Advanced Systems Lab
Spring 2023

Lecture: SIMD extensions, AVX, compiler vectorization

Instructor: Markus Püschel, Ce Zhang
TA: Joao Rivera, several more

Flynn’s Taxonomy

<table>
<thead>
<tr>
<th></th>
<th>Single instruction</th>
<th>Multiple instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single data</strong></td>
<td><strong>SISD</strong></td>
<td><strong>MISD</strong></td>
</tr>
<tr>
<td>Uniprocessor</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multiple data</strong></td>
<td><strong>SIMD</strong></td>
<td><strong>MIMD</strong></td>
</tr>
<tr>
<td>Vector computer</td>
<td></td>
<td>Multiprocessors</td>
</tr>
<tr>
<td>Short vector extensions</td>
<td></td>
<td>VLIW</td>
</tr>
</tbody>
</table>
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
Example AVX Family: Floating Point

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

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

Sandy Bridge and later has (at least) 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 (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 add: vaddpd %ymm1 %ymm0 %ymm1)

(Double precision scalar add (in SSE2): addsd %xmm0 %xmm1)
### Instruction Names (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>single slot</td>
</tr>
<tr>
<td>vaddps</td>
<td>addps</td>
<td></td>
</tr>
<tr>
<td></td>
<td>single precision</td>
<td></td>
</tr>
<tr>
<td>addpd</td>
<td>addpd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double precision</td>
<td></td>
</tr>
<tr>
<td>compiled will use this for floating point</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 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;
}
```
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-512
AVX2: FMA
AVX: 4-way double

*our focus*

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 (at least) AVX

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>SSE4</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:
  
  \[ \begin{array}{cccc}
  R3 & R2 & R1 & R0 \\
  \end{array} \]

• We will use
  
  \[ \begin{array}{cccc}
  R0 & R1 & R2 & R3 \\
  \end{array} \]

AVX Intrinsics (Focus Floating Point)

Data types

\[
\begin{array}{l}
\text{\_m256 } f; \quad // = \{\text{float } f0, f1, f2, f3, f4, f5, f6, f7\} \\
\text{\_m256d } d; \quad // = \{\text{double } d0, d1, d3, d4\} \\
\text{\_m256i } i; \quad // 32 \ 8\text{-bit, } 16 \ 16\text{-bit, } 8 \ 32\text{-bit, or } 4 \ 64\text{-bit} \\
\end{array}
\]
AVX Intrinsics (Focus Floating Point)

Instructions

- **Naming convention:** `__mm256_<intrin_op>_<suffix>`
- **Example:**

  ```c
  // 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**

  ```c
  __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

*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.

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: %yymm0 - %ymmm</td>
<td>32 x 512 bits: %zmm0 - %zmm31</td>
</tr>
<tr>
<td>The lower halves are the %xms</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The lower halves are the %ymms</td>
<td></td>
<td></td>
<td></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*

AVX Instrinsics

Load and store

Constants

Arithmetic

Comparison

Conversion

Shuffles
Loads and Stores

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding AVX Instructions</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>VMASKMOVPD 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 and</td>
<td>VBROADCASTSD ymm, mem</td>
</tr>
<tr>
<td></td>
<td>higher part of vector</td>
<td></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>

Tables show only most important instructions in category

Loads and Stores

```
p

1.0 2.0 3.0 4.0

memory

1.0 2.0 3.0 4.0

a

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
```
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

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

**Skylake:**
Lat = -
Tp = -

**Loads**

\[ a = \_mm256\_maskload\_pd(p, mask); // p any alignment \]

\[ \_m256l \]

**Stores**

\[ a = \_mm256\_i64gather\_pd(p, offset, 8); // p any alignment \]

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

---

**Skylake:**
Lat = -
Tp = -

**Loads**

\[ a = \_mm256\_maskload\_pd(p, mask); // p any alignment \]

\[ \_m256l \]

**Stores**

\[ a = \_mm256\_i64gather\_pd(p, offset, 8); // p any alignment \]

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_storeu2_m128d</td>
<td>Store lower and higher 128-bit parts into</td>
<td>Composite</td>
</tr>
<tr>
<td></td>
<td>different memory locations</td>
<td></td>
</tr>
<tr>
<td>_mm256_stream_pd</td>
<td>Store values without caching, address aligned</td>
<td>VMOVNTPD</td>
</tr>
</tbody>
</table>

Tables show only most important instructions in category

### Constants

- **a**
  ```
  a = _mm256_set_pd(4.0, 3.0, 2.0, 1.0);
  ```

- **b**
  ```
  b = _mm256_set1_pd(1.0);
  ```

- **c**
  ```
  c = _mm256_setzero_pd();
  ```
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 Ceil</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_fmaddsub_pd</td>
<td>Alternatively fmadd, fmsub</td>
<td>VFMAADDSUB132pd</td>
</tr>
</tbody>
</table>

Tables show only most important instructions in category

1.0 2.0 3.0 4.0
a

0.5 1.5 2.5 3.5
b

1.5 3.5 5.5 7.5

\[ c = \text{\_mm256\_add\_pd}(a, b) \] = _mm256_add_pd(a, b);

analogous:

\[ c = \text{\_mm256\_sub\_pd}(a, b) \] = _mm256_sub_pd(a, b);

\[ c = \text{\_mm256\_mul\_pd}(a, b) \] = _mm256_mul_pd(a, b);
Example

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

Vectorization by drawing:

![Diagram of vectorization](image)

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+=4) {
        x_vec = _mm256_load_pd(x+i);
        index = _mm256_set_pd(i+3, i+2, i+1, i);
        x_vec = _mm256_add_pd(x_vec, index);
        _mm256_store_pd(x+i, x_vec);
    }
}
```

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

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

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

analogous:

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

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

Example

#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
    }
}
Arithmetic (FMA)

$$d = _\text{mm256}_\text{fmadd}_\text{pd}(a, b, c);$$

**analogous:**

$$d = _\text{mm256}_\text{fmsub}_\text{pd}(a, b, c);$$

**scalar FMA (128 bit):**

$$d = _\text{mm}_\text{fmadd}_\text{sd}(a, b, c);$$

---

**Example**

```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);
    re = _mm_fmadd_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);
  }
}
```

---

Performance [Flops/cycle]

- **Skyland:**
  - Lat = 4
  - Tp = 2

- **Coffee Lake:**
  - clang 9.0.0
  - -O3 -mavx2 -mfma -fno-tree-vectorize
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

![Diagram](image)

**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_DQ</td>
<td>Equal</td>
</tr>
<tr>
<td></td>
<td>CMP_EQ_UQ</td>
<td>Equal (unordered)</td>
</tr>
<tr>
<td></td>
<td>CMP_GE_DQ</td>
<td>Greater Than or Equal</td>
</tr>
<tr>
<td></td>
<td>CMP_GT_DQ</td>
<td>Greater Than</td>
</tr>
<tr>
<td></td>
<td>CMP_LE_DQ</td>
<td>Less Than or Equal</td>
</tr>
<tr>
<td></td>
<td>CMP_LT_DQ</td>
<td>Less Than</td>
</tr>
<tr>
<td></td>
<td>CMP_NEQ_DQ</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_DQ</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>CMP_ORD_Q</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

Each field:
- 0xffffffff if true
- 0x0 if false

Return type: __m256d

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.;
    }
}
```
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_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);
    }
}
```

Vectorization

= 

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>VCVTQD2DQ</td>
</tr>
<tr>
<td>_mm256_cvtpd_epi32</td>
<td>Convert to 32-bit integer</td>
<td>VCVTPS2DQ</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>VCVTQD2PS</td>
</tr>
<tr>
<td>_mm256_cvtpd_epi32</td>
<td>Convert to 32-bit integer with truncation</td>
<td>VCVTQD2DQ</td>
</tr>
<tr>
<td>_mm256_cvtst_f64</td>
<td>Extract</td>
<td>MOVSD</td>
</tr>
<tr>
<td>_mm256_cvtss_f64</td>
<td>Extract</td>
<td>MVSS</td>
</tr>
</tbody>
</table>

Tables show only most important instructions in category

---

double _mm256_cvtst_f64(__m256d a)

LSB

```
1.0 2.0 3.0 4.0
```

```
d = _mm_cvtst_f64(a);
```
Conversion

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

**convert**

ints  ➔  doubles

See also:

```c
__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_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>VBLENDVDPD</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>VPERMIDP</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

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

\[
c = \_\text{mm256\_unpacklo\_pd}(a, b);
\]

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

Shuffles

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

\[
c = \_\text{mm256\_unpackhi\_pd}(a, b);
\]

\[
\_\text{m256d} \_\text{mm256\_blendv\_pd}(\_\text{m256d} \ a, \_\text{m256d} \ b, \_\text{m256d} \ \text{mask})
\]

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

\text{Example:}

\[
\begin{aligned}
\text{LSB } 0x0 & \ 0xffff & \ 0x0 & \ 0x0 & \text{mask}
\end{aligned}
\]

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

see also \_\text{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.;
    }
}
```

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

Example: Loading 4 Doubles from Arbitrary Memory Locations

Assumes all values are within one array

```
    p0       p1       p2       p3
         |         |         |
       1.0      2.0      3.0      4.0
            |             |             |
        LSB        LSB        LSB        LSB
        *           *           *           *

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

Example compilation:

```c
vmovups ymm0, [rdi]
vmovups ymm1, [-16+rdx]
vunpcklps ymm2, ymm0, [rsi]
vunpcklps ymm3, ymm1, [-16+rcx]
vblendpd ymm0, ymm2, ymm3, 12
```

Example compilation: *no intrinsic for this instruction (Nov 2019)*

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

Other possibility

```c
__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:

```
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):

```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(t4); // 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

```c
__m256d __mm256_shuffle_pd(__m256d a, __m256d b, const int mask)
```

<table>
<thead>
<tr>
<th>LSB</th>
<th>1.0</th>
<th>2.0</th>
<th>3.0</th>
<th>4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LSB</th>
<th>0.5</th>
<th>1.5</th>
<th>2.5</th>
<th>3.5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LSB</th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a0</td>
<td>c0</td>
<td>c1</td>
<td>c2</td>
<td>c3</td>
</tr>
<tr>
<td>a1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
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

---

Shuffles

```c
__m256d __mm256_permute_pd(__m256d a, int mask)
```

Shuffle elements within 128-bits lanes.

**Example:**

```
1101 mask
```

```
1 2 3 4 a
```

```
2 1 4 4 c
```

Does not cross between 128-bit lanes
### Shuffles

\[
\_{\text{m256d \_mm256_permute4x64_pd}}(\_{\text{m256d \ a}}, \text{ int \ mask})
\]

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

**Example:**

```
\[
\begin{array}{c}
\text{LSB}\\
1 & 2 & 3 & 4 \\
\end{array}
\]
```

```
\[
\begin{array}{c}
\text{LSB}\\
2 & 1 & 2 & 4 \\
\end{array}
\]
```

```
\[
\begin{array}{c}
a\\
\end{array}
\]

\[
\begin{array}{c}
m\text{ask}\\
11010001 \\
\end{array}
\]

```
```
\[
\begin{array}{c}
c\\
\end{array}
\]

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
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) {
    __m256d r = _mm_add_pd(a, b);
    return __m256d_from_private(r);
}
```

SIMDe Library vs. Neon Intrinsics

```c
#include <arm_neon.h>

void fcond_neon(double* x, size_t n) {
    float64x2_t vt1, vb1, vr1, vmask1;
    float64x2_t vt2, vb2, vr2, vmask2;
    float64x2_t ones = vdupq_n_f64(1.0);
    float64x2_t mones = vdupq_n_f64(-1.0);
    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.
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

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, xxpdp

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];
}
```

vectorized:

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

Are These Programs Equivalent?

P1:

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

P2:

```c
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

```c
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:

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

Removing Aliasing

Globally with compiler flag:
- `-fno-alias`, `/Qa`
- `-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 (k = 0; k < N; k++)
            for (j = 0; j < N; j++)
                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:**
  
  ```c
  __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