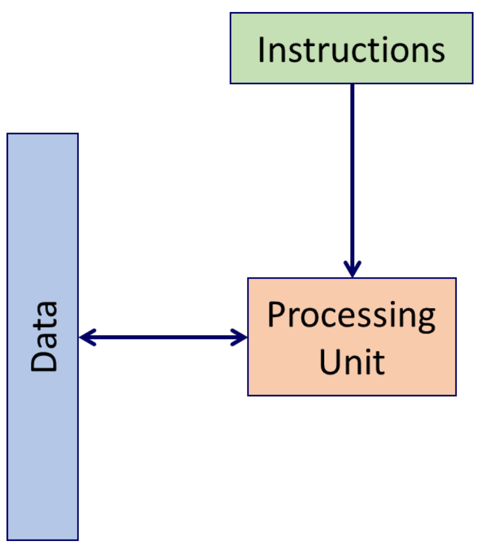
**Execution:**

```
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]
```

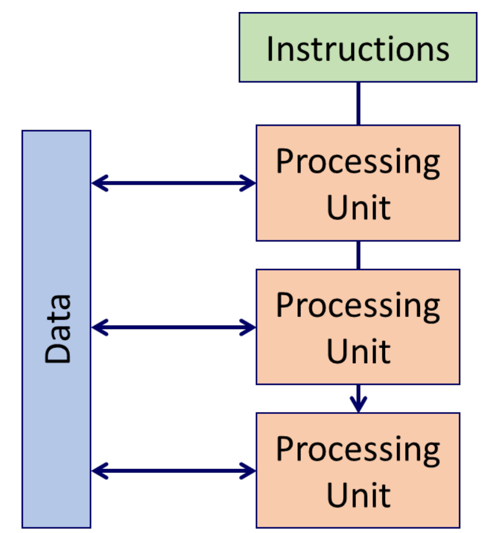



# Practical use of SIMD in code

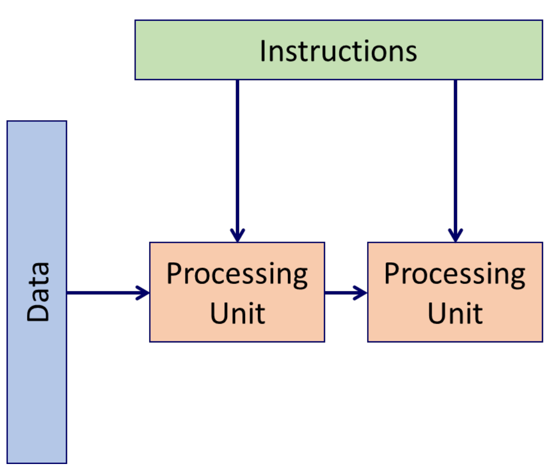
| | | Data Stream | |
|---|---|---|---|
| | | Single | Multi |
| Instruction Stream | Single | SISD (Single-Core Processors) | SIMD (GPUs, Intel SSE/AVX extensions, ...) |
| | Multi | MISD (Systolic Arrays, ...) | MIMD (VLIW, Parallel Computers) |



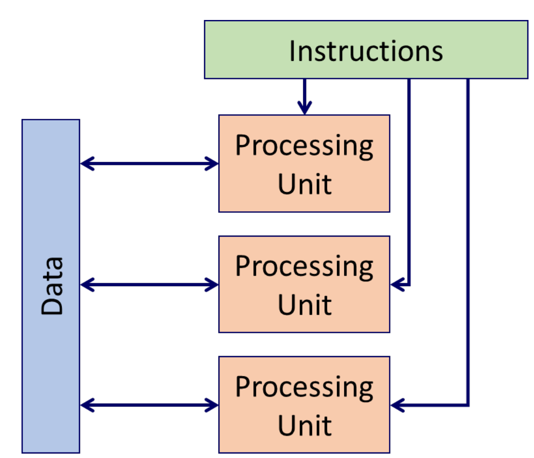

Single-Instruction
Single-Data
(Single-Core Processors)




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

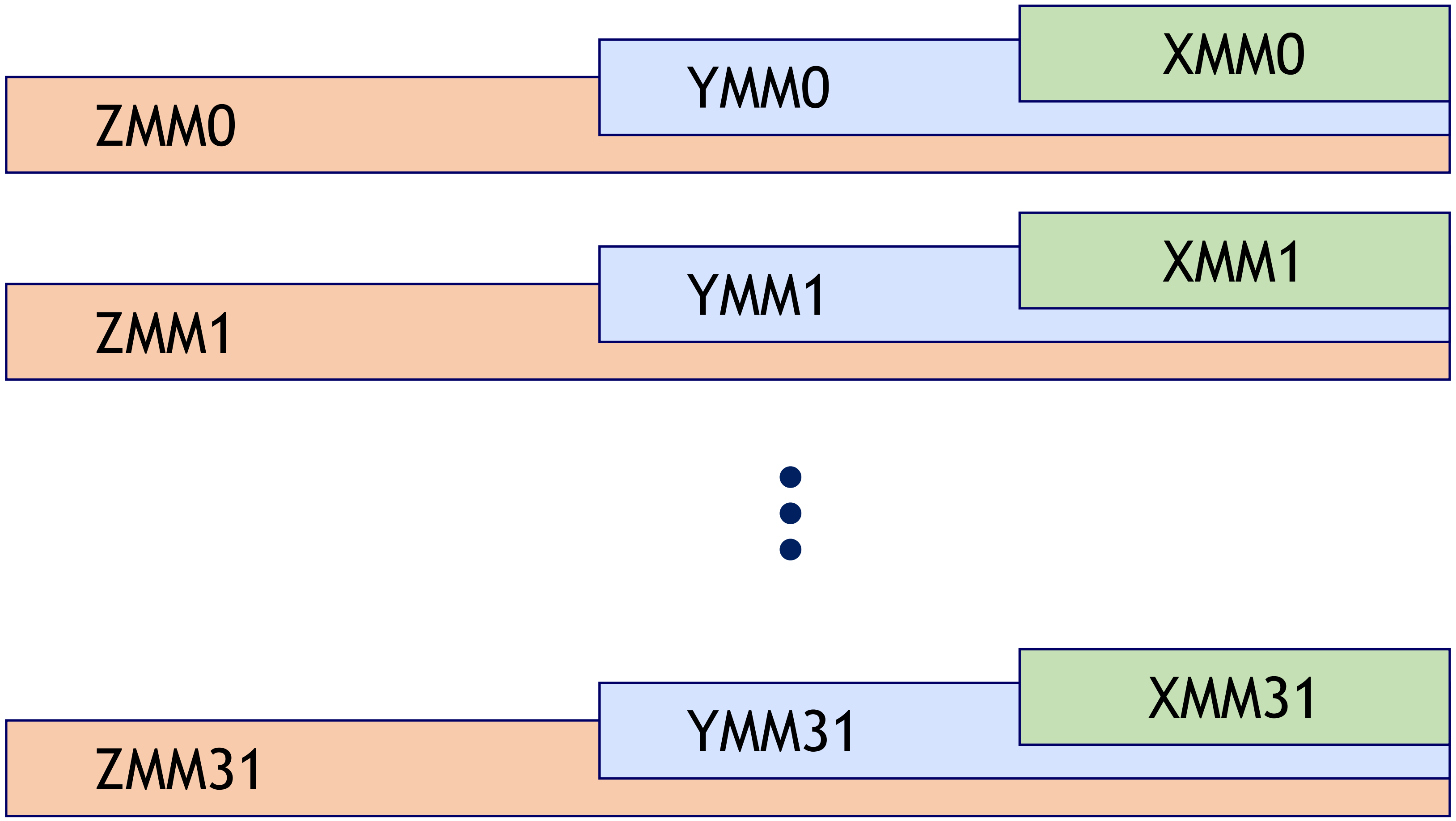



Multi-Instruction
Single-Data
(Systolic Arrays,…)



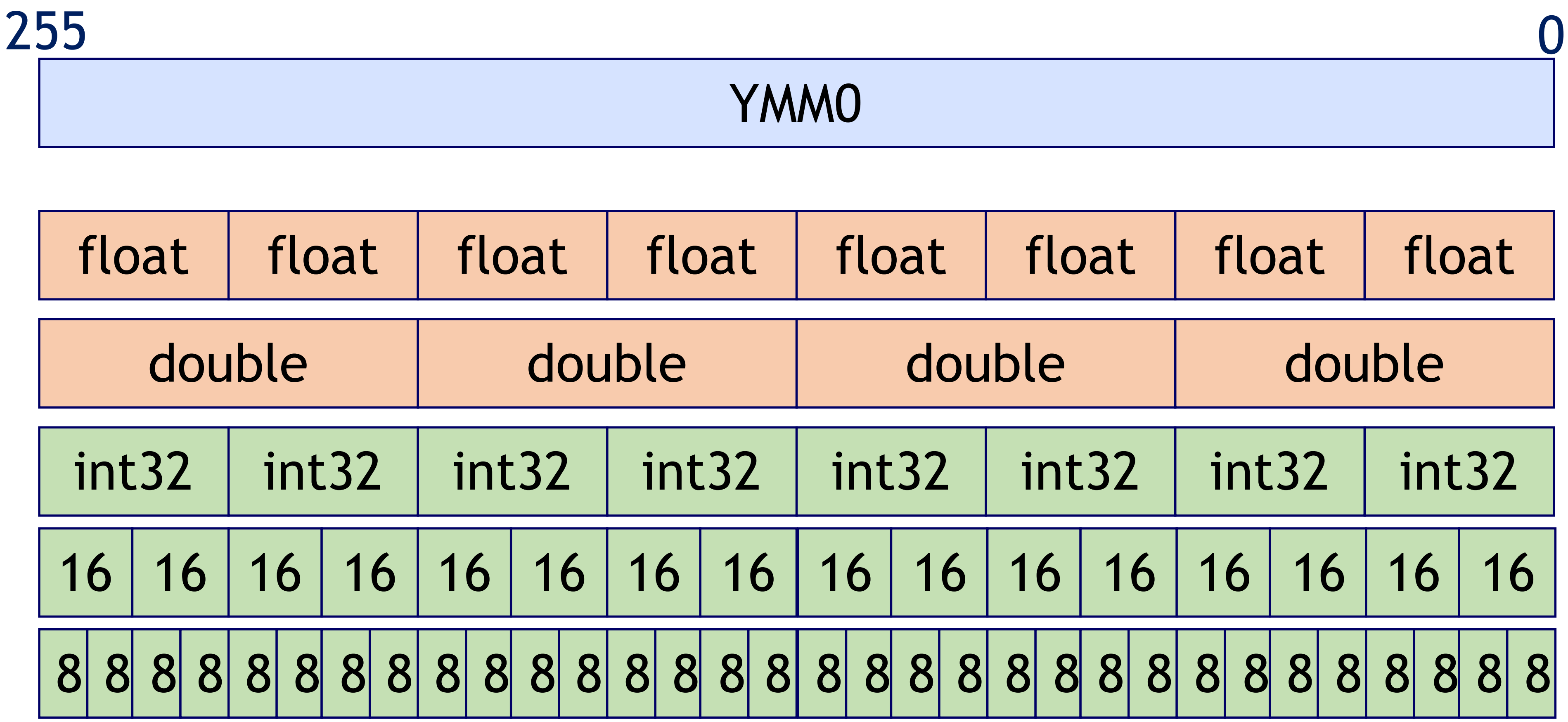

Multi-Instruction
Multi-Data
(Parallel Computers)

# Intel SIMD Registers (AVX-512)



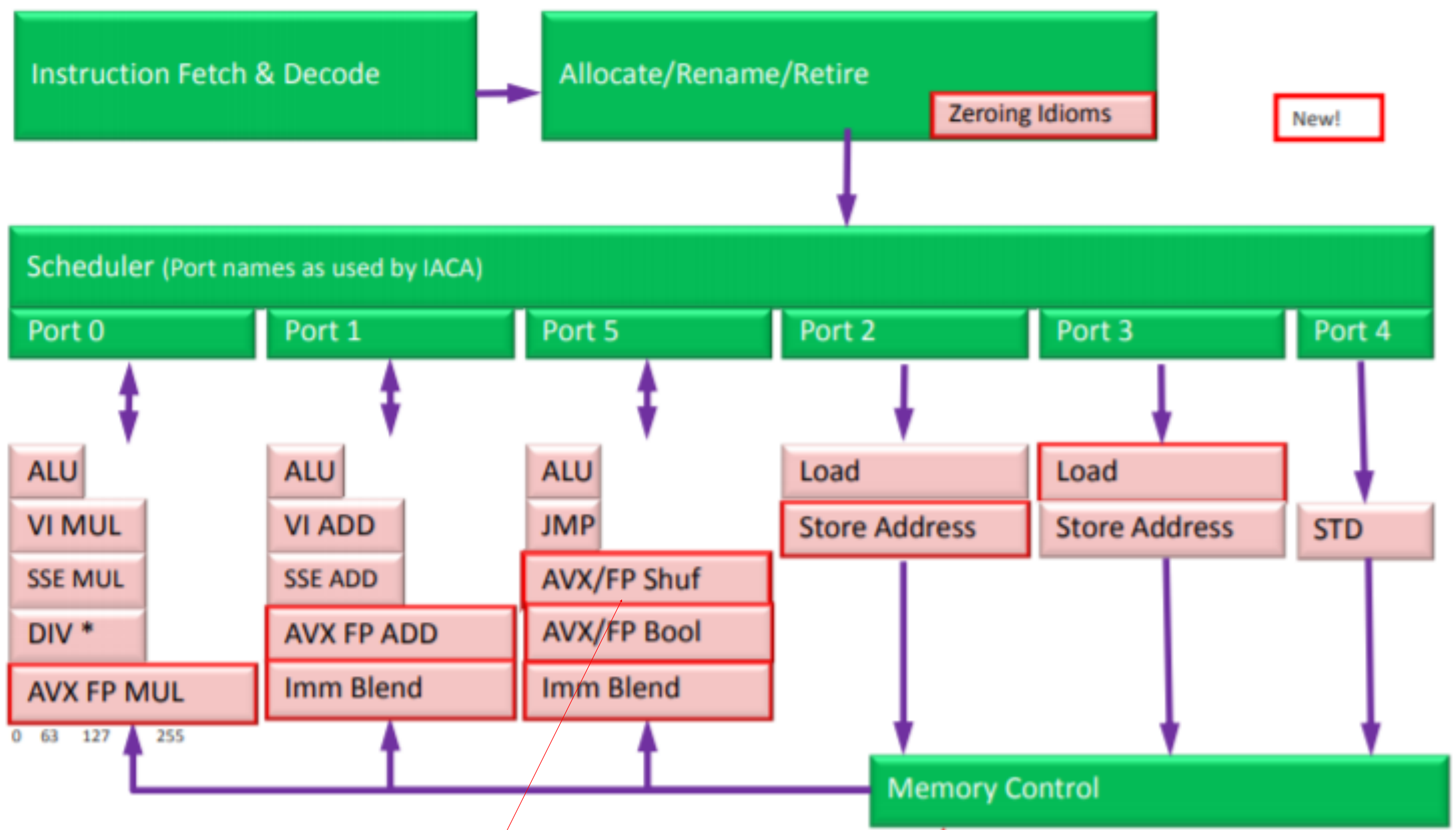

- XMM0 – XMM15
  - 128-bit registers
  - SSE
- YMM0 – YMM15
  - 256-bit registers
  - AVX, AVX2
- ZMM0 – ZMM31
  - 512-bit registers
  - AVX-512

# 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

**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

✗ 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