Flynn’s Taxonomy

<table>
<thead>
<tr>
<th>Single data</th>
<th>Single instruction</th>
<th>Multiple instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>SISD</strong></td>
<td><strong>MISD</strong></td>
</tr>
<tr>
<td></td>
<td>Uniprocessor</td>
<td></td>
</tr>
</tbody>
</table>

| Multiple data     | **SIMD**           | **MIMD**             |
|                   | Vector computer    | Multiprocessors      |
|                   | Short vector extensions | VLIW               |
SIMD Extensions and AVX

- AVX intrinsics
- Compiler vectorization

The first version of this lecture 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
Example AVX Family: Floating Point

- Not drawn to scale
- AVX: introduces three-operand instructions ($c = a + b$ vs. $a = a + b$)
- AVX2: Introduces fused multiply-add (FMA: $c = c + a*b$)
- Sandy Bridge and later has AVX
Haswell/Skylake/ ...

- Have AVX2
- 16 AVX registers

256 bit = 4 doubles = 8 singles

%ymm0  %ymm1  %ymm2  %ymm3  %ymm4  %ymm5  %ymm6  %ymm7  %ymm8  %ymm9  %ymm10  %ymm11  %ymm12  %ymm13  %ymm14  %ymm15

32 zmm (AVX-512)  16 ymm (AVX)  16 xmm (SSE)  scalar
AVX Registers

- Used for different data types and instructions
  - 256 bit
  - 256 bit LSB

- 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 add: vaddpd %ymm1 %ymm0 %ymm1

- Double precision scalar add (in SSE2): addsd %xmm0 %xmm1 (three-operand!) (two-operand!)
Instruction Names (Assembly)

AVX

packed (vector)

vaddps  addps

single precision

vaddpd  addpd

double precision

SSE

Scalar (in SSE)

single slot

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
    xor1   %ecx, %ecx
    jmp    .L8
.L10:
    movslq %ecx,%rax
    incl   %ecx
    movsd  (%rsi,%rax,4), %xmm0
    mulsd  (%rdi,%rax,4), %xmm0
    addsd  %xmm0, %xmm1
.L8:
    cmpl   %edx, %ecx
    jle    .L9
    movapd %xmm1, %xmm0
    .L9:
    ret
```

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)

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

- Not drawn to scale
- AVX: introduces three-operand instructions (c = a + b vs. a = a + b)
- AVX2: Introduces fused multiply-add (FMA)
- Sandy Bridge and later has AVX

Intrinsics

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

<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
  
  ![Memory Diagram]

- **Registers**
  - Commonly:
    - ![Registers Diagram]
  - We will use
    - ![Registers Diagram]

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`

![Data Types Diagram]
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);
    ```

    ![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)
    ```

- **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()`
    - ...
  - Note: not every assembly instruction has a corresponding intrinsic
Intel Intrinsics Guide

- **Intel Intrinsics Guide**
- Great resource to quickly find the right intrinsics
- Has latency and throughput information for many instructions

*Note:* Intel measures throughput in cycles, i.e., really shows $1$/throughput. Example: Intel throughput 0.33 means throughput is 3 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 (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_ps, ...</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*

### 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>_mm256_broadcast_sd</td>
<td>Load one double value into all four words</td>
<td>VBROADCASTSD ymm, mem</td>
</tr>
<tr>
<td>_mm256_broadcast_pd</td>
<td>Load a pair of double values into the lower</td>
<td>VINSERTF128 ymm, ymm</td>
</tr>
<tr>
<td></td>
<td>and higher part of vector.</td>
<td></td>
</tr>
<tr>
<td>_mm256_i64gather_pd</td>
<td>Load double values from memory using indices.</td>
<td>VINSERTF128 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 ymm, ymm</td>
</tr>
</tbody>
</table>

Tables show only most important instructions in category

### Skylake: Lat = 1, Tp = 4

```
load_pd on unaligned pointer: seg fault
```

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

Used to be more expensive
Not anymore
```

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

a = _mm256_broadcast_pd(p1); // p any alignment

b = _mm256_broadcast_sd(p2); // p any alignment

---

Loads and Stores

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

__m256i
Loads and Stores

\[ a = \_mm256\_i64gather\_pd(p, \text{offset}, 8); \] // 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>VMASKMOVAPD</td>
</tr>
<tr>
<td>_mm256_storeu2_m128d</td>
<td>Store lower and higher 128-bit parts into different memory locations</td>
<td>Composite</td>
</tr>
<tr>
<td>_mm256_stream_pd</td>
<td>Store values without caching, address aligned</td>
<td>VMOVNTPD</td>
</tr>
</tbody>
</table>
**Constants**

<table>
<thead>
<tr>
<th>LSB</th>
<th>Value</th>
<th>Intrinsics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1.0 2.0 3.0 4.0</strong></td>
<td>a</td>
<td>(a = _\text{mm256_set_pd}(4.0, 3.0, 2.0, 1.0);)</td>
</tr>
<tr>
<td><strong>1.0 1.0 1.0 1.0</strong></td>
<td>b</td>
<td>(b = _\text{mm256_set1_pd}(1.0);)</td>
</tr>
<tr>
<td><strong>0 0 0 0</strong></td>
<td>c</td>
<td>(c = _\text{mm256_setzero_pd}();)</td>
</tr>
</tbody>
</table>

**Skylake:**

- \(\text{Lat} = 1\)
- \(\text{Tp} = 3\)

---

**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>VHADDPD</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>VFMAADD132pd</td>
</tr>
<tr>
<td>_mm256_fmsub_pd</td>
<td>Fused multiply-subtract</td>
<td>VFMSUB132pd</td>
</tr>
<tr>
<td>_mm256_fmadsub_pd</td>
<td>Alternatively fmadd, fmsub</td>
<td>VFMAADDSUB132pd</td>
</tr>
</tbody>
</table>

Tables show only most important instructions in category
Arithmetic

\[
\begin{array}{cc}
\text{LSB} & 1.0 & 2.0 & 3.0 & 4.0 & a \\
\text{LSB} & 0.5 & 1.5 & 2.5 & 3.5 & b \\
\text{LSB} & 1.5 & 3.5 & 5.5 & 7.5 & c \\
\end{array}
\]

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

analogous:

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

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

Example

```c
#include <immintrin.h>

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

void addindex_vec(double *x, int n) {
    __m256d index, x_vec;
    for (int i = 0; i < n; i+=4) {
        x_vec = __mm256_load_pd(x+i); // load 4 doubles
        index = __mm256_set_pd(i+3, i+2, i+1, i); // create vector with indexes
        x_vec = __mm256_add_pd(x_vec, index); // add the two
        __mm256_store_pd(x+i, x_vec); // store back
    }
}
```

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_vec(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 += 4) {
        x_vec = _mm256_load_pd(x+i);
        x_vec = _mm256_add_pd(x_vec, ind);
        ind = _mm256_add_pd(ind, incr);
        _mm256_store_pd(x+i, x_vec);
    }
}
```

Code style helps with performance! Why?

Arithmetic

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

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

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

\[
c = \_mm256\_hsub\_pd(a, b);
\]
### 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+=4) {
        z[i] = (re[i] + re[i+1])/2;
        z[i+1] = (im[i] + im[i+1])/2;
    }
}
```

---

### Arithmetic (FMA)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</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>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>0.5</td>
<td>1.5</td>
<td>2.5</td>
<td>3.5</td>
</tr>
<tr>
<td>1.0</td>
<td>2.5</td>
<td>4.0</td>
<td>5.5</td>
</tr>
</tbody>
</table>

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

analogous:
```
d = _mm256_fmsub_pd(a, b, c);
```
Example

```c
// 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];
    }
}
```

```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

```c
__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

Skylake:
Lat = 13
Tp = 2/3
Comparisons

<table>
<thead>
<tr>
<th>Intrinsic Name (VCMPPD)</th>
<th>Macro for operation</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm256_cmp_pd</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_OQ</td>
<td>Unordered</td>
</tr>
</tbody>
</table>

Tables show only most important instructions in category

Skylake:
Lat = 4
Tp = 2

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

*analagous:*

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

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

*etc.*
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.;
    }
}
```

```c
#include <xmmintrin.h>
void fcond(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);
    }
}
```

Vectorization

=
## 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>VCVTDO2PD</td>
</tr>
<tr>
<td>_mm256_cvtepi32_ps</td>
<td>Convert from 32-bit integer</td>
<td>VCVTDO2PS</td>
</tr>
<tr>
<td>_mm256_cvtpd_epi32</td>
<td>Convert to 32-bit integer</td>
<td>VCVTPO2DQ</td>
</tr>
<tr>
<td>_mm256_cvtpd_epi32</td>
<td>Convert to 32-bit integer</td>
<td>VCVTPO2DQ</td>
</tr>
<tr>
<td>_mm256_cvtps_pd</td>
<td>Convert from floats</td>
<td>VCVTPS2PD</td>
</tr>
<tr>
<td>_mm256_cvtps_ps</td>
<td>Convert from floats</td>
<td>VCVTPO2PS</td>
</tr>
<tr>
<td>_mm256_cvtpd_epi32</td>
<td>Convert to 32-bit integer with truncation</td>
<td>VCVTPO2DQ</td>
</tr>
<tr>
<td>_mm256_cvtd_f64</td>
<td>Extract</td>
<td>MOVSD</td>
</tr>
<tr>
<td>_mm256_cvtss_f32</td>
<td>Extract</td>
<td>MOVSS</td>
</tr>
</tbody>
</table>

Tables show only most important instructions in category.
Conversion

```cpp
__m256d _mm256_cvtepi32_pd(__m128i a)
```

Convert

ints ➜ doubles

See also:

```cpp
__m256d _mm256_cvtepi64_pd(__m256i a)
__m256d _mm256_cvtepu32_pd(__m256i a)
__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_unpackh_pd</td>
<td>Unpack High</td>
<td>VUNPCKHPD</td>
</tr>
<tr>
<td>__mm256_unpacklo_pd</td>
<td>Unpack Low</td>
<td>VUNPCKLDP</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_blendlv_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>VSHUFFPD</td>
</tr>
<tr>
<td>__mm256_permute_pd</td>
<td>Permute</td>
<td>VPERMULPD</td>
</tr>
<tr>
<td>__mm256_permute4x64_pd</td>
<td>Permute 64-bits elements</td>
<td>VPERMPPD</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
Shuffles

1.0 2.0 3.0 4.0 a
1.0 0.5 3.0 2.5 c

0.5 1.5 2.5 3.5 b

Does not cross between 128-bit lanes

c = _mm256_unpacklo_pd(a, b);

1.0 2.0 3.0 4.0 a
3.0 2.5 4.0 3.5 c

0.5 1.5 2.5 3.5 b

Skylake: Lat = 1
Tp = 1

→ blackboard

Shuffles

__m256d __m256d_blendv_pd(__m256d a, __m256d b, __m256 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 0xff .. f 0x0 0x0 mask

1.0 2.0 3.0 4.0 a
1.0 1.5 3.0 4.0 c

0.5 1.5 2.5 3.5 b

Skylake: Lat = 1
Tp = 1

see also __mm256_blend_pd:
same with integer mask, Tp = 3!
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.;
    }
}
```

#include <immintrin.h>

```c
void fcond(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);
    }
}
```

Example: Loading 4 Real Numbers from Arbitrary Memory Locations

![Diagram](image.png)

4–10x speedup

Example: Loading 4 Real Numbers from Arbitrary Memory Locations

- 4x loadu_pd
- 2x unpacklo_pd
- 1x blend_pd

7 instructions, this is one way of doing it
Code For Previous Slide

```c
#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:

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

no intrinsic for this instruction (Nov 2019)

Example: Loading 4 Real Numbers from Arbitrary Memory Locations (cont’d)

- Whenever possible avoid the previous situation
- Restructure algorithm and use the aligned _mm256_load_pd()
Example: Loading 4 Real Numbers from Arbitrary Memory Locations (cont’d)

- Other possibility

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

Example compilation:

```c
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 compilation:

```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);
}
```

Written in intrinsics (reverse-engineered):
Example: Loading 4 Real Numbers from Arbitrary Memory Locations

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

\texttt{__m256d \_mm\_shuffle\_pd(__m256d \textit{a}, __m256d \textit{b}, int \textit{mask})}

\begin{itemize}
  \item \textit{a} \begin{tabular}{cccc}
  1.0 & 2.0 & 3.0 & 4.0 \\
  \end{tabular}
  \item \textit{b} \begin{tabular}{cccc}
  0.5 & 1.5 & 2.5 & 3.5 \\
  \end{tabular}
  \item \textit{c} \begin{tabular}{cccc}
  \texttt{c0} & \texttt{c1} & \texttt{c2} & \texttt{c3} \\
  \end{tabular}
\end{itemize}

\texttt{a0} or \texttt{a1}

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

\textit{Does not cross between 128-bit lanes}

Shuffles

\texttt{__m256d \_mm256\_permute\_pd(__m256d \textit{a}, int \textit{mask})}

Shuffle elements within 128-bits lanes.

\textit{Example:}

\texttt{a} \begin{tabular}{cccc}
  1 & 2 & 3 & 4 \\
  \end{tabular}
\texttt{mask} \begin{tabular}{cccc}
  1101 \\
  \end{tabular}

\texttt{c} \begin{tabular}{cccc}
  2 & 1 & 4 & 4 \\
  \end{tabular}

\textit{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-bits lanes

---

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!](#))
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];
}
```

```asm
;;;    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
```

vectorized:

```asm
;;;    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
```

why this?

why everything twice?
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 can insert runtime check:

```c
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][j] = c[i][j] + a[i][k] * b[k][j];
}
```

```c
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][j] = c[i][j] + a[i][k] * b[k][j];
}
```
Alignment

double *x = (double *) malloc(1024*sizeof(double));
int i;
for (i = 0; i < 1024; i++)
    x[i] = 1;

Cannot be vectorized in a straightforward way since x may not be aligned

However, the compiler can peel the loop to extract aligned part:

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

peel = x & 0x1f; /* x mod 32 */
if (peel != 0) {
    peel = 32 - peel;
    /* 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
    
    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);
More Tips (icc 19.0)  
[https://software.intel.com/en-us/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.
- ...

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

Assume:
- No aliasing information
- No alignment information

Can compiler vectorize?

In principle yes: through versioning

![Versioning Decision Tree]

However, this causes code size blowup and is not feasible for large code
Compiler Vectorization

- Understand limitations
- Read manual