Flynn’s Taxonomy

<table>
<thead>
<tr>
<th></th>
<th>Single instruction</th>
<th>Multiple instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single data</td>
<td><strong>SISD</strong></td>
<td><strong>MISD</strong></td>
</tr>
<tr>
<td></td>
<td>Uniprocessor</td>
<td></td>
</tr>
<tr>
<td>Multiple data</td>
<td><strong>SIMD</strong></td>
<td><strong>MIMD</strong></td>
</tr>
<tr>
<td></td>
<td>Vector computer</td>
<td>Multiprocessors</td>
</tr>
<tr>
<td></td>
<td>Short vector extensions</td>
<td>VLIW</td>
</tr>
<tr>
<td></td>
<td>GPU</td>
<td></td>
</tr>
</tbody>
</table>

*Flynn, Michael J. "Very high-speed computing systems".*
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

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
### 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
Haswell/Skylake/ ... 

Have AVX2
16 AVX registers

256 bit = 4 doubles = 8 singles

<table>
<thead>
<tr>
<th>%ymm0</th>
<th>%ymm1</th>
<th>%ymm2</th>
<th>%ymm3</th>
<th>%ymm4</th>
<th>%ymm5</th>
<th>%ymm6</th>
<th>%ymm7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%ymm8</th>
<th>%ymm9</th>
<th>%ymm10</th>
<th>%ymm11</th>
<th>%ymm12</th>
<th>%ymm13</th>
<th>%ymm14</th>
<th>%ymm15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

AVX Instructions: Examples

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

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

(two-operand!)

(two-operand!)

Instruction Names (Adds in Assembly)

<table>
<thead>
<tr>
<th>AVX</th>
<th>SSE</th>
<th>Scalar (in SSE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>packed (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddps</td>
<td>addps</td>
<td>addss</td>
</tr>
<tr>
<td>single precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vaddpd</td>
<td>addpd</td>
<td>addsd</td>
</tr>
<tr>
<td>double precision</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compiler will use this for floating point.

X86-64 FP Code Example

Inner product of two vectors
- Double precision arithmetic
- Compiled: not vectorized, uses (single-slot) SSE instructions

```c
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
 jl .l10             # if < goto loop
 movapd %xmm1, %xmm0 # return result
 ret
```
**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)

**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
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

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

<table>
<thead>
<tr>
<th>ISA</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMX</td>
<td>124</td>
</tr>
<tr>
<td>SSE</td>
<td>154</td>
</tr>
<tr>
<td>SSE2</td>
<td>236</td>
</tr>
<tr>
<td>SSE3</td>
<td>11</td>
</tr>
<tr>
<td>SSSE3</td>
<td>32</td>
</tr>
<tr>
<td>SSE41</td>
<td>61</td>
</tr>
<tr>
<td>SSE42</td>
<td>19</td>
</tr>
<tr>
<td>AVX</td>
<td>188</td>
</tr>
<tr>
<td>AVX2</td>
<td>191</td>
</tr>
<tr>
<td>AVX-512</td>
<td>3857</td>
</tr>
<tr>
<td>FMA</td>
<td>32</td>
</tr>
<tr>
<td>KNC</td>
<td>601</td>
</tr>
<tr>
<td>SVML</td>
<td>406</td>
</tr>
</tbody>
</table>

2019
Visual Conventions We Will Use

Memory

Increasing address

Registers

- **Commonly:**
  - LSB (least significant bit)

- **We will use:**
  - LSB

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`
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);
```

```
  1.0 2.0 3.0 4.0
```

- Same result as

```
__m256d t = __mm256_set_pd(4.0, 3.0, 2.0, 1.0)
```

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()
```
Intel Intrinsics 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 $1/\text{throughput}$.
Example: Intel throughput 0.33 means throughput is 3 ops/cycle.

*We show throughput in ops/cycle.*

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
## SSE vs. AVX vs. AVX-512

<table>
<thead>
<tr>
<th></th>
<th>SSE</th>
<th>AVX</th>
<th>AVX-512</th>
</tr>
</thead>
<tbody>
<tr>
<td>float, double</td>
<td>4-way, 2-way</td>
<td>8-way, 4-way</td>
<td>16-way, 8-way</td>
</tr>
<tr>
<td>register</td>
<td>16 x 128 bits: %xmm0 - %xmm15</td>
<td>16 x 256 bits: %ymm0 - %ymm15</td>
<td>32 x 512 bits: %zmm0 - %zmm31</td>
</tr>
<tr>
<td>assembly ops</td>
<td>addps, mulpd, ...</td>
<td>vaddps, vmulpd</td>
<td>vaddps, vmulpd</td>
</tr>
<tr>
<td>intrinsics data type</td>
<td>__m128, __m128d</td>
<td>__m256, __m256d</td>
<td>__m512, __m512d</td>
</tr>
<tr>
<td>Intrinsics instructions</td>
<td>_mm_load_ps, _mm_add_pd, ...</td>
<td>_mm256_load_ps, _mm256_add_pd</td>
<td>_mm512_load_ps, _mm512_add_pd</td>
</tr>
</tbody>
</table>

*Mixing SSE and AVX may incur penalties*

## 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
AVX Intrinsics

Load and store

Constants

Arithmetic

Comparison

Conversion

Shuffles

Loads and Stores

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding AVX Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm256_load_pd</td>
<td>Load four double values, address aligned</td>
<td>VMOVAPD ymm, mem</td>
</tr>
<tr>
<td>_mm256_loadu_pd</td>
<td>Load four double values, address unaligned</td>
<td>VMOVUPD ymm, mem</td>
</tr>
<tr>
<td>_mm256_maskload_pd</td>
<td>Load four double values using mask</td>
<td>VMASKMOVAPD ymm, mem</td>
</tr>
<tr>
<td>_mm256Broadcast_sd</td>
<td>Load one double value into all four words</td>
<td>VINSERTF128</td>
</tr>
<tr>
<td>_mm256_i64gather_pd</td>
<td>Load double values from memory using indices.</td>
<td>VGATHERPD ymm, mem, ymm</td>
</tr>
</tbody>
</table>

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

Tables show only most important instructions in category
Loads and Stores

\[ a = \_mm256\_load\_pd(p); \quad // \text{p 32-byte aligned} \]
\[ a = \_mm256\_loadu\_pd(p); \quad // \text{p not aligned} \]

May incur a significant performance penalty

load\_pd on unaligned pointer: seg fault

Side Note: Load and Stores on Skylake

Skylake (load):
Lat = 7
Tp = 2

Skylake (store):
Lat = 5
Tp = 1

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

```
a = __mm256_broadcast_pd(p1); // p1 any alignment
b = __mm256_broadcast_sd(p2); // p2 any alignment
```

Loads and Stores

```
a = __mm256_maskload_pd(p, mask); // p any alignment
```

Skylake: 
Lat = -
Tp = -
**Loads and Stores**

$p$  $p+1$  $p+4$  $p+7$

1.0  2.0  3.0  4.0  $\text{memory}$

$\text{LSB}$  1.0  2.0  3.0  4.0  $a$  $\text{LSB}$  0  1  4  7  $\text{offset}$

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

`scale = \{1,2,4,8\}`

above: `scale = 8` = size of double

---

**Stores Analogous to Loads**

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding AVX instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm256_store_pd</td>
<td>Store four values, address aligned</td>
<td>VMOVAPD</td>
</tr>
<tr>
<td>_mm256_storeu_pd</td>
<td>Store four values, address unaligned</td>
<td>VMOVUPD</td>
</tr>
<tr>
<td>_mm256_maskstore_pd</td>
<td>Store four values using mask</td>
<td>VMASKMOVPD</td>
</tr>
<tr>
<td>_mm256_stream_pd</td>
<td>Store lower and higher 128-bit parts into different memory locations</td>
<td>Composite</td>
</tr>
<tr>
<td>_mm256_streamx2_m128d</td>
<td>Store values without caching, address aligned</td>
<td>VMOVNTPD</td>
</tr>
</tbody>
</table>

Tables show only most important instructions in category
## Constants

<table>
<thead>
<tr>
<th>LSB</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 2.0 3.0 4.0</td>
<td>a = _mm256_set_pd(4.0, 3.0, 2.0, 1.0);</td>
<td>b = _mm256_set1_pd(1.0);</td>
<td>c = _mm256_setzero_pd();</td>
</tr>
</tbody>
</table>

### Skylake: Lat = 1, Tp = 3

## Arithmetic

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding AVX Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm256_add_pd</td>
<td>Addition</td>
<td>VADDPD</td>
</tr>
<tr>
<td>_mm256_sub_pd</td>
<td>Subtraction</td>
<td>VSUBPD</td>
</tr>
<tr>
<td>_mm256_addsub_pd</td>
<td>Alternatively add and subtract</td>
<td>VADDSUBPD</td>
</tr>
<tr>
<td>_mm256_hadd_pd</td>
<td>Half addition</td>
<td>VHADDDP</td>
</tr>
<tr>
<td>_mm256_hsub_pd</td>
<td>Half subtraction</td>
<td>VHSUBPD</td>
</tr>
<tr>
<td>_mm256_mul_pd</td>
<td>Multiplication</td>
<td>VMULPD</td>
</tr>
<tr>
<td>_mm256_div_pd</td>
<td>Division</td>
<td>VDIVPD</td>
</tr>
<tr>
<td>_mm256_sqrt_pd</td>
<td>Squared Root</td>
<td>VSQRTPD</td>
</tr>
<tr>
<td>_mm256_max_pd</td>
<td>Computes Maximum</td>
<td>VMAXPD</td>
</tr>
<tr>
<td>_mm256_min_pd</td>
<td>Computes Minimum</td>
<td>VMINPD</td>
</tr>
<tr>
<td>_mm256 ceil_pd</td>
<td>Computes Ceiling</td>
<td>VROUNDPD</td>
</tr>
<tr>
<td>_mm256 floor_pd</td>
<td>Computes Floor</td>
<td>VROUNDPD</td>
</tr>
<tr>
<td>_mm256_round_pd</td>
<td>Round</td>
<td>VROUNDPD</td>
</tr>
<tr>
<td>_mm256_dp_ps</td>
<td>Single precision dot product</td>
<td>VDPPS</td>
</tr>
<tr>
<td>_mm256_fmadd_pd</td>
<td>Fused multiply-add</td>
<td>VFMADD132pd</td>
</tr>
<tr>
<td>_mm256_fmsub_pd</td>
<td>Fused multiply-subtract</td>
<td>VFMSUB132pd</td>
</tr>
<tr>
<td>_mm256_fmaddsub_pd</td>
<td>Alternatively fmadd, fmsub</td>
<td>VFMAADDSUB132pd</td>
</tr>
</tbody>
</table>

Tables show only most important instructions in category
## Arithmetic

```
LSB 1.0 2.0 3.0 4.0 a
LSB 0.5 1.5 2.5 3.5 b
```

```
c = _mm256_add_pd(a, b);
analogous:
c = _mm256_sub_pd(a, b);
c = _mm256_mul_pd(a, b);
```

**Skylake:**
Lat = 4  
Tp = 2

## Example

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

Vectorization by drawing:

```
x
```

```
load_pd
```

```
set_pd
```

```
add_pd
```

```
store_pd
```

```
x
```
Example

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

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

Is this the best solution?

*No! _mm256_set_pd may be too expensive*

Example

```c
#include <immintrin.h>

// n a multiple of 4, x is 32-byte aligned
void addindex_vec2(double *x, int n) {
  __m256d x_vec, init, incr, ind;
  ind = _mm256_set_pd(3, 2, 1, 0);
  incr = _mm256_set1_pd(4);
  for (int i = 0; i < n; i++) {
    x_vec = _mm256_load_pd(x+i);
    // load 4 doubles
    x_vec = _mm256_add_pd(x_vec, ind);
    // add the two
    ind = _mm256_add_pd(ind, incr);
    // update ind
    _mm256_store_pd(x+i, x_vec);
    // store back
  }
}
```

Code style helps with performance! *Why?*
Arithmetic

Skylake:
Lat = 4
Tp = 2

c = _mm256_max_pd(a, b);

Arithmetic

Skylake:
Lat = 4
Tp = 2

c = _mm256_addsub_pd(a, b);
### Arithmetic

```plaintext
LSB 1.0 2.0 3.0 4.0 a

∪ ∪ ∪

LSB 0.5 1.5 2.5 3.5 b

LSB 3.0 2.0 7.0 6.0 c
```

\[
c = \_mm256\_hadd\_pd(a, b);
\]

**analogous:**

\[
c = \_mm256\_hsub\_pd(a, b);
\]

*Does not cross between 128-bit lanes*

---

### Example

```plaintext
// 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;
  }
}
```

```plaintext
re
  load_pd
  hadd_pd
  1/2
  mul_pd
  store_pd

im
  load_pd
  set1_pd
  1/2
```

---

**Skylake:**

Lat = 7

Tp = 0.5
Example

```c
#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
    }
}
```

// 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;
    }
}

Arithmetic (FMA)

Scalar FMA (128 bit):
```c
d = _mm256_fmadd_sd(a, b, c);
```

Analogous:
```c
d = _mm256_fmsub_sd(a, b, c);
```

LSB  
```
0.5 1.5 2.5 3.5  
```

vector size
```
,10 ,100 ,1,000 ,10,000 ,100,000 ,1,000,000
```

Performance [Flops/cycle]
```
0 1 2 3
```

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

Skylake: Lat = 4
Tp = 2
Example

```c
// y = a + x^2 on complex numbers, a is constant
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);
    }
}
```

```c
#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);
        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);
    }
}
```

Arithmetic

```
__m256 _mm256_dp_ps(__m256 a, __m256 b, const int mask)
```

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

```
LSB 1.0 2.0 3.0 4.0 a (low half) 0.5 1.5 2.5 3.5 b (low half)

0.5 3.0 7.5 14.0

01110101
```

**Same is done for the upper half**
Comparisons

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Macro for operation</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm256_cmp_pd (VCMPPD)</td>
<td>_CMP_EQ_OQ</td>
<td>Equal</td>
</tr>
<tr>
<td></td>
<td>_CMP_EQ_UQ</td>
<td>Equal (unordered)</td>
</tr>
<tr>
<td></td>
<td>_CMP_GE_OQ</td>
<td>Greater Than or Equal</td>
</tr>
<tr>
<td></td>
<td>_CMP_GT_OQ</td>
<td>Greater Than</td>
</tr>
<tr>
<td></td>
<td>_CMP_LE_OQ</td>
<td>Less Than or Equal</td>
</tr>
<tr>
<td></td>
<td>_CMP_LT_OQ</td>
<td>Less Than</td>
</tr>
<tr>
<td></td>
<td>_CMP_NEQ_OQ</td>
<td>Not Equal</td>
</tr>
<tr>
<td></td>
<td>_CMP_NEQ_UQ</td>
<td>Not Equal (unordered)</td>
</tr>
<tr>
<td></td>
<td>_CMP_NGE_UQ</td>
<td>Not Greater Than or Equal (unordered)</td>
</tr>
<tr>
<td></td>
<td>_CMP_NGT_UQ</td>
<td>Not Greater Than (unordered)</td>
</tr>
<tr>
<td></td>
<td>_CMP_NLE_UQ</td>
<td>Not Less Than or Equal (unordered)</td>
</tr>
<tr>
<td></td>
<td>_CMP_NLT_UQ</td>
<td>Not Less Than (unordered)</td>
</tr>
<tr>
<td></td>
<td>_CMP_TRUE_UQ</td>
<td>True (unordered)</td>
</tr>
<tr>
<td></td>
<td>_CMP_FALSE_OQ</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>_CMP_ORD_OQ</td>
<td>Ordered</td>
</tr>
<tr>
<td></td>
<td>_CMP_UNORD_Q</td>
<td>Unordered</td>
</tr>
</tbody>
</table>

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

Comparisons

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

analogous:

```c
_c = _mm256_cmp_pd(a, b, _CMP_EQ_UQ);
```

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

Return type: __m256d

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

Skylake:
Lat = 4
Tp = 2
Example

```c
#include <xmmintrin.h>
void fcond_vec1(double *x, size_t n) {
    int i;
    __m256d vt, vmask, vp, vn, 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);
        vn = _mm256_andnot_pd(vmask, mones);
        vr = _mm256_add_pd(vt, _mm256_or_pd(vp, vn));
        _mm256_store_pd(x+i, vr);
    }
}
```

Example

```c
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.;
    }
}
```
Vectorization

\[ \text{Picture: www.druckundbestell.de} \]

Conversion

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding AVX Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm256_cvtepi32_pd</td>
<td>Convert from 32-bit integer</td>
<td>_VCVTDQ2PD</td>
</tr>
<tr>
<td>_mm256_cvtepi32_ps</td>
<td>Convert from 32-bit integer</td>
<td>_VCVTDQ2PS</td>
</tr>
<tr>
<td>_mm256_cvtpd_epi32</td>
<td>Convert to 32-bit integer</td>
<td>_VCVTP2DQ</td>
</tr>
<tr>
<td>_mm256_cvtps_epi32</td>
<td>Convert to 32-bit integer</td>
<td>_VCVTPS2DQ</td>
</tr>
<tr>
<td>_mm256_cvteps_pd</td>
<td>Convert from floats</td>
<td>_VCVTPS2PD</td>
</tr>
<tr>
<td>_mm256_cvtepd_ps</td>
<td>Convert to floats</td>
<td>_VCVTP2PS</td>
</tr>
<tr>
<td>_mm256_cvtsd_f64</td>
<td>Convert to 32-bit integer with truncation</td>
<td>_VCVTP2DQ</td>
</tr>
<tr>
<td>_mm256_cvtsd_f64</td>
<td>Extract</td>
<td>MOVSD</td>
</tr>
<tr>
<td>_mm256_cvtsd_f32</td>
<td>Extract</td>
<td>MOVSS</td>
</tr>
</tbody>
</table>

Tables show only most important instructions in category
Conversion

```c
double _mm256_cvtsd_f64(__m256d a)
```

LSB 1.0 2.0 3.0 4.0 a

```c
d double d;
d = _mm_cvtsd_f64(a);
```

Conversion

```c
__m256d _mm256_cvtepi32_pd(__m128i a)
```

```c
__m256d _mm256_cvtepi64_pd(__m256i a)
```

```c
__m256d _mm256_cvtepu32_pd(__m256i a)
```

```c
__m256d _mm256_cvtepu64_pd(__m256i a)
```

Skylake:
Lat = -
Tp = -

See also:
```c
__m256d _mm256_cvtepi64_pd(__m256i a)
```
```c
__m256d _mm256_cvtepu32_pd(__m256i a)
```
```c
__m256d _mm256_cvtepu64_pd(__m256i a)
```
Shuffles

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

Tables show only most important instructions in category

Skylake:
Lat = 1
Tp = 1

Does not cross between 128-bit lanes

→ blackboard
Shuffles

__m256d __m256d_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:  

\[
\begin{array}{cccc}
\text{LSB} & 0x0 & 0x0 & 0x0 & \text{mask} \\
\text{LSB} & 1.0 & 2.0 & 3.0 & 4.0 & a \\
\text{LSB} & 0.5 & 1.5 & 2.5 & 3.5 & b \\
\text{LSB} & 1.0 & 1.5 & 3.0 & 4.0 & c \\
\end{array}
\]

see also __mm256_blend_pd:
same with integer mask, \( T_p = 3/2 \)

Example (Continued From Before)

```c
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.;
    }
}
```

```c
#include <immintrin.h>

void fcond_vec2(double *x, size_t n) {
    int i;
    __m256d vt, vmask, vp, vn, 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:  
Lat = 2  
\( T_p = 3/2 \)
Example: Loading 4 Doubles from Arbitrary Memory Locations

Assumes all values are within one array

```
#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]
vunpcklqd ymm2, ymm0, [rsi]
vunpcklqd ymm3, ymm1, [-16+rcx]
vblendpd ymm0, ymm2, ymm3, 12
```

7 instructions, this is one way of doing it
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()`

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 `movpd` can

(`_mm_loadh_pd`)
Example: Loading 4 Doubles from Arbitrary Memory Locations (cont’d)

Example compilation:

\[
\begin{align*}
&\text{vmovsd } xmm0 \ [rdi] \\
&\text{vmovsd } xmm1, \ [rdx] \\
&\text{vmovhpd } xmm2, xmm0, \ [rsi] \ // \text{ SSE register xmm2 written} \\
&\text{vmovhpd } xmm3, xmm1, \ [rcx] \\
&\text{vinsertf128 } ymm0, ymm2, xmm3, 1 \ // \text{ accessed as ymm2}
\end{align*}
\]

Written in intrinsics (reverse-engineered):

```c
#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);
}
```

Example: Loading 4 Doubles from Arbitrary Memory Locations

Picture for previous slide (this solution always works):
Example: Loading 4 Doubles from Arbitrary Memory Locations (cont’d)

Do not do this (why?):

```c
__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);
```

Shuffles

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

<table>
<thead>
<tr>
<th>LSB</th>
<th>a</th>
<th>LSB</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2.0</td>
<td>3.0</td>
<td>4.0</td>
</tr>
<tr>
<td>0.5</td>
<td>1.5</td>
<td>2.5</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Does not cross between 128-bit lanes
Shuffles

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

Shuffle elements within 128-bits lanes.

Example:

![Diagram showing shuffle process]

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:

![Diagram showing shuffle process]

Somewhat more expensive due to shuffle between 128-bits 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:

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

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):

```c
__m256d __mm256_add_pd (__m256d a, __m256d b) {
    __m128d r;
    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);
}
```

```c
__m128d __mm_add_pd (__m128d a, __m128d b) {
    __m128d r;
    r.neon_f64 = vaddq_f64(a.neon_f64, b.neon_f64); // NEON intrinsic
    return __m256d_from_private(r);
}
```
SIMDe Library vs. Neon Intrinsics

```c
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.;
    }
}
```

```c
#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);
    }
}
```

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 ([intrinsicsguide](#)) and their performance
SIMD Extensions and AVX

AVX intrinsics

*Compiler vectorization*

References:
*Intel icc manual* (look for auto vectorization)

Compiler Vectorization

Compiler flags

Aliasing

Proper code style

Alignment
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

Example

unvectorized: /Qvec-

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

```assembly
    vmovsd xmm0, DWORD PTR [rcx+rax*4]
    vaddsd xmm0, DWORD PTR [rdx+rax*4]
    vmovsd DWORD PTR [rcx+rax*4], xmm0
```

vectorized:

```assembly
    vmovsd xmm0, DWORD PTR [rcx+r11*4]
    vaddsd xmm0, DWORD PTR [rdx+r11*4]
    vmovsd DWORD PTR [rcx+r11*4], xmm0
```

```assembly
    vmovupd ymm0, YMMWORD PTR [rdx+r10*4]
    vaddpd ymm0, ymm0, YMMWORD PTR [rcx+r10*4]
    vaddpd ymm0, ymm0, YMMWORD PTR [rcx+r10*4]
    vmovupd YMMWORD PTR [rcx+r10*4], ymm0
    vmovupd YMMWORD PTR [16+rcx+r10*4], ymm1
```

why this?

why everything twice?
Are These Programs Equivalent?

**P1:**
```plaintext
for (i = 0; i < n; i++) // n even
    a[i] = a[i] + b[i];
```

**P2:**
```plaintext
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*

Aliasing

```plaintext
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:

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

Globally with compiler flag:
- `-fno-alias`, `/Oa`
- `-fargument-noalias`, `/Qalias-args` (function arguments only)

For one loop: pragma

```c
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`)

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

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?

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

```c
void vsum(double *a, double *b, double *c) {
    int i = 0;

    while (a[i] > 0.0) {
        a[i] = b[i] * c[i];
        i++;
    }
}
```
Proper Code Style

Use arrays, structs of arrays, not arrays of structs

Ideally: unit stride access in innermost loop

```c
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];
}
```

```c
void mmm2(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];
}
```

Alignment

```c
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:

```c
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;
```
Ensuring Alignment

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

If compiler cannot analyze:

- **Use pragma for loops**
  ```c
  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);
  ```


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*
Compiler Vectorization

Understand the limitations

Carefully read the manual