

# Advanced Systems Lab

Spring 2024

*Lecture:* SIMD extensions, AVX, intrinsics, compiler vectorization

**Instructor:** Markus Püschel

**TA:** Tommaso Pegolotti, several more



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

1

## Flynn's Taxonomy

	Single instruction	Multiple instruction
Single data	<i>SISD</i> Uniprocessor	<i>MISD</i>
Multiple data	<i>SIMD</i> Vector computer Short vector extensions GPU	<i>MIMD</i> Multiprocessors VLIW

Flynn, Michael J. "*Very high-speed computing systems*".  
*Proceedings of the IEEE*. 54(12): 1901–1909, 1966

2

2

# SIMD Extensions and AVX

AVX intrinsics

Compiler vectorization

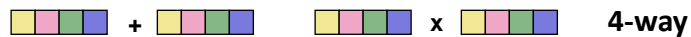
*The first version of this lecture (for SSE) was created together with Franz Franchetti (ECE, Carnegie Mellon) in 2008*

*Joao Rivera helped with the update to AVX in 2019*

3

3

# SIMD Vector Extensions



What is it?

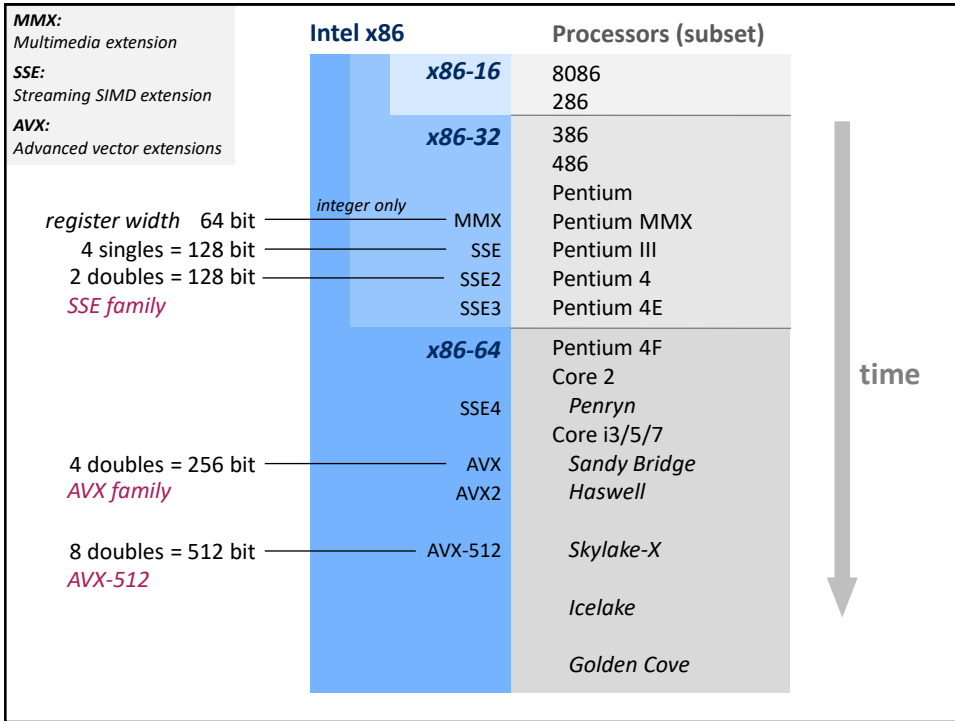
- *Extension of the ISA*
- *Data types and instructions for the parallel computation on short (length 2, 4, 8, ...) vectors of integers or floats*
- *Names: SSE, SSE2, AVX, AVX2 ...*

Why do they exist?

- **Useful:** *Many applications have the necessary fine-grain parallelism  
Then: speedup by a factor close to vector length*
- **Doable:** *Relatively easy to design by replicating functional units and space  
became available due to increasing transistor density*

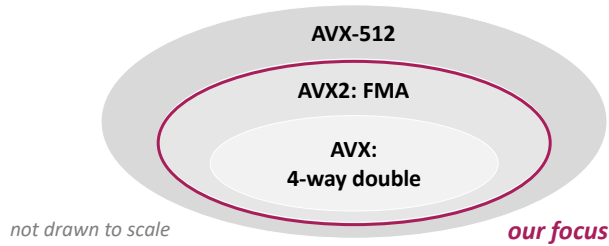
4

4



5

## Example AVX Family: Floating Point



AVX: introduces three-operand instructions ( $c = a + b$  vs.  $a = a + b$ )

AVX2: Introduces fused multiply-add (FMA):  $c = c + a * b$

Intel: three-operand FMA (FMA3), FMA4 exists on some AMD processors

6

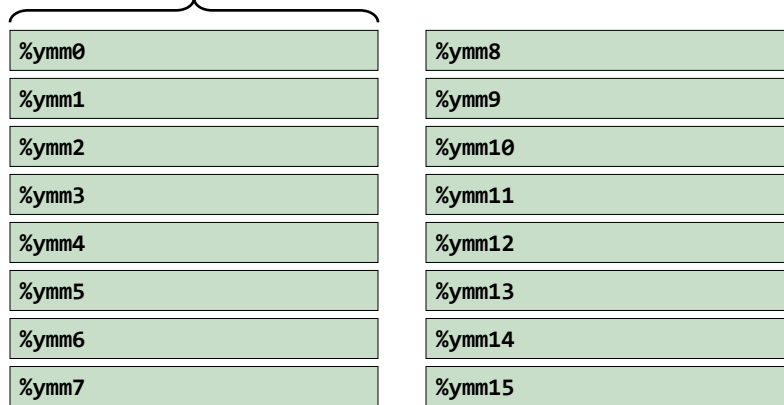
6

# Haswell/Skylake/ ...

Have AVX2

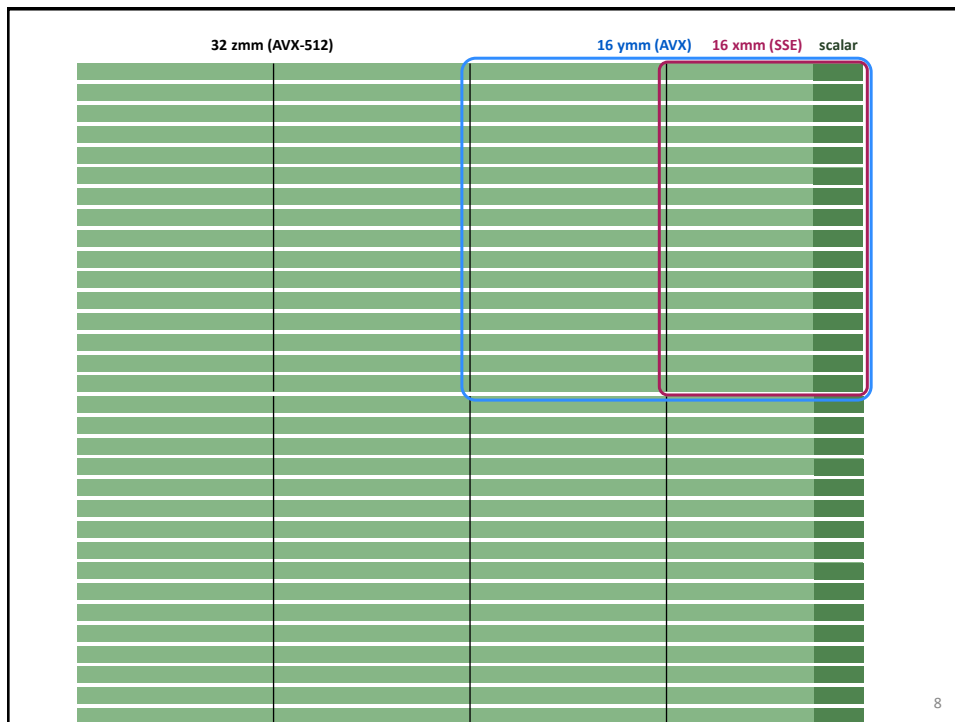
16 AVX registers

256 bit = 4 doubles = 8 singles



7

7



8

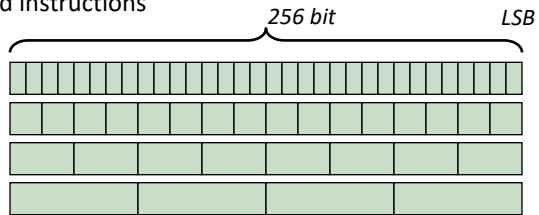
8

# AVX Registers (ymm0-ymm15)

Used for different data types and instructions

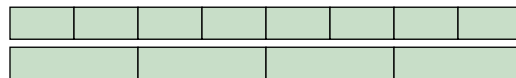
Integer vectors:

- 32-way byte
- 16-way 2 bytes
- 8-way 4 bytes
- 4-way 8 bytes



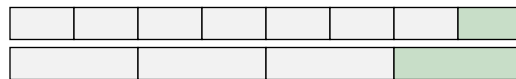
Floating point vectors:

- 8-way single
- 4-way double



Floating point scalars:

- single
- double



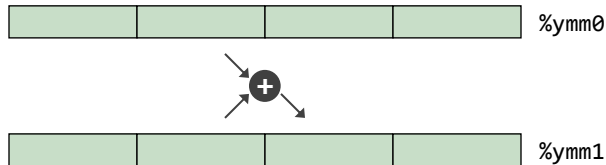
9

9

# AVX Instructions: Examples

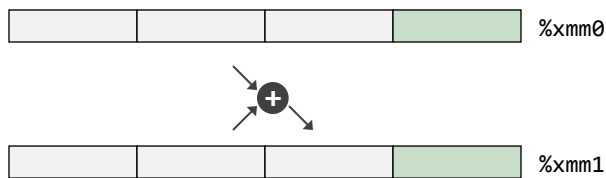
*(three-operand!)*

Double precision *4-way vector float add*: `vaddpd %ymm1 %ymm0 %ymm1`



*(two-operand!)*

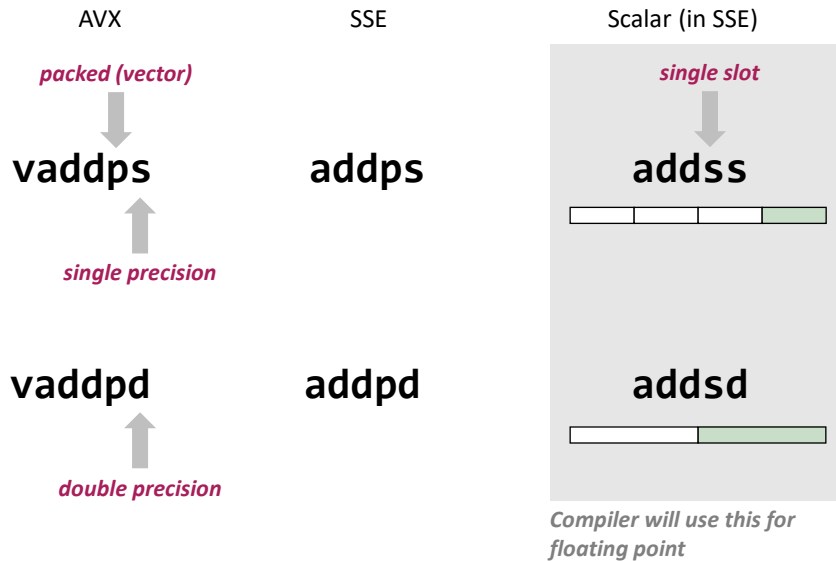
Double precision *scalar float add (in SSE2)*: `addsd %xmm0 %xmm1`



10

10

## Instruction Names (Adds in Assembly)



11

11

## x86-64 FP Code Example

Inner product of two vectors

- Double precision arithmetic
- Compiled: **not vectorized**, uses (single-slot) SSE instructions

```
double ipf (double x[],
            double y[],
            int n) {
    int i;
    double result = 0.0;
    for (i = 0; i < n; i++)
        result += x[i]*y[i];
    return result;
}
```

```
ipf:
    xorpd    %xmm1, %xmm1           # result = 0.0
    xorl    %ecx, %ecx             # i = 0
    jmp     .L8                   # goto middle
.L10:
    movslq  %ecx, %rax             # icpy = i
    incl    %ecx                  # i++
    movsd   (%rsi,%rax,4), %xmm0   # t = y[icpy]
    mulsd   (%rdi,%rax,4), %xmm0   # t *= x[icpy]
    addsd   %xmm0, %xmm1           # result += t
.L8:
    cmpl   %edx, %ecx             # i:n
    jnl    .L10                  # if < goto loop
    movapd %xmm1, %xmm0           # return result
    ret
```

12

12

## AVX: How to Take Advantage?



Necessary: fine grain parallelism

Options (ordered by effort):

- Use vectorized libraries (*easy, not always available*)
- Compiler vectorization (*this lecture*)
- Use intrinsics (*this lecture*)
- Write assembly

We will focus on floating point and double precision (4-way)

13

13

## SIMD Extensions and AVX

Overview: AVX family

*AVX intrinsics*

Compiler vectorization

References:

[\*Intel Intrinsics Guide\*](#)

*(easy access to all instructions, nicely done!)*

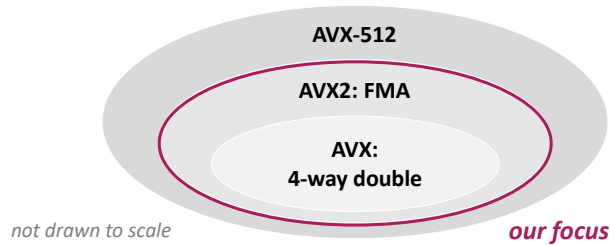
*Intel icc compiler manual*

*Visual Studio manual*

14

14

## Example AVX Family: Floating Point



AVX: introduces three-operand instructions ( $c = a + b$  vs.  $a = a + b$ )

AVX2: Introduces fused multiply-add (FMA):  $c = c + a * b$

15

15

## Intrinsics

Enable explicit use of vector instructions in C/C++

Assembly coded C functions

Expanded inline upon compilation: no overhead

Like writing assembly inside C

Floating point:

- *Intrinsics for basic operations (add, mult, ...)*
- *Intrinsics for math functions: log, sin, ...*

Our introduction is based on icc

- *Almost all intrinsics work with gcc and Visual Studio (VS)*
- *Some language extensions are icc (or even VS) specific*

### Number of intrinsics

ISA	Count
MMX	124
SSE	154
SSE2	236
SSE3	11
SSSE3	32
SSE41	61
SSE42	19
AVX	188
AVX2	191
AVX-512	3857
FMA	32
KNC	601
SVML	406
<hr/>	
2019	

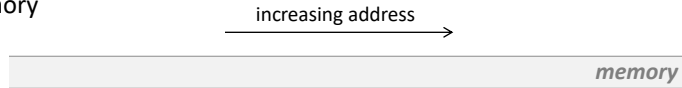
16

16



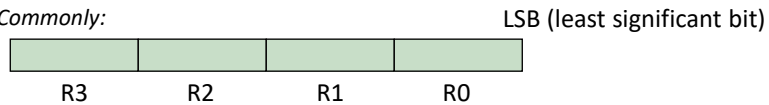
## Visual Conventions We Will Use

Memory



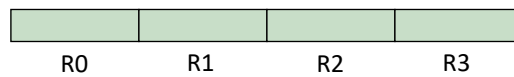
Registers

- Commonly:



- We will use

**LSB**



17

17

## AVX Intrinsics (Focus Floating Point)

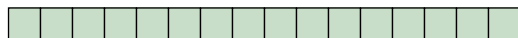
Data types

```

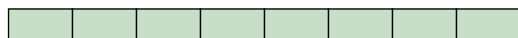
__m256 f; // = {float f0, f1, f2, f3, f4, f5, f6, f7}
__m256d d; // = {double d0, d1, d3, d4}
__m256i i; // 32 8-bit, 16 16-bit, 8 32-bit, or 4 64-bit
    
```



ints



ints



ints or floats



ints or doubles

18

18

# AVX Intrinsics (Focus Floating Point)

## Instructions

- Naming convention: `_mm256_<intrin_op>_<suffix>`
- Example:

```
// a is 32-byte aligned
double a[4] = {1.0, 2.0, 3.0, 4.0};
__m256d t = _mm256_load_pd(a);
```

*p: packed*  
*d: double precision*

LSB 

1.0	2.0	3.0	4.0
-----	-----	-----	-----

- Same result as

```
__m256d t = _mm256_set_pd(4.0, 3.0, 2.0, 1.0)
```

19

19

# AVX Intrinsics

## Native instructions (one-to-one with assembly)

```
_mm256_load_pd() ↔ vmovapd
_mm256_add_pd() ↔ vaddpd
_mm256_mul_pd() ↔ vmulpd
...
```

## Multi instructions (map to several assembly instructions)

```
_mm256_set_pd()
_mm256_set1_pd()
...
```

## Macros and helpers

```
_MM_SHUFFLE()
...
```

20

20

## Intel Intrinsic Guide

### [Intel Intrinsic Guide](#)

Great resource to quickly find the right intrinsics

Has latency and throughput information for many instructions

Shows whether intrinsic translated to one assembly op or several

Can also be used to search for assembly ops

*Note: Intel measures throughput in cycles, i.e., really shows 1/throughput.  
Example: Intel throughput 0.33 means throughput is 3 ops/cycle.*

*We show throughput in ops/cycle.*

21

21

## What Are the Main Issues?

Alignment is important (256 bit = 32 byte)

You need to code explicit loads and stores

Overhead through shuffles

Not all instructions in SSE (AVX) have a counterpart in AVX (or AVX-512)

*Reason: Building in hardware an AVX unit by pasting together 2 SSE units is easy (e.g., vaddpd is just 2 parallel addpd); if SSE “lanes” need to be crossed it is expensive*

22

22

## SSE vs. AVX vs. AVX-512

	SSE	AVX	AVX-512
float, double	4-way, 2-way	8-way, 4-way	16-way, 8-way
register	16 x 128 bits: %xmm0 - %xmm15	16 x 256 bits: %ymm0 - %ymm15 <i>The lower halves are the %xmms</i>	32 x 512 bits: %zmm0 - %zmm31 <i>The lower halves are the %ymms</i>
assembly ops	addps, mulpd, ...	vaddps, vmulpd	vaddps, vmulpd
intrinsics data type	__m128, __m128d	__m256, __m256d	__m512, __m512d
intrinsics instructions	_mm_load_ps, _mm_add_pd, ...	_mm256_load_ps, _mm256_add_pd	_mm512_load_ps, _mm512_add_pd

*Mixing SSE and AVX may incur penalties*

23

23

## A Note on AVX-512

Some instructions names (e.g., vaddps) are the same as for AVX2 but they use a different encoding ([EVEX prefix](#)) and are thus an extension. Doing so

- Enables the use of the 16 additional registers zmm16–31, including there ymm16–31 and xmm16–31 and scalar parts.
- Scalar add on these registers: vaddss
- Enables additional functionality, e.g., vaddps and vaddss support masking

24

24

# AVX Intrinsic

Load and store

Constants

Arithmetic

Comparison

Conversion

Shuffles

25

25

# Loads and Stores

Intrinsic Name	Operation	Corresponding AVX Instructions
<code>_mm256_load_pd</code>	Load four double values, address aligned	VMOVAPD ymm, mem
<code>_mm256_loadu_pd</code>	Load four double values, address unaligned	VMOVUPD ymm, mem
<code>_mm256_maskload_pd</code>	Load four double values using mask	VMASKMOVPD ymm, mem
<code>_mm256_broadcast_sd</code>	Load one double value into all four words	VBROADCASTSD ymm, mem
<code>_mm256_broadcast_pd</code>	Load a pair of double values into the lower and higher part of vector.	VBROADCASTSD ymm, mem
<code>_mm256_i64gather_pd</code>	Load double values from memory using indices.	VGATHERPD ymm, mem, ymm

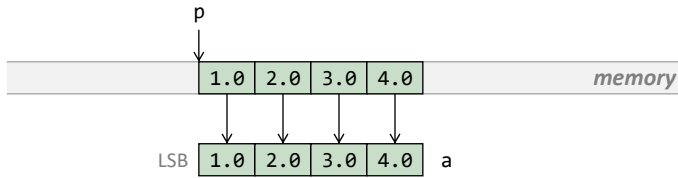
Intrinsic Name	Operation	Corresponding AVX Instruction
<code>_mm256_set1_pd</code>	Set all four words with the same value	Composite
<code>_mm256_set_pd</code>	Set four values	Composite
<code>_mm256_setr_pd</code>	Set four values, in reverse order	Composite
<code>_mm256_setzero_pd</code>	Clear all four values	VXORPD
<code>_mm256_set_m128d</code>	Set lower and higher 128-bit parts	VINSERTF128

Tables show only most important instructions in category

26

26

# Loads and Stores



Skylake (load):  
 Lat = 7  
 Tp = 2  
 Skylake (store):  
 Lat = 5  
 Tp = 1

```
a = _mm256_load_pd(p); // p 32-byte aligned
```

```
a = _mm256_loadu_pd(p); // p not aligned
```

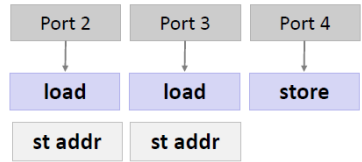
*May incur a significant performance penalty*

load\_pd on unaligned pointer: seg fault

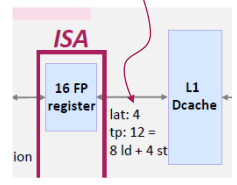
→ blackboard

# Side Note: Load and Stores on Skylake

Skylake (load):  
 Lat = 7  
 Tp = 2  
 Skylake (store):  
 Lat = 5  
 Tp = 1



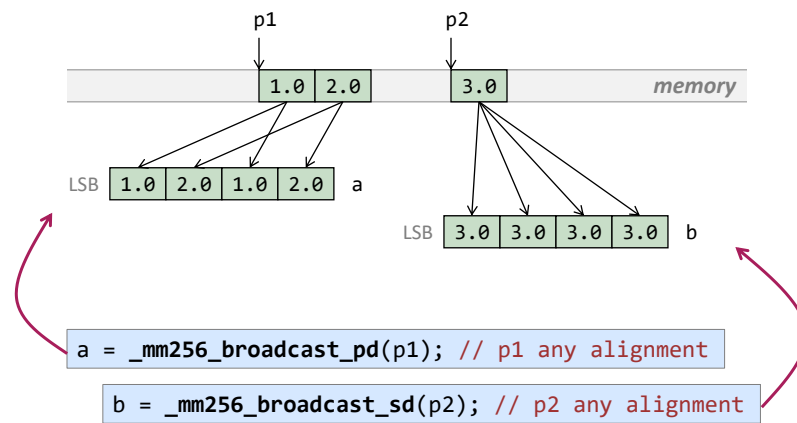
fastest possible latency  
 not available for every load



*Different ways we saw the throughput*  
 per cycle: two loads of AVX vectors = 8 doubles

# Loads and Stores

Skylake:  
Lat = -  
Tp = -

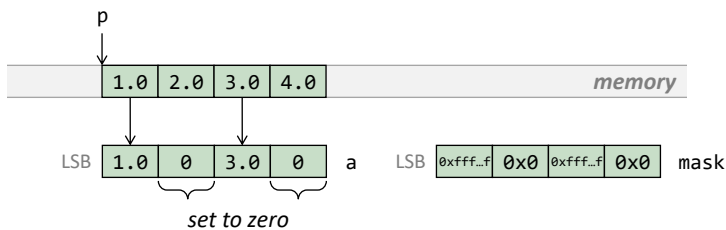


29

29

# Loads and Stores

Skylake:  
Lat = -  
Tp = -



```

a = _mm256_maskload_pd(p, mask); // p any alignment
    
```

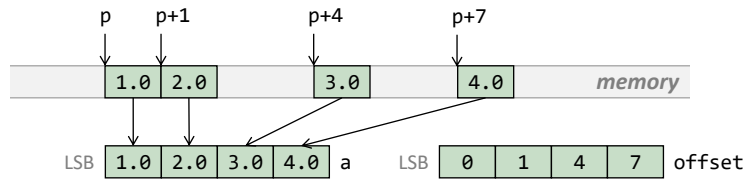
↑  
\_m256i

30

30

# Loads and Stores

Skylake:  
Lat = -  
Tp = -



```
a = _mm256_i64gather_pd(p, offset, 8); // p any alignment
```

scale = {1,2,4,8}  
above: scale = 8 = size of double

31

# Stores Analogous to Loads

Intrinsic Name	Operation	Corresponding AVX Instruction
_mm256_store_pd	Store four values, address aligned	VMOVAPD
_mm256_storeu_pd	Store four values, address unaligned	VMOVUPD
_mm256_maskstore_pd	Store four values using mask	VMASKMOVPD
_mm256_storeu2_m128d	Store lower and higher 128-bit parts into different memory locations	Composite
_mm256_stream_pd	Store values without caching, address aligned	VMOVNTPD

Tables show only most important instructions in category

32



## Constants

LSB	<table border="1"><tr><td>1.0</td><td>2.0</td><td>3.0</td><td>4.0</td></tr></table>	1.0	2.0	3.0	4.0	a	<code>a = _mm256_set_pd(4.0, 3.0, 2.0, 1.0);</code>
1.0	2.0	3.0	4.0				
LSB	<table border="1"><tr><td>1.0</td><td>1.0</td><td>1.0</td><td>1.0</td></tr></table>	1.0	1.0	1.0	1.0	b	<code>b = _mm256_set1_pd(1.0);</code>
1.0	1.0	1.0	1.0				
LSB	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	c	<code>c = _mm256_setzero_pd();</code>
0	0	0	0				

→ blackboard

33

## Arithmetic

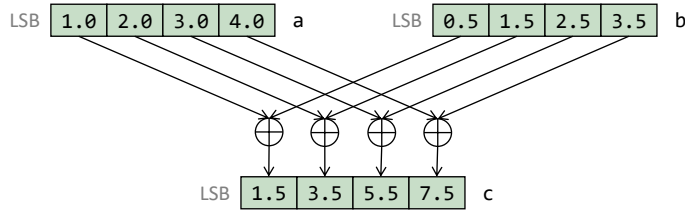
Intrinsic Name	Operation	Corresponding AVX Instruction
<code>_mm256_add_pd</code>	Addition	VADDPD
<code>_mm256_sub_pd</code>	Subtraction	VSUBPD
<code>_mm256_addsub_pd</code>	Alternatively add and subtract	VADDSUBPD
<code>_mm256_hadd_pd</code>	Half addition	VHADDPD
<code>_mm256_hsub_pd</code>	Half subtraction	VHSUBPD
<code>_mm256_mul_pd</code>	Multiplication	VMULPD
<code>_mm256_div_pd</code>	Division	VDIVPD
<code>_mm256_sqrt_pd</code>	Squared Root	VSQRTPD
<code>_mm256_max_pd</code>	Computes Maximum	VMAXPD
<code>_mm256_min_pd</code>	Computes Minimum	VMINPD
<code>_mm256_ceil_pd</code>	Computes Ceil	VROUNDPD
<code>_mm256_floor_pd</code>	Computes Floor	VROUNDPD
<code>_mm256_round_pd</code>	Round	VROUNDPD
<code>_mm256_dp_ps</code>	Single precision dot product	VDPPS
<code>_mm256_fmadd_pd</code>	Fused multiply-add	VFMAADD132pd
<code>_mm256_fmsub_pd</code>	Fused multiply-subtract	VFMSUB132pd
<code>_mm256_fmaddsub_pd</code>	Alternatively fmadd, fmsub	VFMAADDSUB132pd

Tables show only most important instructions in category

34

34

# Arithmetic



```
c = _mm256_add_pd(a, b);
```

*analogous:*

```
c = _mm256_sub_pd(a, b);
```

```
c = _mm256_mul_pd(a, b);
```

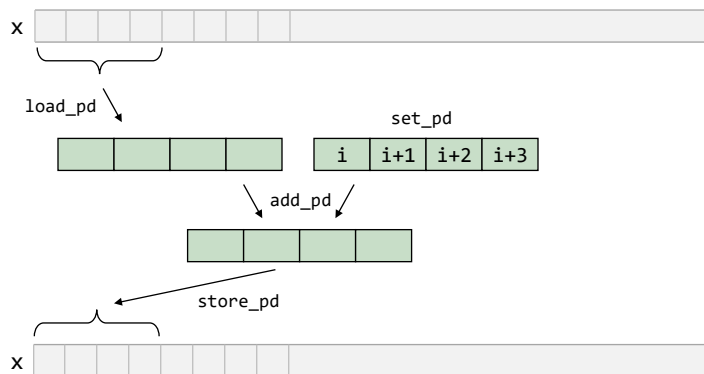
→ blackboard

35

# Example

```
void addindex(double *x, int n) {
    for (int i = 0; i < n; i++)
        x[i] = x[i] + i;
}
```

Vectorization by drawing:



36

36

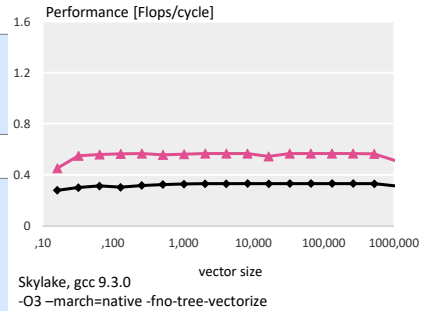
## Example

```
void addindex(double *x, int n) {
    for (int i = 0; i < n; i++)
        x[i] = x[i] + i;
}
```

```
#include <immintrin.h>

// n a multiple of 4, x is 32-byte aligned
void addindex_vec1(double *x, int n) {
    __m256d index, x_vec;

    for (int i = 0; i < n; i+=4) {
        x_vec = _mm256_load_pd(x+i);
        index = _mm256_set_pd(i+3, i+2, i+1, i); // load 4 doubles
        x_vec = _mm256_add_pd(x_vec, index); // create vector with indexes
        _mm256_store_pd(x+i, x_vec); // add the two
    } // store back
}
```



Is this the best solution?

**No! `_mm256_set_pd` may be too expensive**

37

37

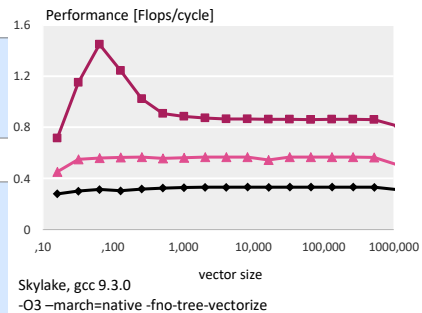
## Example

```
void addindex(double *x, int n) {
    for (int i = 0; i < n; i++)
        x[i] = x[i] + i;
}
```

```
#include <immintrin.h>

// n a multiple of 4, x is 32-byte aligned
void addindex_vec2(double *x, int n) {
    __m256d x_vec, ind;

    ind = _mm256_set_pd(3, 2, 1, 0);
    incr = _mm256_set1_pd(4);
    for (int i = 0; i < n; i+=4) {
        x_vec = _mm256_load_pd(x+i);
        x_vec = _mm256_add_pd(x_vec, ind); // load 4 doubles
        ind = _mm256_add_pd(ind, incr); // add the two
        _mm256_store_pd(x+i, x_vec); // update ind
    } // store back
}
```

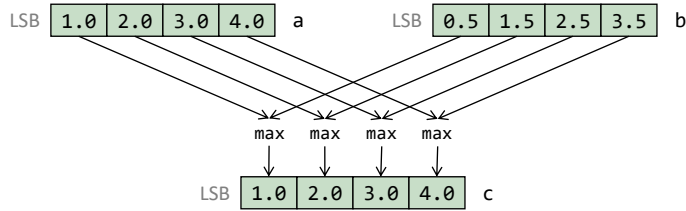


Code style helps with performance! **Why?**

38

38

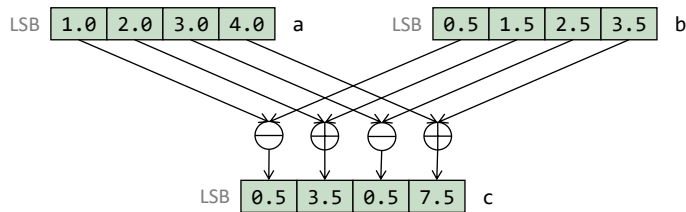
# Arithmetic



```
c = _mm256_max_pd(a, b);
```

39

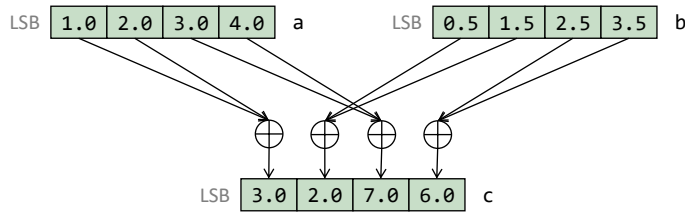
# Arithmetic



```
c = _mm256_addsub_pd(a, b);
```

40

# Arithmetic



```
c = _mm256_hadd_pd(a, b);
```

*analogous:*

```
c = _mm256_hsub_pd(a, b);
```

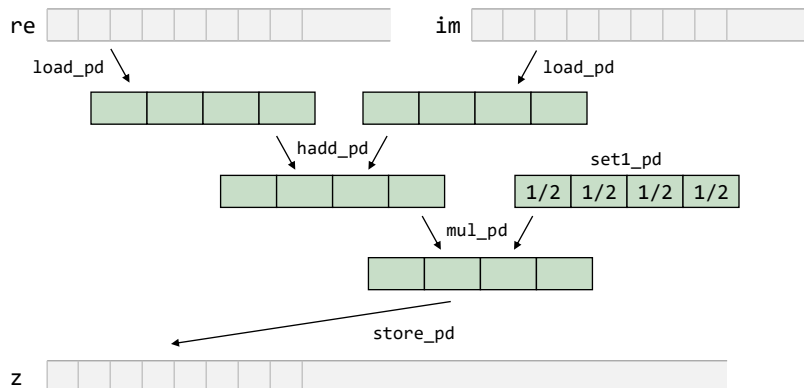
*Does not cross between 128-bit lanes*

→ blackboard

41

# Example

```
// n is even, low pass filter on complex numbers
// output z is in interleaved format
void cfp(double *re, double *im, double *z, int n) {
    for (int i = 0; i < n; i+=2) {
        z[i] = (re[i] + re[i+1])/2;
        z[i+1] = (im[i] + im[i+1])/2;
    }
}
```

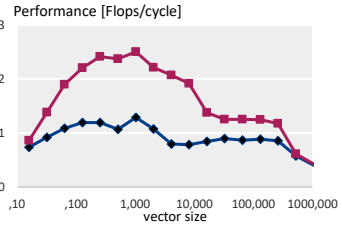


42

42

# Example

```
// n is even, low pass filter on complex numbers
// output z is in interleaved format
void clp(double *re, double *im, double *z, int n) {
    for (int i = 0; i < n; i+=2) {
        z[i] = (re[i] + re[i+1])/2;
        z[i+1] = (im[i] + im[i+1])/2;
    }
}
```



```
#include <immintrin.h>

// n a multiple of 4, re, im, z are 32-byte aligned
void clp_vec(double *re, double *im, double *z, int n) {
    __m256d half, v1, v2, avg;

    half = _mm256_set1_pd(0.5); // set vector to all 0.5
    for(int i = 0; i < n; i+=4) {
        v1 = _mm256_load_pd(re+i); // load 4 doubles of re
        v2 = _mm256_load_pd(im+i); // load 4 doubles of im
        avg = _mm256_hadd_pd(v1, v2); // add pairs of doubles
        avg = _mm256_mul_pd(avg, half); // multiply with 0.5
        _mm256_store_pd(z+i, avg); // save result
    }
}
```

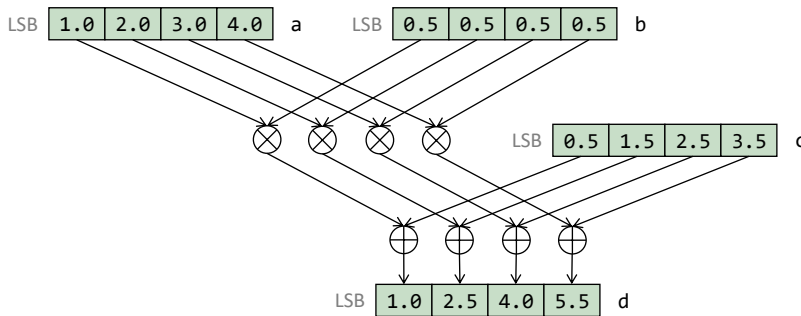
Skylake, gcc 9.3.0  
-O3 -march=native -fno-tree-vectorize

43

43

# Arithmetic (FMA)

Skylake:  
Lat = 4  
Tp = 2



```
d = _mm256_fmadd_pd(a, b, c);
```

analogous:

```
d = _mm256_fmsub_pd(a, b, c);
```

scalar FMA (128 bit):

```
d = _mm_fmadd_sd(a, b, c);
```

44

44

# Example

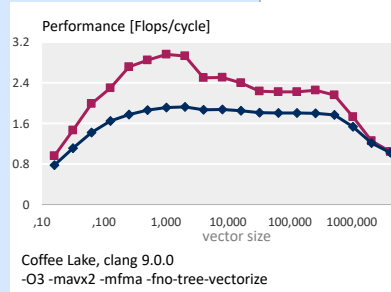
```
// y = a + x^2 on complex numbers, a is constant
void complex_square(double *a, double *x, double *y, int n) {
    for (int i = 0; i < n; i+=2) {
        y[i] = a[0] + x[i]*x[i] - x[i+1]*x[i+1];
        y[i+1] = a[1] + 2.0*x[i]*x[i+1];
    }
}
```

```
#include <immintrin.h>

void complex_square_fma(double *a, double *x, double *y, int n) {
    __m128d re, im, a_re, a_im, two;

    two = _mm_set_sd(2.0);
    a_re = _mm_set_sd(a[0]);
    a_im = _mm_set_sd(a[1]);
    for (int i = 0; i < n; i+=2) {
        x_re = _mm_load_sd(x+i);
        x_im = _mm_load_sd(x+i+1);
        re = _mm_fmadd_sd(x_re, x_re, a_re);
        re = _mm_fnmadd_sd(x_im, x_im, re);
        im = _mm_mul_sd(two, x_re);
        im = _mm_fmadd_sd(im, x_im, a_im);
        _mm_store_sd(y+i, re);
        _mm_store_sd(y+i+1, im);
    }
}
```

not vectorized!



45

45

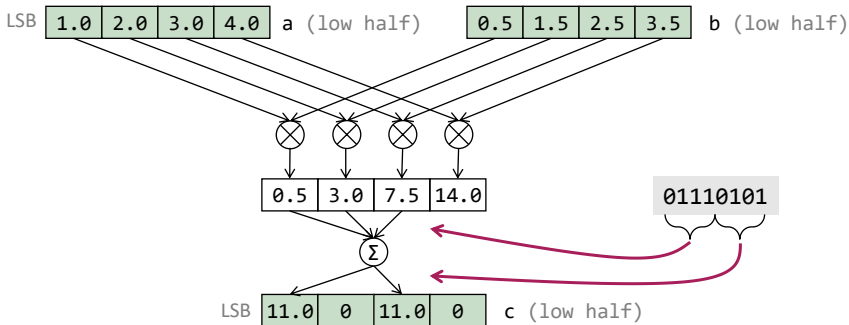
# Arithmetic

Skylake:  
Lat = 13  
Tp = 2/3

```
__m256 _mm256_dp_ps(__m256 a, __m256 b, const int mask) __mm256_dp_pd does not exist
```

Computes the pointwise product of a and b and writes a selected sum of the resulting numbers into selected elements of c; the others are set to zero. The selections are encoded in the mask. (Only for floats)

Example: mask = 117 = 01110101



Same is done for the upper half

46

46

# Comparisons

Intrinsic Name	Macro for operation	Operation
_mm256_cmp_pd (VCMPPD)	_CMP_EQ_OQ	Equal
	_CMP_EQ_UQ	Equal (unordered)
	_CMP_GE_OQ	Greater Than or Equal
	_CMP_GT_OQ	Greater Than
	_CMP_LE_OQ	Less Than or Equal
	_CMP_LT_OQ	Less Than
	_CMP_NEQ_OQ	Not Equal
	_CMP_NEQ_UQ	Not Equal (unordered)
	_CMP_NGE_UQ	Not Greater Than or Equal (unordered)
	_CMP_NGT_UQ	Not Greater Than (unordered)
	_CMP_NLE_UQ	Not Less Than or Equal (unordered)
	_CMP_NLT_UQ	Not Less Than (unordered)
	_CMP_TRUE_UQ	True (unordered)
	_CMP_FALSE_OQ	False
	_CMP_ORD_Q	Ordered
	_CMP_UNORD_Q	Unordered

The last two letters (U or O, Q or S) only have to do with the handling of NaNs.  
E.g., O (ordered): NaN is not equal to 1.0; U (unordered): NaN is equal to 1.0.

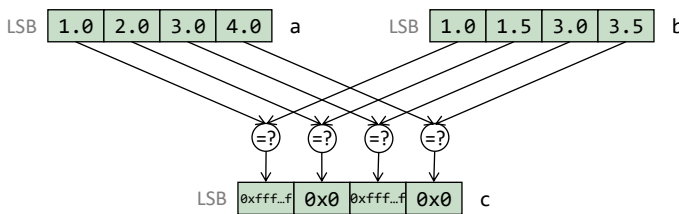
Tables show only most important instructions in category

47

47

# Comparisons

Skylake:  
Lat = 4  
Tp = 2



```
c = _mm256_cmp_pd(a, b, _CMP_EQ_OQ);
```

**analogous:**

```
c = _mm256_cmp_pd(a, b, _CMP_GE_OQ);
```

```
c = _mm256_cmp_pd(a, b, _CMP_LT_OQ);
```

etc.

**Each field:**  
0xffff...f if true  
0x0 if false

Return type: \_\_m256d

→ blackboard

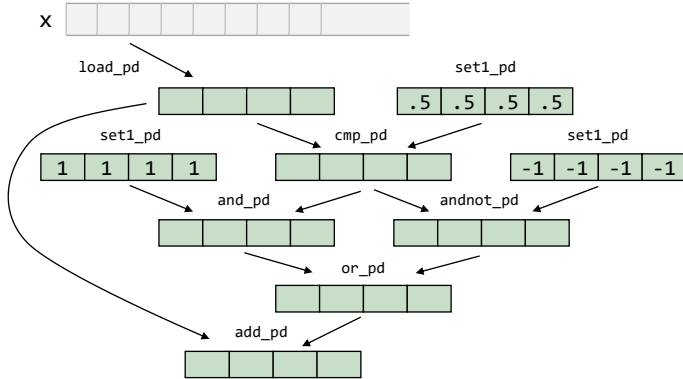
48



## Example

```
void fcond(double *x, size_t n) {
    int i;

    for(i = 0; i < n; i++) {
        if(x[i] > 0.5)
            x[i] += 1.;
        else x[i] -= 1.;
    }
}
```



49

49

## Example

```
void fcond(double *x, size_t n) {
    int i;

    for(i = 0; i < n; i++) {
        if(x[i] > 0.5)
            x[i] += 1.;
        else x[i] -= 1.;
    }
}
```

```
#include <xmmintrin.h>

void fcond_vec1(double *x, size_t n) {
    int i;
    __m256d vt, vmask, vp, vm, vr, ones, mones, thresholds;

    ones      = _mm256_set1_pd(1.);
    mones     = _mm256_set1_pd(-1.);
    thresholds = _mm256_set1_pd(0.5);
    for(i = 0; i < n; i+=4) {
        vt      = _mm256_load_pd(x+i);
        vmask   = _mm256_cmp_pd(vt, thresholds, _CMP_GT_OQ);
        vp      = _mm256_and_pd(vmask, ones);
        vm      = _mm256_andnot_pd(vmask, mones);
        vr      = _mm256_add_pd(vt, _mm256_or_pd(vp, vm));
        _mm256_store_pd(x+i, vr);
    }
}
```

50

50

# Vectorization

=



Picture: [www.druckundbestell.de](http://www.druckundbestell.de)

51

## Conversion

Intrinsic Name	Operation	Corresponding AVX Instruction
<code>_mm256_cvtepi32_pd</code>	Convert from 32-bit integer	<code>VCVTDQ2PD</code>
<code>_mm256_cvtepi32_ps</code>	Convert from 32-bit integer	<code>VCVTDQ2PS</code>
<code>_mm256_cvtpd_epi32</code>	Convert to 32-bit integer	<code>VCVTPD2DQ</code>
<code>_mm256_cvtps_epi32</code>	Convert to 32-bit integer	<code>VCVTPS2DQ</code>
<code>_mm256_cvtps_pd</code>	Convert from floats	<code>VCVTPS2PD</code>
<code>_mm256_cvtpd_ps</code>	Convert to floats	<code>VCVTPD2PS</code>
<code>_mm256_cvttpd_epi32</code>	Convert to 32-bit integer with truncation	<code>VCVTPD2DQ</code>
<code>_mm256_cvtsd_f64</code>	Extract	<code>MOVSD</code>
<code>_mm256_cvtsd_f32</code>	Extract	<code>MOVSS</code>

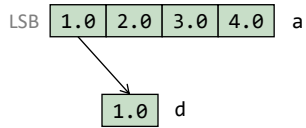
Tables show only most important instructions in category

52

52

## Conversion

```
double __mm256_cvtsd_f64(__m256d a)
```



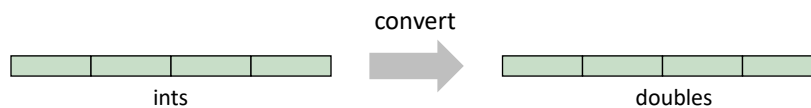
```
double d;  
d = __mm_cvtsd_f64(a);
```

53

53

## Conversion

```
__m256d __mm256_cvtepi32_pd(__m128i a)
```



See also:

```
__m256d __mm256_cvtepi64_pd(__m256i a)
```

```
__m256d __mm256_cvtepu32_pd(__m256i a)
```

```
__m256d __mm256_cvtepu64_pd(__m256i a)
```

54

54

# Shuffles

Intrinsic Name	Operation	Corresponding AVX Instruction
<code>_mm256_unpackhi_pd</code>	Unpack High	VUNPCKHPD
<code>_mm256_unpacklo_pd</code>	Unpack Low	VUNPCKLPD
<code>_mm256_movemask_pd</code>	Create four-bit mask	VMOVMSKPD
<code>_mm256_movedup_pd</code>	Duplicates	VMOVDUP
<code>_mm256_blend_pd</code>	Selects data from 2 sources using constant mask	VBLENDPD
<code>_mm256_blendv_pd</code>	Selects data from 2 sources using variable mask	VBLENDVPD
<code>_mm256_insertf128_pd</code>	Insert 128-bit value into packed array elements selected by index.	VINSERTF128
<code>_mm256_extractf128_pd</code>	Extract 128-bits selected by index.	VEXTRACTF128
<code>_mm256_shuffle_pd</code>	Shuffle	VSHUFPD
<code>_mm256_permute_pd</code>	Permute	VPERMILPD
<code>_mm256_permute4x64_pd</code>	Permute 64-bits elements	VPERMPD
<code>_mm256_permute2f128_pd</code>	Permute 128-bits elements	VPERM2F128

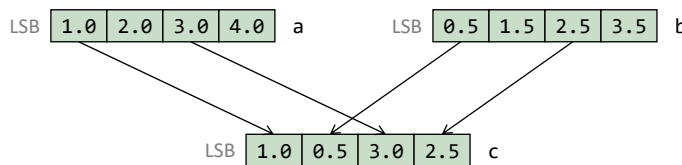
Tables show only most important instructions in category

55

55

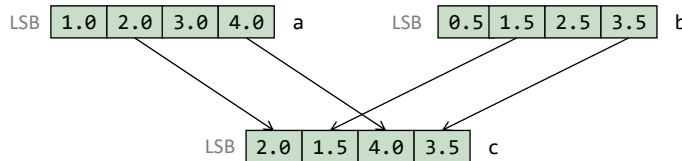
# Shuffles

Skylake:  
Lat = 1  
Tp = 1



`c = _mm256_unpacklo_pd(a, b);`

*Does not cross between 128-bit lanes*



`c = _mm256_unpackhi_pd(a, b);`

→ blackboard

56

# Shuffles

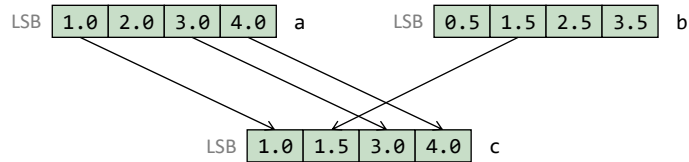
```
__m256d _mm256_blendv_pd(__m256d a, __m256d b, __m256d mask)
```

Result is filled in each position by an element of a or b in the same position as specified by mask

**Example:** LSB 

0x0	0xffff..f	0x0	0x0
-----	-----------	-----	-----

 mask



see also `_mm256_blend_pd`:  
same with integer mask, Tp = 3!

## Example (Continued From Before)

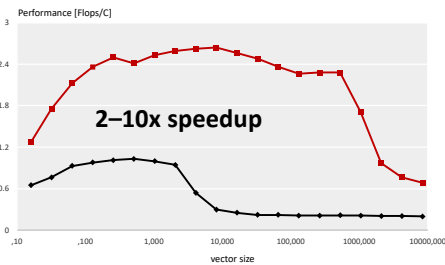
```
void fcond(double *x, size_t n) {
    int i;

    for(i = 0; i < n; i++) {
        if(x[i] > 0.5)
            x[i] += 1.;
        else x[i] -= 1.;
    }
}
```

```
#include <immintrin.h>

void fcond_vec2(double *x, size_t n) {
    int i;
    __m256d vt, vmask, vp, vm, vr, ones, mones, thresholds;

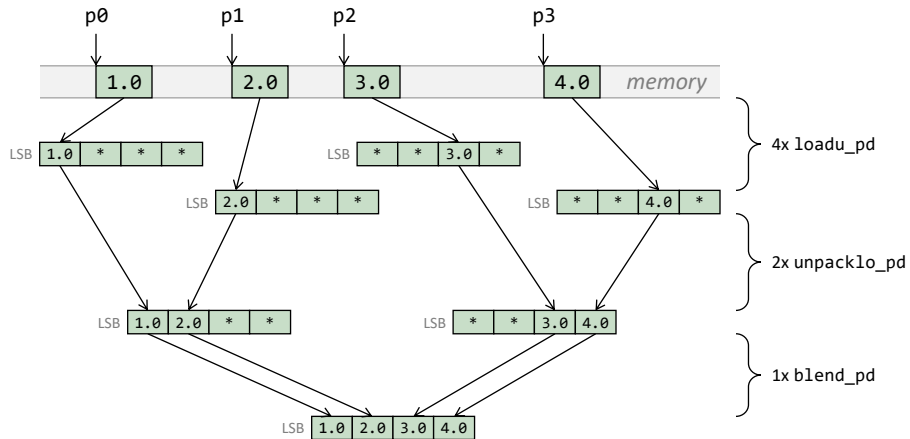
    ones = _mm256_set1_pd(1.);
    mones = _mm256_set1_pd(-1.);
    thresholds = _mm256_set1_pd(0.5);
    for(i = 0; i < n; i+=4) {
        vt = _mm256_load_pd(x+i);
        vmask = _mm256_cmp_pd(vt, thresholds, _CMP_GT_OQ);
        vb = _mm256_blendv_pd(mones, ones, vmask);
        vr = _mm256_add_pd(vt, vb);
        _mm256_store_pd(x+i, vr);
    }
}
```



Skylake, gcc 9.3.0  
-O3 -march=native -fno-tree-vectorize

## Example: Loading 4 Doubles from Arbitrary Memory Locations

Assumes all values are within one array



*7 instructions, this is one way of doing it*

59

59

## Code For Previous Slide

```
#include <immintrin.h>

__m256d LoadArbitrary(double *p0, double *p1, double *p2, double *p3) {
    __m256d a, b, c, d, e, f;

    a = _mm256_loadu_pd(p0);
    b = _mm256_loadu_pd(p1);
    c = _mm256_loadu_pd(p2-2);
    d = _mm256_loadu_pd(p3-2);
    e = _mm256_unpacklo_pd(a, b);
    f = _mm256_unpacklo_pd(c, d);
    return _mm256_blend_pd(e, f, 0b1100);
}
```

Example compilation:

```
vmovupd  ymm0, [rdi]
vmovupd  ymm1, [-16+rdx]
vunpcklpd ymm2, ymm0, [rsi]
vunpcklpd ymm3, ymm1, [-16+rcx]
vblendpd ymm0, ymm2, ymm3, 12
```

} no intrinsic for this instruction  
(Nov 2019)

60

60

## Example: Loading 4 Doubles from Arbitrary Memory Locations (cont'd)

Whenever possible avoid the previous situation

Restructure algorithm and use the aligned  
`_mm256_load_pd()`

61

61

## Example: Loading 4 Doubles from Arbitrary Memory Locations (cont'd)

Other possibility

```
__m256 vf;  
vf = _mm256_set_pd(*p3, *p2, *p1, *p0);
```

Example compilation:

```
vmovsd xmm0, [rdi]  
vmovsd xmm1, [rdx]  
vmovhpd xmm2, xmm0, [rsi] // SSE register xmm2 written  
vmovhpd xmm3, xmm1, [rcx]  
vinsertf128 ymm0, ymm2, xmm3, 1 // accessed as ymm2
```

`vmovhpd` cannot be expressed as intrinsic (Nov 2019) but `movupd` can  
(`_mm_loadh_pd`)

62

62

## Example: Loading 4 Doubles from Arbitrary Memory Locations (cont'd)

Example compilation:

```
vmovsd xmm0, [rdi]
vmovsd xmm1, [rdx]
vmovhpd xmm2, xmm0, [rsi] // SSE register xmm2 written
vmovhpd xmm3, xmm1, [rcx]
vinsertf128 ymm0, ymm2, xmm3, 1 // accessed as ymm2
```

Written in intrinsics (reverse-engineered):

```
#include <immintrin.h>

__m256d myArbitraryLoad2(double *a, double *b, double *c, double *d) {
    __m128d t1, t2, t3, t4;
    __m256d t5;

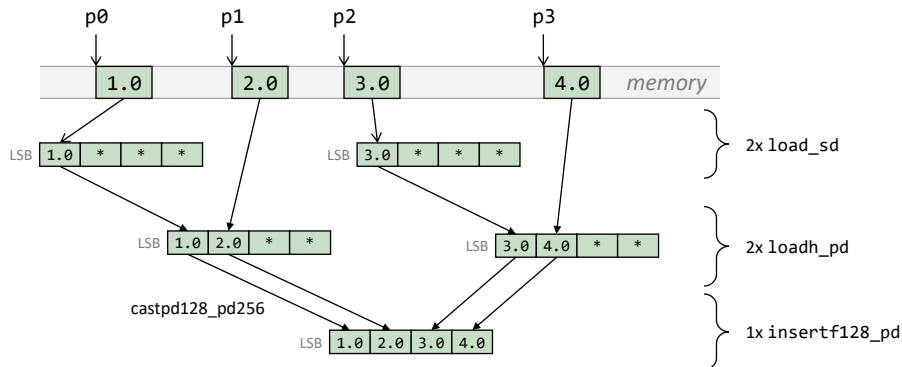
    t1 = _mm_load_sd(a); // SSE
    t2 = _mm_loadh_pd(t1, b); // SSE
    t3 = _mm_load_sd(c); // SSE
    t4 = _mm_loadh_pd(t3, d); // SSE
    t5 = _mm256_castpd128_pd256(t2); // cast __m128d -> __m256d
    return _mm256_insertf128_pd(t5, t4, 1);
}
```

63

63

## Example: Loading 4 Doubles from Arbitrary Memory Locations

Picture for previous slide (this solution always works):



64

64



## Example: Loading 4 Doubles from Arbitrary Memory Locations (cont'd)

Do not do this (why?):

```
__declspec(align(32)) double g[4];
__m256d vf;

g[0] = *p0;
g[1] = *p1;
g[2] = *p2;
g[3] = *p3;
vf = _mm256_load_pd(g);
```

65

65

## Shuffles

Skylake:

Lat = 1

Tp = 1

```
__m256d _mm256_shuffle_pd(__m256d a, __m256d b, const int mask)
```

LSB 

1.0	2.0	3.0	4.0
-----	-----	-----	-----

 a

LSB 

0.5	1.5	2.5	3.5
-----	-----	-----	-----

 b

LSB 

c0	c1	c2	c3
----	----	----	----

 c

*a0 or a1*

```
c0 = mask.bit0 ? a1 : a0
c1 = mask.bit1 ? b1 : b0
c2 = mask.bit2 ? a3 : a2
c3 = mask.bit3 ? b3 : b2
```

*Does not cross between 128-bit lanes*

66

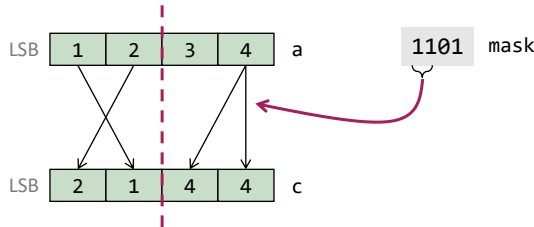
66

# Shuffles

```
__m256d _mm256_permute_pd(__m256d a, int mask)
```

Shuffle elements within 128-bit lanes.

*Example:*



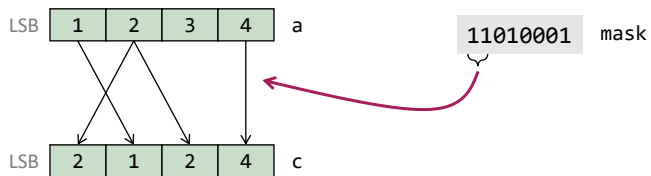
*Does not cross between 128-bit lanes*

# Shuffles

```
__m256d _mm256_permute4x64_pd(__m256d a, int mask)
```

Result is filled in each position by any element of a, as specified by mask

*Example:*



*Somewhat more expensive due to shuffle between 128-bit lanes*

# Apple M1 Processor

ISA: ARMv8.4 with 128-bit Neon vector instructions

2-way double (float64x2\_t), 4-way single (float32x4\_t),  
8-way half (float16x8\_t)

Some common intrinsics: [\(see intrinsics website\)](#)

- vaddq\_f64
- vmulq\_f64
- vld1q\_f64 (*load vector of 2 doubles*)
- vst1q\_f64

Example code:

```
#include <arm_neon.h>
float64x2_t a, b, c;
.....
c = vaddq_f64(a, b);
```

69

69

# SIMDe Library for M1

[Easy to use library](#): Provides header file (e.g., x86/avx) to make Intel's SIMD intrinsics available on M1.

Define SIMDE\_ENABLE\_NATIVE\_ALIASES before the header to use the same names as intel's intrinsics, e.g., \_mm256\_add\_pd, (otherwise it must be prefixed with "simde\_").

Define SIMDE\_ARM\_NEON\_A64V8\_NATIVE to specify that the native platform supports NEON and the library uses those intrinsics.

Example of internal \_mm256\_add\_pd implementation provided by library (simplified for readability):

```
__m256d _mm256_add_pd (__m256d a, __m256d b) {
    ...
    r_.m128d[0] = _mm_add_pd(a_.m128d[0], b_.m128d[0]);
    r_.m128d[1] = _mm_add_pd(a_.m128d[1], b_.m128d[1]);
    return __m256d_from_private(r_);
}

__m128d _mm_add_pd (__m128d a, __m128d b) {
    ...
    r_.neon_f64 = vaddq_f64(a_.neon_f64, b_.neon_f64); // NEON intrinsic
    return __m256d_from_private(r_);
}
```

70

70

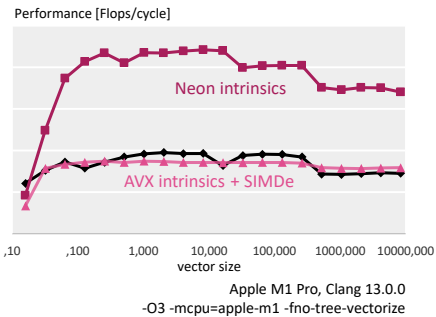
## SIMDe Library vs. Neon Intrinsics

```
void fcond(double *x, size_t n) {
    int i;

    for(i = 0; i < n; i++) {
        if(x[i] > 0.5)
            x[i] += 1.;
        else x[i] -= 1.;
    }
}
```

```
#include <arm_neon.h>
void fcond_neon(double* x, int n) {
    float64x2_t vt1, vb1, vr1, vmask1;
    float64x2_t vt2, vb2, vr2, vmask2;
    float64x2_t ones = vdupq_n_f64(1.);
    float64x2_t mones = vdupq_n_f64(-1.);
    float64x2_t thresholds = vdupq_n_f64(0.5);

    for(int i = 0; i < n; i+=4) {
        vt1 = vld1q_f64(x+i);
        vt2 = vld1q_f64(x+i+2);
        vmask1 = vcgtq_f64(vt1, thresholds);
        vmask2 = vcgtq_f64(vt2, thresholds);
        vb1 = vbslq_f64(vmask1, ones, mones);
        vb2 = vbslq_f64(vmask2, ones, mones);
        vr1 = vaddq_f64(vt1, vb1);
        vr2 = vaddq_f64(vt2, vb2);
        vst1q_f64(x+i, vr1);
        vst1q_f64(x+i+2, vr2); }
}
```



No guarantee that translation with SIMDe is close to optimal in more complicated cases

71

71

## Vectorization With Intrinsics: Key Points

Use aligned loads and stores as much as possible

Minimize shuffle instructions

Minimize use of suboptimal arithmetic instructions.

E.g., add\_pd has higher throughput than hadd\_pd

Be aware of available instructions ([intrinsics guide!](#)) and their performance

72

72

## SIMD Extensions and AVX

AVX intrinsics

*Compiler vectorization*

References:

[Intel icc manual](#) (look for auto vectorization)

73

73

## Compiler Vectorization

Compiler flags

Aliasing

Proper code style

Alignment

74

74

# How Do I Know the Compiler Vectorized?

vec-report

Look at assembly: `vmulpd`, `vaddpd`, `xxxpd`

Generate assembly with source code annotation:

- *Visual Studio + icc: /Fas*
- *icc on Linux/Mac: -S*

75

75

## Example

```
void myadd(double *a, double *b, const int n) {  
    for (int i = 0; i < n; i++)  
        a[i] = a[i] + b[i];  
}
```

**unvectorized:** /Qvec-

```
<more>  
;;; a[i] = a[i] + b[i];  
vmovsd    xmm0, DWORD PTR [rcx+rax*4]  
vaddsd    xmm0, DWORD PTR [rdx+rax*4]  
vmovsd    DWORD PTR [rcx+rax*4], xmm0  
<more>
```

**vectorized:**

```
<more>  
;;; a[i] = a[i] + b[i];  
vmovsd    xmm0, DWORD PTR [rcx+r11*4]  
vaddsd    xmm0, DWORD PTR [rdx+r11*4]  
vmovsd    DWORD PTR [rcx+r11*4], xmm0  
...  
vmovupd    ymm0, YMMWORD PTR [rdx+r10*4]  
vmovupd    ymm1, YMMWORD PTR [16+rdx+r10*4]  
vaddpd    ymm0, ymm0, YMMWORD PTR [rcx+r10*4]  
vaddpd    ymm1, ymm1, YMMWORD PTR [16+rcx+r10*4]  
vmovupd    YMMWORD PTR [rcx+r10*4], ymm0  
vmovupd    YMMWORD PTR [16+rcx+r10*4], ymm1  
<more>
```

} why this?

} why everything twice?

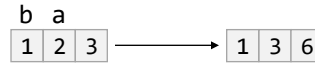
76

76

## Are These Programs Equivalent?

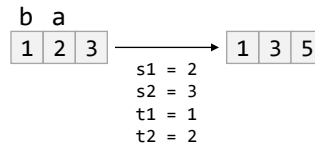
P1:

```
for (i = 0; i < n; i++) // n even
  a[i] = a[i] + b[i];
```



P2:

```
for (i = 0; i < n; i+=2) // n even
{
  s1 = a[i];
  s2 = a[i+1];
  t1 = b[i];
  t2 = b[i+1];
  s1 = s1 + t1;
  s2 = s2 + t2;
  a[i] = s1;
  a[i+1] = s2;
}
```



*No! Possible aliasing*

77

77

## Aliasing

```
for (i = 0; i < n; i++)
  a[i] = a[i] + b[i];
```

Cannot be vectorized in a straightforward way due to potential aliasing.

However, in this case compiler could insert runtime check:

```
if (a + n < b || b + n < a)
  /* vectorized loop */
  ...
else
  /* serial loop */
  ...
```

78

78

## Removing Aliasing

Globally with compiler flag:

- `-fno-alias, /Oa`
- `-fargument-noalias, /QaLias-args-` (function arguments only)

For one loop: pragma

```
void add(double *a, double *b, int n) {  
    #pragma ivdep  
    for (i = 0; i < n; i++)  
        a[i] = a[i] + b[i];  
}
```

For specific arrays: restrict (needs compiler flag `-restrict, /Qrestrict`)

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

79

79

## Proper Code Style

Use countable loops = number of iterations known at runtime

- *Number of iterations is a:*
  - constant*
  - loop invariant term*
  - linear function of outermost loop indices*

Countable or not?

```
for (i = 0; i < n; i++)  
    a[i] = a[i] + b[i];
```

```
void vsum(double *a, double *b, double *c) {  
    int i = 0;  
  
    while (a[i] > 0.0) {  
        a[i] = b[i] * c[i];  
        i++;  
    }  
}
```

80

80



# Proper Code Style

Use arrays, structs of arrays, not arrays of structs

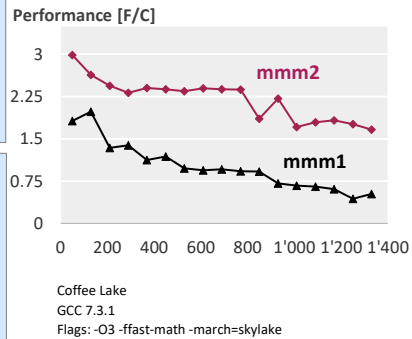
Ideally: unit stride access in innermost loop

```
void mmm1(double *a, double *b, double *c) {
    int N = 100;
    int i, j, k;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```

```
void mmm2(double *a, double *b, double *c) {
    int N = 100;
    int i, j, k;

    for (i = 0; i < N; i++)
        for (k = 0; k < N; k++)
            for (j = 0; j < N; j++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```



81

81

# Alignment

```
double *x = (double *) malloc(1024*sizeof(double));
int i;

for (i = 0; i < 1024; i++)
    x[i] = 1;
```

Without alignment information would require unaligned loads if vectorized.  
However, the compiler can peel the loop to start it at an aligned address:  
the generated assembly would mimic the below C code:

```
double *x = (double *) malloc(1024*sizeof(double));
int i;

peel = (unsigned long) x & 0x1f; /* x mod 32 */
if (peel != 0) {
    peel = (32 - peel)/sizeof(double);
    /* initial segment */
    for (i = 0; i < peel; i++)
        x[i] = 1;
}
/* 32-byte aligned access */
for (i = peel; i < 1024; i++)
    x[i] = 1;
```

82

82

## Ensuring Alignment

Align arrays to 32-byte boundaries (see earlier discussion)

If compiler cannot analyze:

- *Use pragma for loops*

```
double *x = (double *) malloc(1024*sizeof(double));
int i;

#pragma vector aligned
for (i = 0; i < 1024; i++)
    x[i] = 1;
```

- *For specific arrays:*  
`__assume_aligned(a, 32);`

83

83

## More Tips (icc 19.1)

<https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-programming-guidelines-for-vectorization>

Use simple for loops. Avoid complex loop termination conditions – the upper iteration limit must be invariant within the loop. For the innermost loop in a nest of loops, you could set the upper limit iteration to be a function of the outer loop indices.

Write straight-line code. Avoid branches such as switch, goto, or return statements, most function calls, or if constructs that can not be treated as masked assignments.

Avoid dependencies between loop iterations or at the least, avoid read-after-write dependencies.

Try to use array notations instead of the use of pointers. C programs in particular impose very few restrictions on the use of pointers; aliased pointers may lead to unexpected dependencies. Without help, the compiler often cannot tell whether it is safe to vectorize code containing pointers.

Wherever possible, use the loop index directly in array subscripts instead of incrementing a separate counter for use as an array address.

Access memory efficiently:

- ***Favor inner loops with unit stride.***
- ***Minimize indirect addressing.***
- ***Align your data to 32 byte boundaries (for AVX instructions).***

Choose a suitable data layout with care. Most multimedia extension instruction sets are rather sensitive to alignment.

*Read the above website*

84

84

# Compiler Vectorization

Understand the limitations

Carefully read the manual

85

85