HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 18

Fabian Wermelinger
Harvard University
CS205

Thursday, April 6th 2023

LAST TIME

- Basics of assembly language.
- Differences between `IA32` (32-bit) and `x86-64` ISA's.
- General purpose registers.
- `x86-64` operand forms and addressing modes.
- Procedure calls and argument passing
- `x86-64` floating-point operations.

TODAY

Main topic: ***SIMD, auto-vectorization and data layouts***

Details: (*Disclaimer:* this lecture is dense, we will use two meetings.)

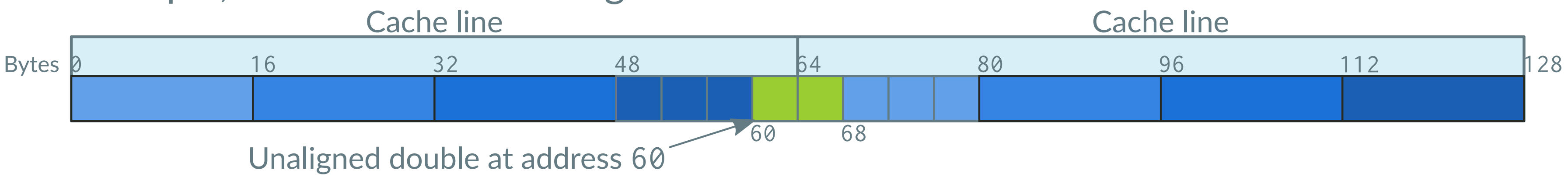
- Data alignment and how to align memory in code.
- Data layouts concerned with SIMD (data-level parallelism)
- Aside: aliasing (and its effects on auto-vectorization).
- Single-Instruction-Multiple-Data (SIMD):
  - Compiler auto-vectorization.
  - SIMD support in OpenMP
  - Intel intrinsics (x86-64 specific).
  - Manual vectorization of code (SIMD instructions).
  - Related examples.

AGENDA CHECK:

- Project design presentations due next week. Initial results for a sequential baseline code should be presented which can then be parallelized.

DATA ALIGNMENT

- ***Recall:*** lecture 16 → interpretation of memory addresses. We talked about ***byte order*** (endianness) and ***byte alignment***.
- Many computer systems have *alignment restrictions* to *simplify* the design of the hardware interface between the processor and the memory system.
- These alignment restrictions depend on the data type and are typically 2, 4, 8, 16 or 32 byte boundaries.
- For example, consider the following situation:



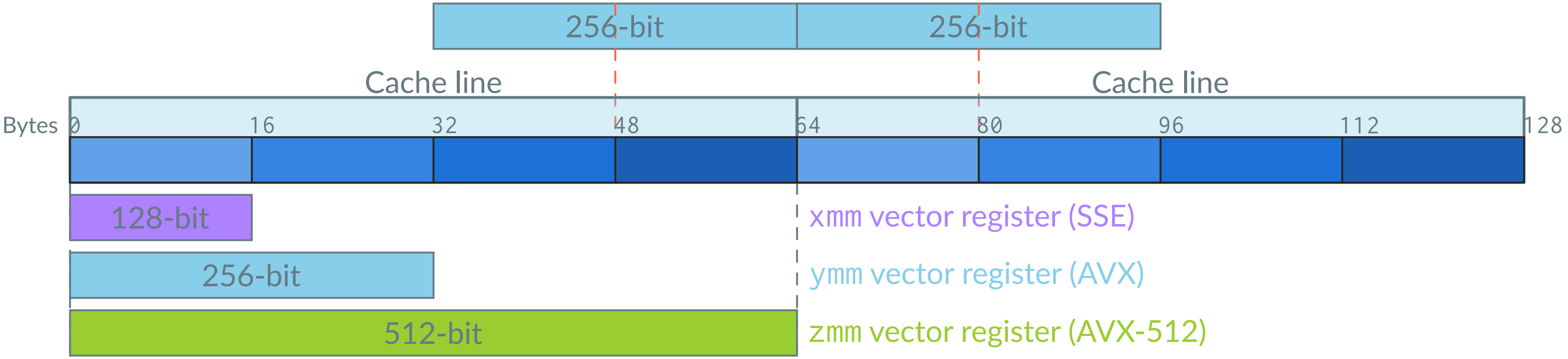
  A `double` type at byte address `60` is *unaligned* and would require *two memory operations* to load or store. A `double` is 8 bytes and would require a 8 byte alignment, i.e., it would be aligned at address `56` or `64`. ***The compiler takes care of this for built-in types***.

- `x86-64` hardware *will work correctly* regardless of data alignment. It is however recommended that data be aligned to improve the memory system performance. This ***does not*** mean that you must vigorously enforce alignment *everywhere*. You should think about alignment for the *performance critical data structures*, e.g., computational grid or particle vectors.

DATA ALIGNMENT

- If you use SSE instructions then the data *must* be aligned at *16 byte* boundaries because the `xmm` registers are 128-bit wide vectors. The memory move operations in SSE require this alignment. *Any operation that works with data that is not properly aligned will lead to a memory exception and crash.*
- *The AVX extensions do not have this strict requirement*. It is nevertheless recommended to align data at *16 byte* boundaries for data move instructions that take `xmm` operands (SSE, 128-bit) or *32 byte* boundaries for instructions that take `ymm` operands (AVX, 256-bit).
- Implications of memory alignment:
  - The starting address of any block generated by memory allocators must be a multiple of 16 or 32, respectively.
  - The *stack frame* for most functions must be aligned on 16 or 32 byte boundaries, respectively.
- *Most AVX unaligned memory moves (today) have similar performance to aligned memory moves.* A penalty occurs when unaligned memory moves involve two cache lines:

Example: Data aligned at 16 byte boundaries may result in this situation when working with `ymm` registers and will impact performance of load/store operations.



HOW TO ALIGN MEMORY?

- There are different possibilities to align memory and they differ whether you use C or C++.
- In C:
  - See https://www.gnu.org/software/libc/manual/html_node/Aligned-Memory-Blocks.html
  - Since the C11 standard `_Alignas` or `alignas` can be used. You must include the `stdalign.h` header for the latter. Similarly, use `_Alignof` and `alignof` for the alignment requirement of a type.

Example 1: type alignment

```c
#include <stdio.h>
int main(void)
{
    printf("Byte alignment of double is: %zu\n",
        _Alignof(double));
    _Alignas(8) double d; // _Alignas(8) is already the minimum!
    printf("Address of d %  8 = %zu\n", ((size_t)&d) % 8);
    return 0;
}
```

Example 2: dynamic memory alignment

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    double *pd;
    posix_memalign((void **)&pd, 8, // segfault if alignment <8
        sizeof(double));
    printf("Address of pd %  8 = %zu\n", ((size_t)pd) % 8);
    free(pd);
    return 0;
}
```

- In C++:
  - You can use the same allocators as for C.
  - C++11 comes with built-in alignment specifier similar to C11. Use `alignas` to specify a type alignment and `alignof` to query an alignment requirement of a type.

ASIDE: ALIASING

Consider the function:

```c
void f(double *a, double *b) {
    // do some work with data a and b
}
```

If you were the compiler, what would you be concerned about?

- The pointers a and b may point to *overlapping* memory. If this is the case we say pointers a and b *alias* each other.
- If pointers alias, the compiler *cannot* reorder write operations. They must maintain *program order* in such a case.
- If you were a C/C++ compiler then you should worry about this. If you were a Fortran compiler you would not care at all → in Fortran aliasing is *illegal* (two memory regions cannot overlap).
- This is often the reason *some people claim Fortran is faster than C*, which is true if you don't pay attention to aliasing.

ASIDE: ALIASING

Consider the function:

```
void f(double *a, double *b) {
    // do some work with data a and b
}
```

Ways to support the compiler:

- ***Command line options*** (weakest form): see `man gcc`
  - `-fstrict-aliasing` this is implied with `-O2`, `-O3` and `-Os`.
  - `-Wstrict-aliasing[=level]` enable warnings when aliasing rules are broken. `-fstrict-aliasing` implies `-Wstrict-aliasing=3`, see `man gcc`.
- ***Strict aliasing rule made by C/C++ compiler:*** dereferencing pointers to objects with *different types will never* refer to the same memory location. Undefined behavior otherwise!
- ***Be precise when you code:*** if function parameter are read-only, *tell the compiler!*

```
void f(const double *a, const double *b) {
    // pointers a and b are read-only -> since you cannot write to them, the compiler can optimize!
}
```

- Indicate that pointers do not alias if they point to mutable data:

```
void f(double *__restrict__ a, double *__restrict__ b) {
    // pointers a and b are writable but do not overlap
}
```

ASIDE: ALIASING EXAMPLE I

Consider the two identical functions f and g:

Strict aliasing rule: compiler must assume that pointers a and b *may alias*:

```c
int f(int *a, int *b)
{
    *a = 1;
    *b = 2;
    return *a; // 1 or 2?
}
```

gcc -O3 -S -masm=intel

```asm
f:
    mov DWORD PTR [rdi], 1    ; store 1 in *a
    mov DWORD PTR [rsi], 2    ; store 2 in *b
    mov eax, DWORD PTR [rdi] ; must re-read *a!
    ret
```

```c
int f(int *a, int *b);
int main(void) {
    int a = 0;
    int *b = &a;
    printf("%d\n", f(&a, b));
}
```

Output: 2

Contract: you assure the compiler that pointers a and b *do not alias*:

```c
int g(int *__restrict__ a, int *__restrict__ b)
{
    *a = 1;
    *b = 2;
    return *a; // definitely 1!
}
```

gcc -O3 -S -masm=intel

```asm
g:
    mov DWORD PTR [rdi], 1 ; store 1 in *a
    mov eax, 1 ; no need to access memory again!
    mov DWORD PTR [rsi], 2 ; store 2 in *b
    ret
```

```c
int g(int *a, int *b);
int main(void) {
    int a = 0;
    int *b = &a;
    printf("%d\n", g(&a, b));
}
```

Output: 1

ASIDE: ALIASING EXAMPLE II

Recall the SGEMV kernel from HW2:

```c
void sgemv(const float *A, const float *x, float *y, const size_t n)
{
    for (size_t j = 0; j < n; ++j)
        for (size_t i = 0; i < n; ++i)
            y[j] += A[j * n + i] * x[i]; // what if y is an alias to x?
}
```

Assembly code with strict aliasing:

```c
void sgemv(const float *A, const float *x, float *y, const size_t n)
```

```asm
sgemv:
.L12: ; outer loop
    movss   xmm1, DWORD PTR [rdx] ; load y[j]
.L13: ; inner loop
    movss   xmm0, DWORD PTR [rdi+rax*4] ; load A_ji
    mulss   xmm0, DWORD PTR [rsi+rax*4] ; A_ji * x[i]
    add     rax, 1 ; increment loop index i
    addss   xmm1, xmm0 ; add to y[j]
    movss   DWORD PTR [rdx], xmm1 ; store y[j]!
    cmp     rcx, rax ; check inner loop index
    jne     .L13 ; continue inner loop
    add     rdx, 4 ; advance y by 4 bytes
    add     rdi, r8 ; move to next row in A
    cmp     r9, rdx ; check outer loop index
    jne     .L12 ; continue with outer loop
```

Assembly code with no aliasing:

```c
void sgemv(const float *A, const float *x, float *__restrict__ y, const size_t n)
```

```asm
sgemv:
.L3: ; outer loop
    movss   xmm1, DWORD PTR [rdx] ; load y[j]
.L4: ; inner loop
    movss   xmm0, DWORD PTR [rdi+rax*4] ; load A_ji
    mulss   xmm0, DWORD PTR [rsi+rax*4] ; A_ji * x[i]
    add     rax, 1 ; increment loop index i
    addss   xmm1, xmm0 ; add to y[j]
    cmp     rcx, rax ; check inner loop index
    jne     .L4 ; continue inner loop
    movss   DWORD PTR [rdx], xmm1 ; store y[j] after!
    add     rdx, 4 ; advance y by 4 bytes
    add     rdi, r8 ; move to next row in A
    cmp     r9, rdx ; check outer loop index
    jne     .L3 ; continue with outer loop
```

Without aliasing, the result of the inner-product is stored only ***after*** the inner loop!

ASIDE: ALIASING EXAMPLE II

Checking the load/store counts with PAPI: (problem size $n = 1000$)

Assembly code with strict aliasing:

```c
void sgemv(const float *A, const float *x, float *y, const size_t n)
```

```
Result:                       200004
Total cycles:                 2985979
Total instructions:           7009115
Instructions per cycle (IPC): 2.34734
L1 cache size:                32 KB
L2 cache size:                256 KB
L3 cache size:                40960 KB
Total problem size:           3914 KB
Total L1 data misses:         62823
Total load/store:             3002030
Operational intensity:        1.666372e-01
Performance [Gflop/s]:        9.991571e-01
Wall-time   [micro-seconds]:  2.002688e+03
```

Assembly code with no aliasing:

```c
void sgemv(const float *A, const float *x, float *__restrict__ y, const size_t n)
```

```
Result:                       200004
Total cycles:                 2946528
Total instructions:           6010115
Instructions per cycle (IPC): 2.03973
L1 cache size:                32 KB
L2 cache size:                256 KB
L3 cache size:                40960 KB
Total problem size:           3914 KB
Total L1 data misses:         62805
Total load/store:             2003029
Operational intensity:        2.497468e-01
Performance [Gflop/s]:        1.079912e+00
Wall-time   [micro-seconds]:  1.852928e+03
```

- We expect $2n^2 + 2n$ memory operations (load and store).
- Without aliasing we do meet this expectation.
- The code that follows the strict aliasing rule issues another n^2 writes.
- Fortran assumes memory will never alias and therefore generates code corresponding to the right column above. This code is faster of course.

DATA LAYOUTS

> The efficient exploitation of *data-level parallelism* naturally depends on the *data layout*.
> ***Niklaus Wirth:*** *Algorithms + Data Structures = Programs*

- Not all data structures are equally well suited for vectorizing code.
- Modern software design is often *object oriented*.
- Object oriented design *does not imply* well structured data (for SIMD).
- We can distinguish the following:

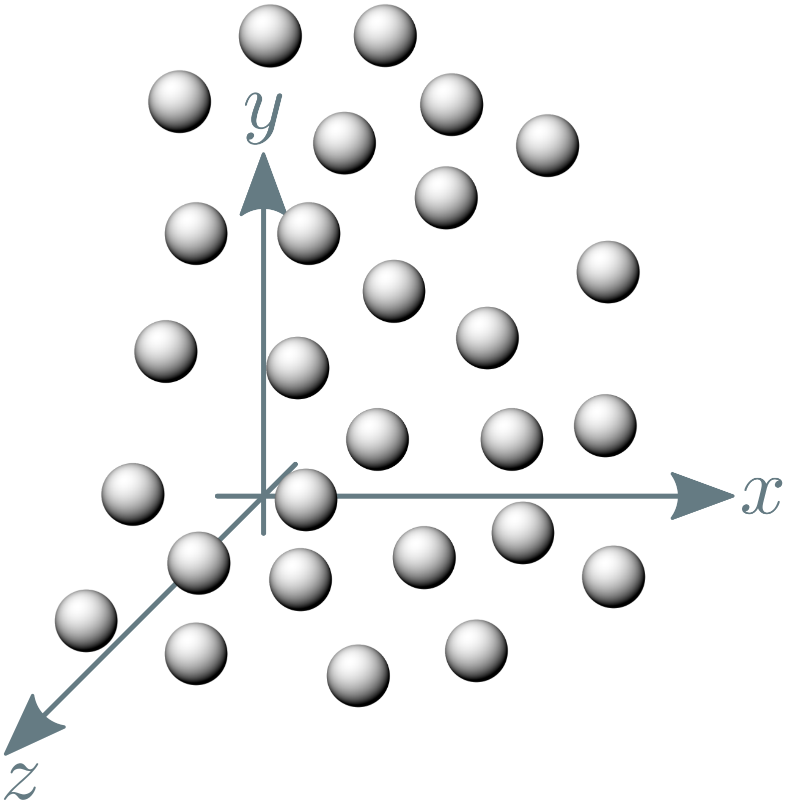
  Object Oriented Programming (OOP):

  is concerned with *data encapsulation*. Allows for modularity and *reuse* of code via *polymorphism*. OOP is good in the right place, too much OOP often is not data parallel.

  Data Oriented Programming (DOP):

  is concerned with *how data is represented, transformed and shared*. The questions you ask in *data encapsulation* (OOP) is what do objects represent? What can you do with them? How do they interact? In DOP you want to *isolate tasks and their data*; the data is modeled based on *access patterns*. Data oriented layouts are often in favor for *vectorizing* code.

DATA LAYOUTS: PARTICLE EXAMPLE



- Particles have properties like *position* x, y, z and *velocity* u, v, w and possibly other quantities they carry.
- Particles can be represented with different data structures.

Object oriented:

```
struct Particle { // the object (abstraction)
    // A particle is modeled as an object (a structure), its
    // data is encapsulated in the object.
    double x, y, z;
    double u, v, w;
};
// The set of all particles is modeled in an array (vector) of
// structures (AoS).
std::vector<Particle> particles; // an array of structures

// The data is accessed by object. The accessing pattern is
// modeled based on objects.
particles[i].x = 0.0; // x-coordinate of particle i
```

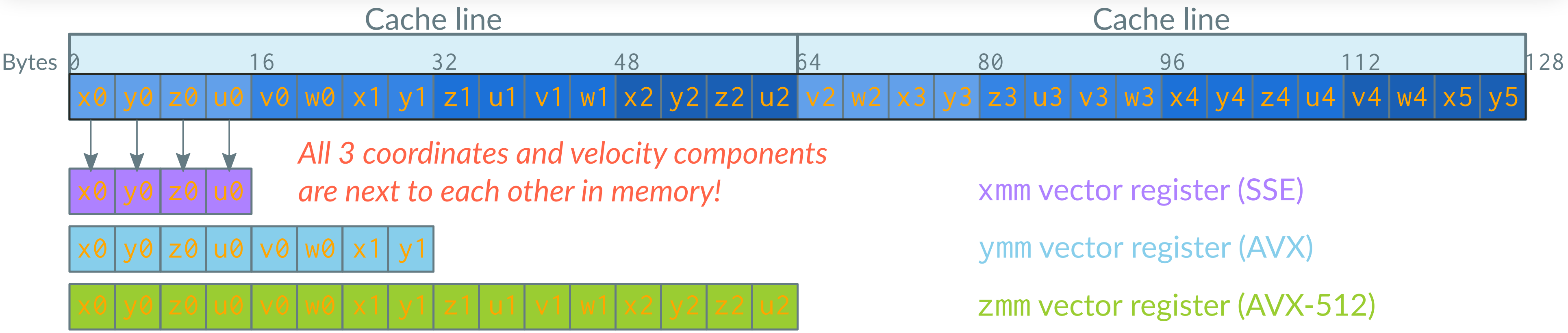
Data oriented:

```
struct Particles { // vectors of data
    // The abstraction of a single particle does not exist.  The
    // focus is on the data of all particles.
    std::vector<double> x, y, z;
    std::vector<double> u, v, w;
};
// The set of all particles is modeled in a structure of arrays
// (SoA).
Particles particles; // a structure of arrays

// The data for a particular quantity is accessed consecutively
// for all particles.
particles.x[i] = 0.0; // x-coordinate of particle i
```

DATA LAYOUTS: ARRAY OF STRUCTURES (AOS)

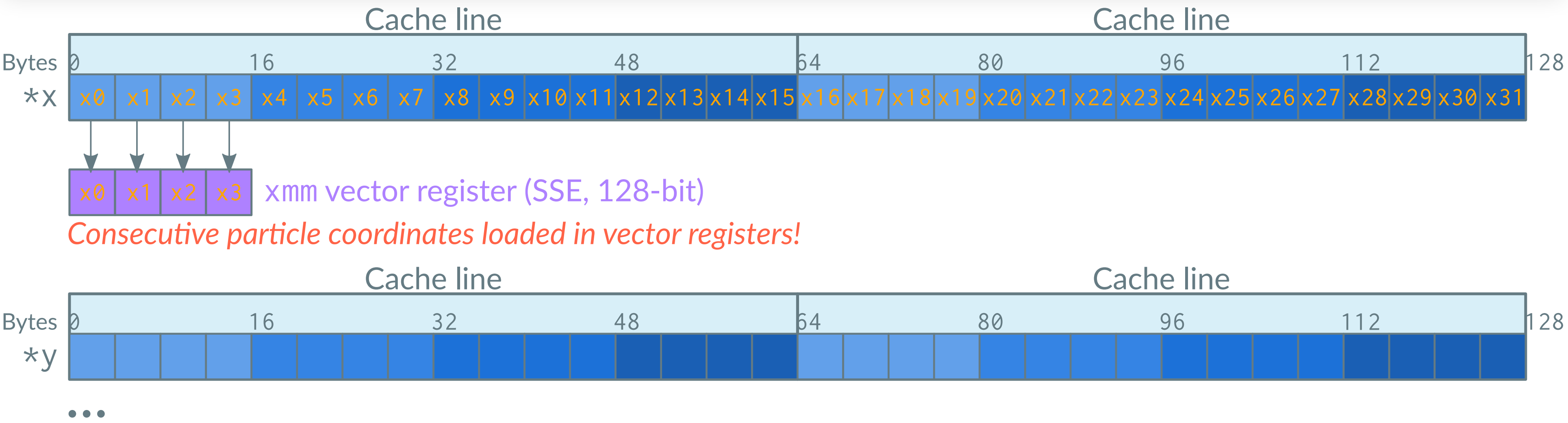
```
struct Particle {
    float x, y, z;
    float u, v, w;
};
Particle AoS[10]; // array of particle structures
```



- Object oriented designs tend to *pollute the cache* with data that may not be necessary for the current operation.
- For example, your computation requires positions only but the cache line will also fetch velocities→*waste of bandwidth!*
- This data layout is not optimal for SIMD since you issue *one instruction* which will operate on *multiple data* but the data represents *different quantities*.
- Using AoS with SIMD almost always requires *data re-ordering*.

DATA LAYOUTS: STRUCTURE OF ARRAYS (SOA)

```
struct Particles {
    float *x, *y, *z;
    float *u, *v, *w;
};
Particles SoA; // structure of arrays
posix_memalign((void**)&SoA.x, 16, 10 * sizeof(float));
// more memory allocations
```



SINGLE INSTRUCTION MULTIPLE DATA (SIMD)

- Last time we discussed the following table for floating point instructions:

Instruction				Latency single/double	
Single	**Double**	**Operation**	**Description**	**Haswell**	**Broadwell**
`vaddss`	`vaddsd`	`dst ← src2 + src1`	Floating-point add	3/3	3/3
`vsubss`	`vsubsd`	`dst ← src2 - src1`	Floating-point subtract	3/3	3/3
`vmulss`	`vmulsd`	`dst ← src2 * src1`	Floating-point multiply	5/5	3/5
`vdivss`	`vdivsd`	`dst ← src2 / src1`	Floating-point divide	<13/14-20	<11/<14
`vmaxss`	`vmaxsd`	`dst ← max(src2, src1)`	Floating-point maximum	3/3	3/3
`vminss`	`vminsd`	`dst ← min(src2, src1)`	Floating-point minimum	3/3	3/3
`vsqrtss`	`vsqrtsd`	`dst ← √src1`	Floating-point square root	13/20	13/20

- These instructions are SISD. The last two characters in the instruction name stand for *scalar single (ss)* or *scalar double (sd)*.
- *Scalar* means *scalar data* (one element for the corresponding precision).
- SIMD instructions work on *packed data*, where packed refers to multiple data elements in the (packed) vector register.

SINGLE INSTRUCTION MULTIPLE DATA (SIMD)

Recall the reduction kernel we have been discussing in conjunction with loop-unrolling last time: $s = \sum_{i=1}^{n} x_i$

```
reduce:
.L3: ; loop label
    addss xmm1, DWORD PTR [rdi]   ; latency 3
    addss xmm0, DWORD PTR 4[rdi]  ; latency 3
    add   rdi, 16
    addss xmm3, DWORD PTR -8[rdi] ; latency 3
    addss xmm2, DWORD PTR -4[rdi] ; latency 3
    cmp   rax, rdi
    jne   .L3
    addss xmm0, xmm1 ; reduce partial sum
    addss xmm0, xmm3 ; reduce partial sum
    addss xmm0, xmm2 ; reduce partial sum
    ret
```

Assembly generated with: `gcc -O2 -S`

```c
float reduce(const float *x, const int n)
{
    float sum0 = 0.0;
    float sum1 = 0.0;
    float sum2 = 0.0;
    float sum3 = 0.0;
    for (int i = 0; i < n; i += 4) {
        sum0 += x[i + 0];
        sum1 += x[i + 1];
        sum2 += x[i + 2];
        sum3 += x[i + 3];
    }
    return sum0 + sum1 + sum2 + sum3;
}
```

SINGLE INSTRUCTION MULTIPLE DATA (SIMD)

Recall the reduction kernel we have been discussing in conjunction with loop-unrolling last time: $s = \sum_{i=1}^{n} x_i$

```
reduce:
.L3: ; loop label
    addss xmm1, DWORD PTR [rdi]   ; latency 3
    addss xmm0, DWORD PTR 4[rdi]  ; latency 3
    add   rdi, 16
    addss xmm3, DWORD PTR -8[rdi] ; latency 3
    addss xmm2, DWORD PTR -4[rdi] ; latency 3
    cmp   rax, rdi
    jne   .L3
    addss xmm0, xmm1 ; reduce partial sum
    addss xmm0, xmm3 ; reduce partial sum
    addss xmm0, xmm2 ; reduce partial sum
    ret
```

Assembly generated with: `gcc -O2 -S`.

- `-O3` enables *auto-vectorization* in GCC.
- Here `addps` is a SIMD instruction. The mnemonic *ps* stands for *packed single*.
- `movups` moves 128-bit *unaligned* memory into vector register `xmm0`.

```
reduce:
.L3:
    mov      rcx, rax
    add      rax, 1
    sal      rcx, 4 ; arithmetic left shift <<4
    movups   xmm0, XMMWORD PTR [rdi+rcx] ; SIMD
    addps    xmm1, xmm0 ; latency 3, SIMD
    cmp      edx, eax
    ja       .L3 ; jump to loop label
    movaps   xmm0, xmm1
    movaps   xmm2, xmm1
    shufps   xmm0, xmm1, 85
    addss    xmm0, xmm1
    unpckhps xmm2, xmm1
    shufps   xmm1, xmm1, 255
    addss    xmm0, xmm2
    addss    xmm0, xmm1
    ret
```

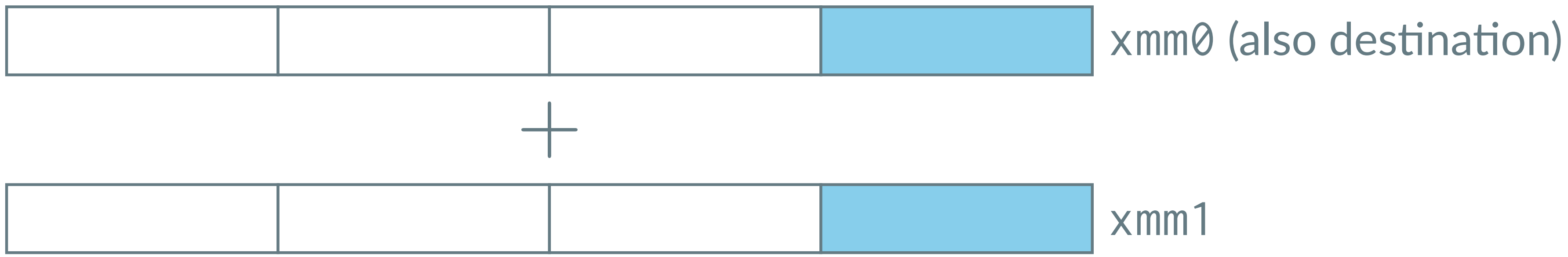
Assembly generated with: `gcc -O3 -S`. The SIMD code reduces the latency roughly by a factor of 4 → *~4x faster*.

SINGLE INSTRUCTION MULTIPLE DATA (SIMD)

Visualization of what happened in the previous two versions for the reduction kernel:

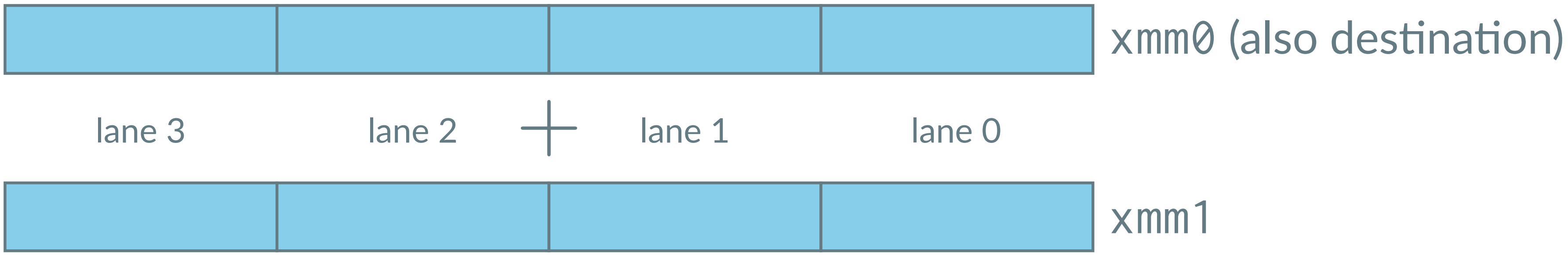
- `gcc -O2 -S`: results in single precision *scalar add*:

  `addss xmm0, xmm1`



- `gcc -O3 -S`: results in single precision *4-way vector add (SIMD)*:

  `addps xmm0, xmm1`



- ***4-way SIMD:*** there are *4 SIMD lanes* in this example. The `xmm` registers are 128-bit wide and single precision floats are 32-bit each → 4 floats in one `xmm` register.

SINGLE INSTRUCTION MULTIPLE DATA (SIMD)

Visualization of what happened in the previous two versions for the reduction kernel:

- For double precision data the number of SIMD lanes for `xmm` registers halves.

  Recall: we had to take into account the number of SIMD lanes when we calculated the nominal peak performance π for the roofline in lecture 10.

- Double precision *scalar add*: `addsd xmm0, xmm1`

- Double precision *2-way vector add (SIMD)*: `addpd xmm0, xmm1`

WORKFLOW FOR VECTORIZING CODE (SIMD)

How should you go about vectorizing code (in order of increasing effort):

1. Check if there are *libraries* that implement vectorized versions of the kernel you are looking for. Examples are Eigen or OpenBLAS for linear algebra.
2. Help the compiler with auto-vectorizing code by using the techniques we discussed in the previous slides (prevent pointer aliasing). For complex algorithms this may often lead to results that are not satisfying.
3. Use a specific *programming model* designed to exploit SIMD. Examples include: Intel ISPC, OpenMP or libraries such as highway or `vectorclass`.
4. For *performance critical kernel:*
   - Write *explicit* SIMD code using intrinsics (high-level abstraction for assembly).
   - Write assembly as a last resort.

INTEL INTRINSICS

What are Intel intrinsics?

- Special functions that can be used to write *explicit* SIMD code in C or C++.
- Similar to writing assembly inside C or C++. Better readability and more portable.
- These functions are expanded inline when compiled, no overhead associated with calling them.
- These intrinsics are specific for Intel and AMD architectures. *SIMD on CPUs is a domain specific architecture (DSA).*
- Intrinsic functions are available for *floating-point* and *integer* operations.
- Intrinsics exist for SSE, AVX, AVX2 and AVX-512. If you have a compute kernel and you want an SSE and AVX2 version you would have to implement *both*. *We will focus on SSE in class.*
- ***Helpful reference:*** Intel Intrinsics Guide

INTEL INTRINSICS HEADER AND TYPES

- To use intrinsic functions you must include the `x86intrin.h` header
- The data types used with intrinsics functions indicate the vector (media) register size:

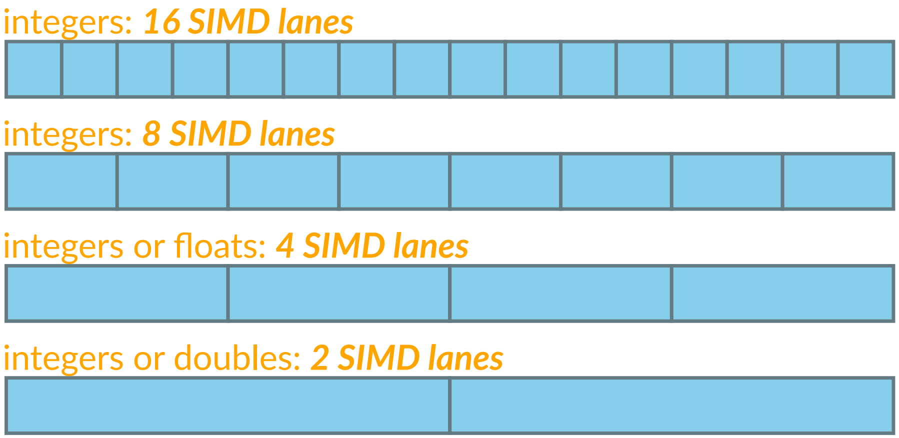
```c
// SSE 128-bit xmm registers
__m128  f; // 4 single precision floats 32-bit
__m128d d; // 2 double precision floats 64-bit
__m128i i; // int: 16 8-bit; 8 16-bit; 4 32-bit; 2 64-bit
```

```c
// AVX 256-bit ymm registers
__m256  f; // 8 single precision floats 32-bit
__m256d d; // 4 double precision floats 64-bit
__m256i i; // int: 32 8-bit; 16 16-bit; 8 32-bit; 4 64-bit
```

Examples:

SIMD lanes:

128-bit xmm register:



Broadcast value `1.0f` in all 4 SIMD lanes:

```c
#include <x86intrin.h>
__m128 ones(void) { return _mm_set_ps1(1.0f); }
```

- Assembly: https://godbolt.org/z/949vY4sqc
- ***Recall:*** function return value is stored in register `xmm0` for floating-point.
- The integer `1065353216` represented in hexadecimal is: `0x3f800000`. ***What do these 4 bytes represent?***

INTEL INTRINSICS NAMING CONVENTIONS

- The naming conventions for SSE intrinsics (128-bit):

```
_mm_<intrin_op>_<suffix>
```

  and for AVX (256-bit):

```
_mm256_<intrin_op>_<suffix>
```

- The suffix we have already seen. There are *four* permutations:
  - **ss**: ***scalar single*** (scalar code for single precision).
  - **sd**: ***scalar double*** (scalar code for double precision).
  - **ps**: ***packed single*** (packed (SIMD) code for single precision).
  - **pd**: ***packed double*** (packed (SIMD) code for double precision).
- The `<intrin_op>` can either be a single instruction, e.g., `load`, `store`, `add`, `mul`, `sub`, and so on; or it can be an operation that is a *sequence* of multiple instructions, e.g., `set`, `set1` or certain math functions. See the Intel intrinsics guide.
- The `x86intrin.h` header also defines helper macros such as `_MM_TRANSPOSE4_PS` and `_MM_SHUFFLE` for vector transposition and shuffles, respectively.
- Groups of intrinsics: *load* and *store*; *constants*; *arithmetic*; *comparison*; *conversion*; *shuffles*; and special (approximate) math.

ISSUES TO BE AWARE OF WHEN VECTORIZING CODE

- Memory alignment is important for data structures your compute kernels rely on when reading or writing data (AVX2 can handle unaligned data much better than earlier SSE).
- When you work with SSE (`xmm` registers) memory should be aligned at *16 byte* boundaries (128-bit). For AVX (`ymm` registers) it should be aligned at *32 byte* boundaries (256-bit).
- When you write SIMD code explicitly, you also must perform memory loads and stores explicitly.
- Often you need to permute data in vectors with *shuffle instructions* (we see an example later). These instructions are very powerful for advanced SIMD code.

LOAD AND STORE DATA

- Load and store operations must be done explicitly.
- There are *aligned* and *unaligned* memory operations. *Alignment is not a strict requirement* on modern machines and the penalty for unaligned memory operations is close to zero. Issues with cache lines can still degrade performance → good practice to align critical data.

Examples: (see Intel intrinsics guide for all intrinsics)

Intrinsic name	Operation	Instructions issued
`_mm_load_ps`/`_mm_store_ps`	Load or store *four* values (memory must be aligned)	`movaps`
`_mm_loadu_ps`/`_mm_storeu_ps`	Load or store *four* values (memory may be unaligned)	`movups`
`_mm_load_ss`/`_mm_store_ss`	Load or store *one* 32-bit value into first 32-bits of register, zero the other values in register (memory may be unaligned)	`movss`
`_mm_loadr_ps`/`_mm_storer_ps`	Load or store *four* values *in reverse order* (memory must be aligned)	`movaps` + shuffles

Variations of these intrinsics for double precision values exist as well!

LOAD AND STORE EXAMPLES

*Simple load and store of **aligned data**:* https://godbolt.org/z/avb49YvzP

C++ — x86-64 gcc 11.2 — -O3 -msse4

```cpp
#include <x86intrin.h>
void f(float *a, float *b)
{
    __m128 r0 = _mm_load_ps(a);
    r0 = _mm_add_ps(r0, _mm_load_ps(a + 4));
    _mm_store_ps(b, r0);
}
```

```asm
f(float*, float*):
        movaps  xmm0, XMMWORD PTR [rdi]
        addps   xmm0, XMMWORD PTR [rdi+16]
        movaps  XMMWORD PTR [rsi], xmm0
        ret
```

Edit on Compiler Explorer

> ***Note:*** the compiler did not care whether pointers `a` and `b` *alias*. *It translated the intrinsics directly into assembly!*

*Simple load and store of **unaligned data**:* https://godbolt.org/z/Y3hr4aara

C++ — x86-64 gcc 11.2 — -O3 -msse4

```cpp
#include <x86intrin.h>
void f(float *a, float *b)
{
    __m128 r0 = _mm_loadu_ps(a);
    r0 = _mm_add_ps(r0, _mm_loadu_ps(a + 4));
    _mm_storeu_ps(b, r0);
}
```

```asm
f(float*, float*):
        movups  xmm0, XMMWORD PTR [rdi]
        movups  xmm1, XMMWORD PTR [rdi+16]
        addps   xmm0, xmm1
        movups  XMMWORD PTR [rsi], xmm0
        ret
```

Edit on Compiler Explorer

> ***Note:*** unaligned loads *are not allowed* for memory operands of the `addps` instruction for example! *Less efficient.*

LOAD AND STORE EXAMPLES

Aligned reverse load ***and unaligned store:*** https://godbolt.org/z/dne4zPM7n

C++

```cpp
#include <x86intrin.h>
void f(float *a, float *b) {
    _mm_storeu_ps(b, _mm_loadr_ps(a));
}
```

x86-64 gcc 11.2 -O3 -msse4

```asm
f(float*, float*):
        movaps  xmm0, XMMWORD PTR [rdi]
        shufps  xmm0, xmm0, 27
        movups  XMMWORD PTR [rsi], xmm0
        ret
```

Edit on Compiler Explorer

Note: Reversing the vector elements must be done with another *shuffle* instruction (`shufps`). *Intra-vector* shuffles are very powerful operations that allow you to vectorize more complex code/algorithms. We look at how they work at the end of this slide set.

CONSTANTS

- Floating-point constants cannot be encoded in the instruction like some integers can.
- They are compiled into the binary representation and must be *loaded* into registers when they are needed.

Examples: (see Intel intrinsics guide for all intrinsics)

Intrinsic name	Operation	Instructions issued
`_mm_set_ps`	Set *four* values passed by arguments	Sequence
`_mm_set1_ps`/`_mm_set_ps1`	Broadcast *one* value to all SIMD lanes	Sequence
`_mm_set_ss`	Set value in the low 32-bit and zero the higher bits	Sequence
`_mm_setr_ps`	Same as `_mm_set_ps` but set values in *reverse* order	Sequence
`_mm_setzero_ps`	Returns a vector with all values set to zero	`xorps`, `pxor`

Variations of these intrinsics for double precision values exist as well!

CONSTANTS EXAMPLES

Least significant
bit (LSB)

1.0	2.0	3.0	4.0	= _mm_set_ps(4.0, 3.0, 2.0, 1.0)
4.0	3.0	2.0	1.0	= _mm_setr_ps(4.0, 3.0, 2.0, 1.0)
2.0	2.0	2.0	2.0	= _mm_set1_ps(2.0)
1.0	0.0	0.0	0.0	= _mm_set_ss(1.0)
0.0	0.0	0.0	0.0	= _mm_setzero_ps()

xmm register

Examples:

_mm_set_ps:	https://godbolt.org/z/xsY6Gqf9x
_mm_setr_ps:	https://godbolt.org/z/916YfPWc3
_mm_set1_ps:	https://godbolt.org/z/3EMKTzv37
_mm_set_ss:	https://godbolt.org/z/xaenv8hsb
_mm_setzero_ps:	https://godbolt.org/z/jnaa1W6oT

ARITHMETIC

SSE (you have seen the *scalar forms* already in the previous lecture)

Intrinsic name	Operation	Instructions issued
_mm_add_ps	Addition	addps
_mm_sub_ps	Subtraction	subps
_mm_mul_ps	Multiplication	mulps
_mm_div_ps	Division	divps
_mm_sqrt_ps	Square root	sqrtps
_mm_rcp_ps	Reciprocal (approximate)	rcpps
_mm_min_ps	Minimum	minps
_mm_max_ps	Maximum	maxps

SSE3

Intrinsic name	Operation	Instructions issued
_mm_addsub_ps	Alternate add and subtract	addsubps
_mm_hadd_ps	Horizontal add	haddps
_mm_hsub_ps	Horizontal subtract	hsubps

SSE4

Intrinsic name	Operation	Instructions issued
_mm_dp_ps	Conditional dot-product	dpps

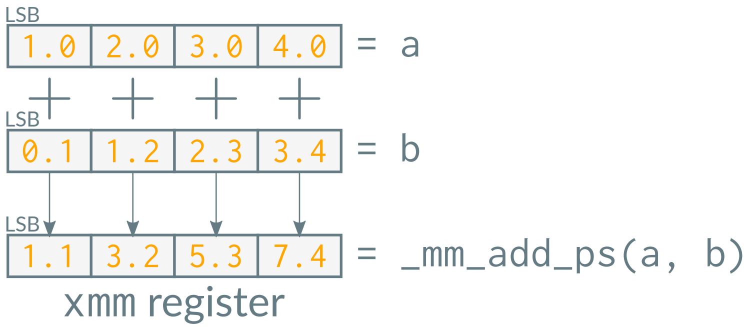
FMA

Intrinsic name	Operation	Instructions issued
_mm_fmadd_ps	Fused multiply-add	vfmadd132ps, vfmadd213ps, vfmadd231ps
_mm_fmsub_ps	Fused multiply-subtract	vfmsub132ps, vfmsub213ps, vfmsub231ps
_mm_fnmadd_ps	Fused negated-multiply-add	vfnmadd132ps, vfnmadd213ps, vfnmadd231ps
_mm_fnmsub_ps	Fused negated-multiply-subtract	vfnmsub132ps, vfnmsub213ps, vfnmsub231ps
_mm_fmaddsub_ps	Fused alternate multiply-subtract-add	vfmaddsub132ps, vfmaddsub213ps, vfmaddsub231ps
_mm_fsubadd_ps	Fused alternate multiply-add-subtract	vfmsubadd132ps, vfmsubadd213ps, vfmsubadd231ps

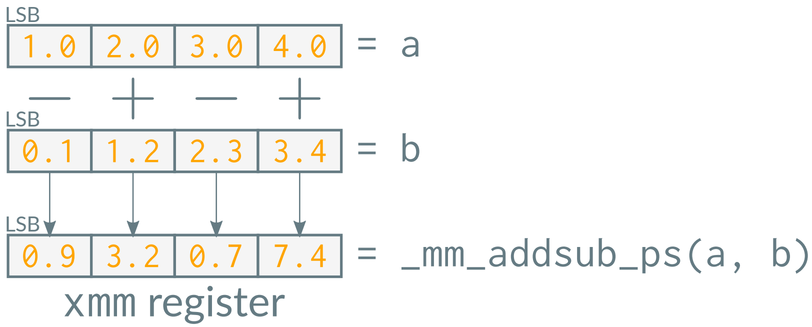
Many of these also exist for scalar code and double precision: ss, sd and pd.

ARITHMETIC EXAMPLES

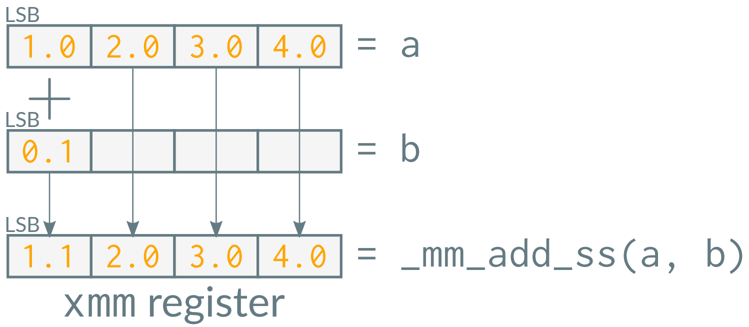
SIMD addition:



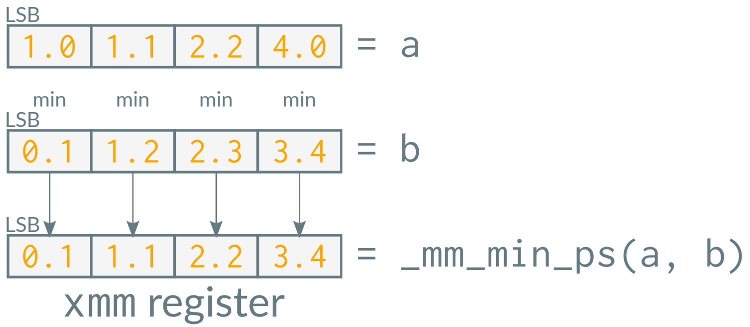
SIMD alternate sub/add:



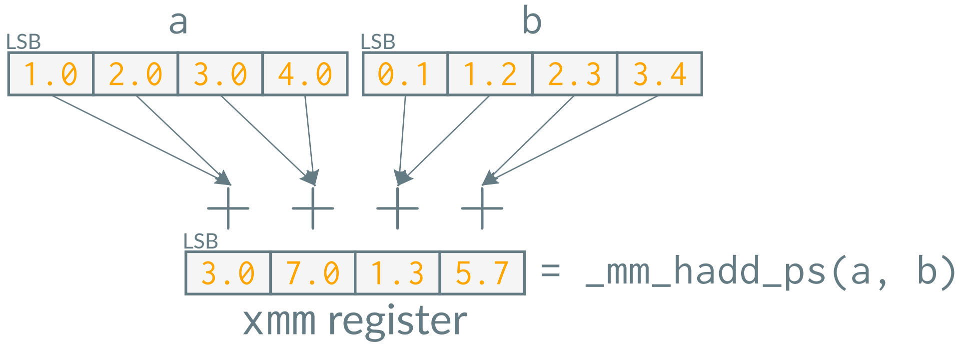
Scalar addition:



SIMD minimum:



SIMD horizontal add:



Operations are similar for other intrinsics:

- _mm_sub_ps
- _mm_mul_ps
- _mm_hsub_ps
- and others

EXAMPLE: SIMD VECTOR ADDITION

Consider the SAXPY kernel:

$$y_i = \alpha x_i + y_i \quad i = 1, 2, \ldots, N$$

```c
#include <x86intrin.h>
#define N (1 << 11) // size of vectors

// compute kernels
void saxpy(float *x, float *y, float a, size_t n);
void saxpy_SSE(float *x, float *y, float a, size_t n);

int main(void)
{
    // allocate aligned vectors and initialize (could use any
    // other aligned allocator as well, e.g., posix_memalign)
    float *x = (float *)_mm_malloc(N * sizeof(float), 16);
    float *y = (float *)_mm_malloc(N * sizeof(float), 16);
    initialize(x, N);
    initialize(y, N);

    // run SAXPY kernels
    saxpy(x, y, 2.0, N); // scalar kernel implementation
    saxpy_SSE(x, y, 2.0, N); // SIMD kernel implementation

    // clean up
    _mm_free(x);
    _mm_free(y);
    return 0;
}
```

Scalar kernel:

```c
#include <stddef.h>
void saxpy(const float *x, float *y, float a, size_t n)
{
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

SSE (SIMD) kernel:

```c
#include <x86intrin.h>
void saxpy_SSE(const float *x, float *y, float a, size_t n)
{
    const __m128 a4 = _mm_set1_ps(a);
    // assumes n % 4 == 0 -> 4 SIMD lanes
    for (size_t i = 0; i < n; i += 4) {
        __m128 r0 = _mm_load_ps(x + i);
        __m128 r1 = _mm_load_ps(y + i);
        _mm_store_ps(y + i, _mm_add_ps(_mm_mul_ps(r0, a4), r1));
    }
}
```

- Load the constant α into a register.
- Explicit load of data in x and y.
- Explicit store of result computed with packed `addps` and `mulps` SIMD instructions.
- Loop counter must be advanced by the number of SIMD lanes.

EXAMPLE: SIMD VECTOR ADDITION

Scalar kernel:

```c
#include <stddef.h>
void saxpy(float *x, float *y, float a, size_t n)
{
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

```asm
saxpy:
.L3:
    movaps  xmm1, xmm0
    mulss   xmm1, DWORD PTR [rdi+rax*4]
    addss   xmm1, DWORD PTR [rsi+rax*4]
    movss   DWORD PTR [rsi+rax*4], xmm1
    add     rax, 1
    cmp     rdx, rax
    jne     .L3
    ret
```

The SIMD code should be 4x faster because we use 4 SIMD lanes. See `saxpy` lecture code in `git` repo for an implementation example.

SSE (SIMD) kernel:

```c
#include <x86intrin.h>
void saxpy_SSE(float *x, float *y, float a, size_t n)
{
    const __m128 a4 = _mm_set1_ps(a);
    // assumes n % 4 == 0 -> 4 SIMD lanes
    for (size_t i = 0; i < n; i += 4) {
        __m128 r0 = _mm_load_ps(x + i);
        __m128 r1 = _mm_load_ps(y + i);
        _mm_store_ps(y + i, _mm_add_ps(_mm_mul_ps(r0, a4), r1));
    }
}
```

```asm
saxpy_SSE:
.L3:
    movaps  xmm1, XMMWORD PTR [rdi+rax*4]
    mulps   xmm1, xmm0
    addps   xmm1, XMMWORD PTR [rsi+rax*4]
    movaps  XMMWORD PTR [rsi+rax*4], xmm1
    add     rax, 4
    cmp     rdx, rax
    ja      .L3
    ret
```

Running the code on a Broadwell node on the academic cluster:

```
$ ./main
Scalar saxpy: 3.682487e-06 sec
SSE saxpy: 9.097718e-07 sec (4.05x speedup)
```

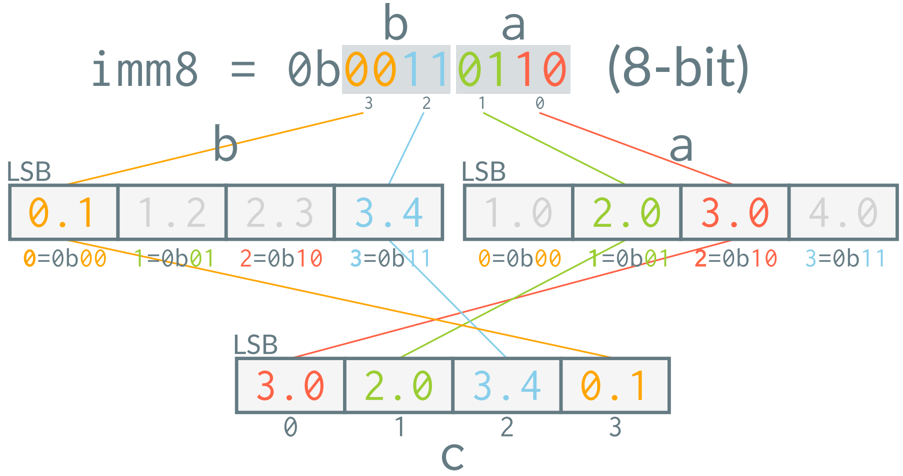
SHUFFLES

- It is often necessary to permute the data elements in a SIMD vector (inter-lane communication).
- Such operations are called *shuffles*.
- You have already seen them in some of the assembly we studied. Examples are the `_mm_set1_ps` or `_mm_loadr_ps` intrinsics.
- Shuffles do have a latency of about 1 clock cycle and are not for free but they can accomplish complex tasks.
- Sometimes you can achieve a more efficient code using shuffles instead of an alternative instruction.

Example:

```
__m128 _mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8);
```

- The `imm8` is an 8-bit control variable to *select* elements in vectors a and b and return the permutation.
- Pairs of *two* bits are used to index into the SIMD vector.
- The first *four* bits are used to select *two* elements in a and next *four* bits are used to select two elements in b.



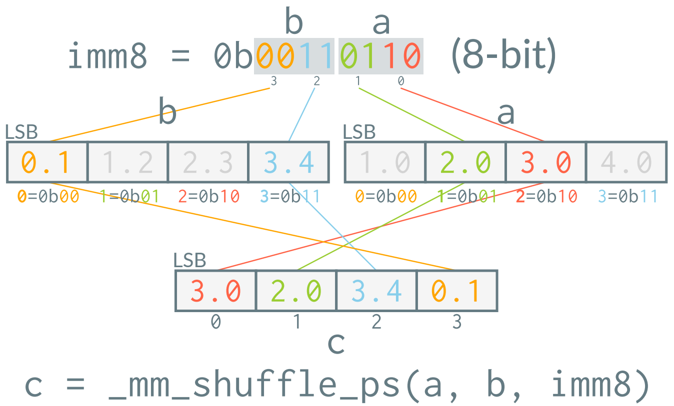
c = _mm_shuffle_ps(a, b, imm8)

SHUFFLES

- A helper macro can be used with shuffles if the binary notation for `imm8` is not intuitive: `_MM_SHUFFLE`.
- The example on the previous slide is identical to

```
__m128 c = _mm_shuffle_ps(a, b, _MM_SHUFFLE(0, 3, 1, 2));
```

Read it in *reverse*: from `a` take element 2, then 1 and from `b` take 3 then 0.



Intrinsic name	Operation	Instructions issued
`_mm_shuffle_ps`	Shuffle vector	`shufps`
`_mm_unpackhi_ps`	Unpack and interleave from high half of register	`unpckhps`
`_mm_unpacklo_ps`	Unpack and interleave from low half of register	`unpcklps`
`_mm_move_ss`	Copy low 32-bit from `b` and pass the higher bit values from `a`	`movss`
`_mm_movehl_ps`	Move high values to low	`movhlps`
`_mm_movelh_ps`	Move low values to high	`movlhps`
`_mm_movemask_ps`	Create sign mask	`movmskps`

A few more shuffle intrinsics were added in SSE3 and SSE4, in particular `_mm_blend_ps`, `_mm_blendv_ps`, `_mm_insert_ps`, `_mm_extract_ps`, `_mm_movehdup_ps` and `_mm_moveldup_ps`.

SHUFFLE EXAMPLES

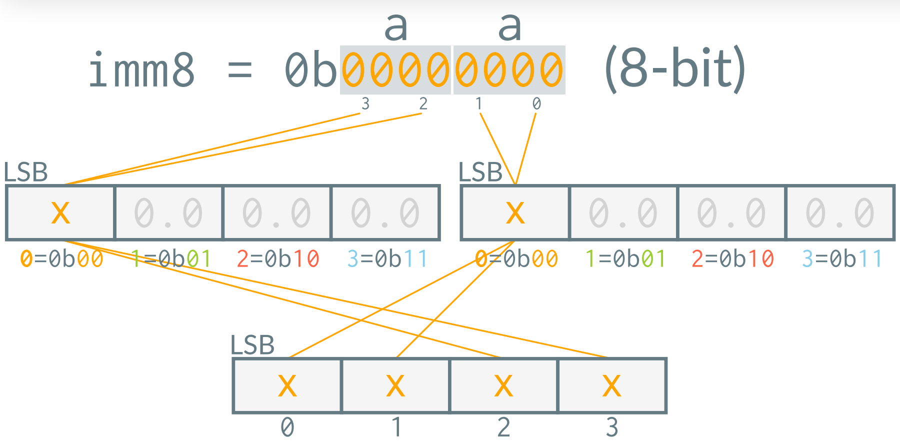
Broadcasting a scalar value:

Recall the code from the previous example to broadcast a value in a vector:

```c
#include <x86intrin.h>
__m128 set1_ps(float x) {
    return _mm_set1_ps(x);
}
```

This function compiles to:

```asm
set1_ps: ; xmm0 contains x in the low 32-bit
    shufps  xmm0, xmm0, 0
    ret
```



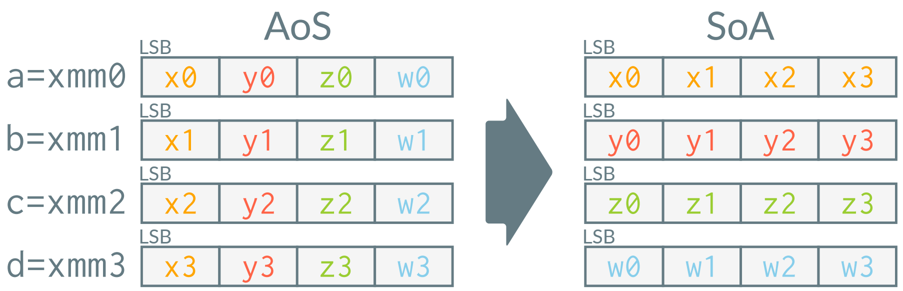
a = _mm_shuffle_ps(a, a, 0)

Transposing data from AoS to SoA:

Consider this array of structures:

```c
struct Pixel {
    float x, y, z, w;
};
Pixel image[1024];
```

Writing SIMD code with the `image` data structure often requires *data reordering* to a structure of arrays (better suited for vectorization). The `_MM_TRANSPOSE4_PS` macro can be used to perform the shuffles:



_MM_TRANSPOSE4_PS(a,b,c,d)

Results in *8 shuffle* instructions and 4 register moves (https://godbolt.org/z/aM16rxGPM)

SHUFFLE EXAMPLES

Revisit: the SIMD reduction kernel from a previous slide: $s = \sum_{i=1}^{n} x_i$

```
1  reduce:
2  .L3:
3      mov     rcx, rax
4      add     rax, 1
5      sal     rcx, 4 ; arithmetic left shift <<4
6      movups  xmm0, XMMWORD PTR [rdi+rcx] ; SIMD
7      addps   xmm1, xmm0 ; latency 3, SIMD
8      cmp     edx, eax
9      ja      .L3 ; jump to loop label
10     movaps  xmm0, xmm1
11     movaps  xmm2, xmm1
12     shufps  xmm0, xmm1, 85
13     addss   xmm0, xmm1
14     unpckhps xmm2, xmm1
15     shufps  xmm1, xmm1, 255
16     addss   xmm0, xmm2
17     addss   xmm0, xmm1
18     ret
```

In lines 10–17 the SIMD lanes must be reduced as well! This results in shuffles.

Line 12: (after reduction SIMD loop)

a=xmm0 (LSB)

x0	x1	x2	x3

b=xmm1 (LSB)

x0	x1	x2	x3

xmm0 (LSB)

x1	x1	x1	x1

= `_mm_shuffle_ps(a,b,85)`

Line 14:

a=xmm2 (LSB)

x0	x1	x2	x3

b=xmm1 (LSB)

x0	x1	x2	x3

xmm2 (LSB)

x2	x2	x3	x3

= `_mm_unpackhi_ps(a,b)`

Line 15:

a=xmm1 (LSB)

x0	x1	x2	x3

b=xmm1 (LSB)

x0	x1	x2	x3

xmm1 (LSB)

x3	x3	x3	x3

= `_mm_shuffle_ps(a,b,255)`

After all the scalar `addss` instructions, register `xmm0` contains:

xmm0 (LSB)

x0+x1+x2+x3	x1	x1	x1

AUTO-VECTORIZATION

- When you pass the `-O3` flag to GCC or LLVM Clang you turn on auto-vectorization.
- The compiler uses algorithms to auto-vectorize code:
    1. Loops with: `-ftree-loop-vectorize`,
    2. Basic blocks (loop-unrolling) with SLP (superword-level parallelism): `-ftree-slp-vectorize`.

  See https://gcc.gnu.org/projects/tree-ssa/vectorization.html for more details about auto-vectorization in GCC and https://www.llvm.org/docs/Vectorizers.html for LLVM.
- Both of these vectorization techniques are enabled by `-ftree-vectorize` which is implied by `-O3`. See `man gcc` for further information.
- Auto-vectorization works well for simple code structures. Often you need to help the compiler to obtain better results.
- Help with hints or manual code transformations that allow the compiler to identify DLP or ILP automatically.
- The *strict aliasing rule* will prevent the compiler from doing an optimization such as auto-vectorization.

AUTO-VECTORIZATION

Compiler flags for auto-vectorization:

- You have more fine grained control to target a specific architecture when you compile SIMD code.
- You can use the flag `-mARCH` to specify a target architecture. There are many possibilities for `ARCH`, some important ones are: `sse2`, `sse3`, `sse4`, `avx`, `avx2`, `fma`.
- If you specify both, `-msse3` and `-mavx2`, then AVX2 instructions will be used.
- To enable FMA instructions, pass the flag `-mfma`.
- With GCC you can use the flag `-fdump-tree-vect-stats` to generate a report and see if the compiler vectorized code.
- The best way to verify what the compiler did is to look at the generated assembly code. GCC and Clang results are usually *very* different.
- Use the `-S` flag with GCC to generate assembly code. GCC uses AT&T syntax by default. You can dump Intel syntax with the flag `-masm=intel`

ALIASING AND AUTO-VECTORIZATION

```c
void f(float *dst, float *src, float a)
{
    for (int i = 0; i < 1024; ++i) {
        dst[i] = dst[i] + a * src[i];
    }
}
```

- The code would be easy to vectorize.
- However, the compiler cannot vectorize this code in a straightforward way because of potential *aliasing* of `src` and `dst`.
- It will insert a *runtime check* to decide whether to execute vectorized code or not.
- Check for AVX2 target with FMA instructions: `-O3 -mavx2 -mfma`.

```asm
f:
  lea rdx, [rsi+4] ; these lines perform a check for
  mov rax, rdi     ; aliasing of dst (rdi) and src
  sub rax, rdx     ; (rsi).  GCC does only check for
  cmp rax, 24      ; if there is overlap in the first
  jbe .L5          ; few bytes.  Jump to .L5 if alias.
  vbroadcastss ymm0, xmm0
  xor eax, eax
.L3: ; vectorized code (AVX2)
  vmovups ymm1, YMMWORD PTR [rsi+rax]
  vfmadd213ps ymm1, ymm0, YMMWORD PTR [rdi+rax]
  vmovups YMMWORD PTR [rdi+rax], ymm1
  add rax, 32 ; advance by 32 byte (256-bit ymm)
  cmp rax, 4096
  jne .L3
  vzeroupper
  ret
.L5:
  xor eax, eax
.L2: ; scalar code (runs on single floats!)
  vmovss xmm1, DWORD PTR [rsi+rax]
  vfmadd213ss xmm1, xmm0, DWORD PTR [rdi+rax]
  vmovss DWORD PTR [rdi+rax], xmm1
  add rax, 4 ; advance by 4 byte (float)
  cmp rax, 4096
  jne .L2
  ret
```

GCC 11.2
(see https://godbolt.org/z/PddGbvP6T)

ALIASING AND AUTO-VECTORIZATION

```
void f(float *dst, float *src, float a)
{
    for (int i = 0; i < 1024; ++i) {
        dst[i] = dst[i] + a * src[i];
    }
}
```

- Same code compiled with LLVM Clang.
- Clang does only run vectorized code if the two arrays are *completely non-overlapping!*
- More aggressive optimizations than GCC for `-O3`. Clang performs a 4-way loop unrolling in addition to SIMD.
- Code generated with Clang 14.0.0 (see https://godbolt.org/z/14z3jK9cz)

```
f:
  lea rax, [rsi + 4096] ; checks for full array overlap!
  cmp rax, rdi
  jbe .LBB0_4 ; execute SIMD if non-overlapping
  lea rax, [rdi + 4096] ; checks for full array overlap!
  cmp rax, rsi
  jbe .LBB0_4 ; execute SIMD if non-overlapping
  xor eax, eax
.LBB0_3: ; scalar code (single float, 4-way loop unroll)
  vmovss xmm1, dword ptr [rsi + 4*rax]
  vfmadd213ss xmm1, xmm0, dword ptr [rdi + 4*rax]
  vmovss dword ptr [rdi + 4*rax], xmm1
  vmovss xmm1, dword ptr [rsi + 4*rax + 4]
  vfmadd213ss xmm1, xmm0, dword ptr [rdi + 4*rax + 4]
  vmovss dword ptr [rdi + 4*rax + 4], xmm1
  vmovss xmm1, dword ptr [rsi + 4*rax + 8]
  vfmadd213ss xmm1, xmm0, dword ptr [rdi + 4*rax + 8]
  vmovss dword ptr [rdi + 4*rax + 8], xmm1
  vmovss xmm1, dword ptr [rsi + 4*rax + 12]
  vfmadd213ss xmm1, xmm0, dword ptr [rdi + 4*rax + 12]
  vmovss dword ptr [rdi + 4*rax + 12], xmm1
  add rax, 4
  cmp rax, 1024 ; 1024 only because of loop unrolling
  jne .LBB0_3
  jmp .LBB0_6
.LBB0_4: ; vectorized code entry point (AVX2)
  vbroadcastss ymm0, xmm0
  xor eax, eax
.LBB0_5: ; 4-way unrolled SIMD code
  vmovups ymm1, ymmword ptr [rsi + 4*rax]
  vmovups ymm2, ymmword ptr [rsi + 4*rax + 32]
  vmovups ymm3, ymmword ptr [rsi + 4*rax + 64]
  vmovups ymm4, ymmword ptr [rsi + 4*rax + 96]
  vfmadd213ps ymm1, ymm0, ymmword ptr [rdi + 4*rax]
  vfmadd213ps ymm2, ymm0, ymmword ptr [rdi + 4*rax + 32]
```

ALIASING AND AUTO-VECTORIZATION

- You can avoid this additional overhead if you know that your arrays will not alias each other (most of the time they will not overlap).
- You could use the `__restrict__` keyword to help the compiler. For example:

```
void f(float *__restrict__ dst,
       float *__restrict__ src,
       float a)
```

- The `__restrict__` keyword does not always work. For example:

```
void f(int *a, int k, int c, int m)
{
// The following GCC specific pragma enforces
// auto-vectorization for this loop and neglect
// the cases when k < -m or k >= m
#pragma GCC ivdep
    for (int i = 0; i < m; i++) {
        a[i] = a[i + k] * c;
    }
}
```

See https://gcc.gnu.org/onlinedocs/gcc-11.2.0/gcc/Loop-Specific-Pragmas.html#Loop-Specific-Pragmas

- Both methods yield auto-vectorized SIMD code: Clang 14.0.0 with `__restrict__` (see https://godbolt.org/z/1rcP19ajP)
- GCC 11.2 with `ivdep` pragma (see https://godbolt.org/z/Wc9MWa3Gx)
- These methods (`#pragma`) are to help the compiler generate auto-vectorized code but are specific to a certain compiler. The `__restrict__` is specific to C but may not work in all cases.
- An alternative to enforce SIMD in loops which is more portable than the `ivdep` pragma in GCC is provided by OpenMP.

SIMD SUPPORT IN OpenMP

openmp.org

- OpenMP has SIMD support that can aid the compiler to auto-vectorize code in loops and functions.
- Since OpenMP is a specification, helping the compiler with auto-vectorization through OpenMP is a more portable approach.
- The construct to use is

```
#pragma omp simd [clause[[,]clause]…] new-line
    loop-nest
```

for SIMD *without threads* (not related to shared memory).

- You can combine this construct with a *worksharing-loop* in a parallel region

```
#pragma omp for simd [clause[[,]clause]…] new-line
    loop-nest
```

for SIMD *with threads*

- See Section 2.11.5 in the specification.

Example:

- Consider again the code

```
void f(float *dst, float *src, float a)
{
#pragma omp simd // force auto-vectorization of loop
    for (int i = 0; i < 1024; ++i) {
        dst[i] = dst[i] + a * src[i];
    }
}
```

- GCC 11.2: `-O3 -fopenmp` results in SSE code (https://godbolt.org/z/jsdxPaj6v)
- GCC 11.2: `-O3 -fopenmp -mavx2 -mfma` results in AVX2 with FMA code (https://godbolt.org/z/ozbW4rjvx)
- Both versions *ignore* aliasing between `src` and `dst`.
- Similar results can be obtained with `__restrict__`.
- *Without* the `-fopenmp` flag the compiler generates *scalar code*.

RECAP

- Data alignment is not a strict requirement but results in better performance in general.
- You should be aware of the following fundamental data layouts: array of structures (AoS) and structure of arrays (SoA).
- Aliasing in C/C++ is often a reason for slower code compared to Fortran for example.
- Single-Instruction-Multiple-Data (SIMD) exploits data-level parallelism (DLP).
- The compiler and the OpenMP specification may help you with SIMD.
- You can use Intel intrinsic functions to write highly optimized code (*but be careful, it is also easy to degrade performance!*)
- Manual vectorization is related with large development overhead and should only be considered for performance critical kernels.

Further reading:

- Intel Intrinsics Guide: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
- *"Intel© 64 and IA-32 Architectures Software Developer's Manual"*,
  Volume 2, Instruction Set Reference, pdf
- Chapter 4 in *"Computer Architecture"*,
  J. Hennessy and D. Patterson, Morgan Kaufmann 2019