How to Write Fast Numerical Code
Spring 2019
Lecture: SIMD extensions, SSE, compiler vectorization

Instructor: Markus Püschel
TA: Tyler Smith, Gagandeep Singh, Alen Stojanov

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> Uniprocessor</td>
<td><strong>MISD</strong></td>
</tr>
<tr>
<td>Multiple data</td>
<td><strong>SIMD</strong> Vector computer Short vector extensions</td>
<td><strong>MIMD</strong> Multiprocessors VLIW</td>
</tr>
</tbody>
</table>
SIMD Extensions and SSE

- SSE intrinsics
- Compiler vectorization

This lecture and material was created together with Franz Franchetti (ECE, Carnegie Mellon)

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: MMX, SSE, SSE2, ...

- 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
MMX: Multimedia extension
SSE: Streaming SIMD extension
AVX: Advanced vector extensions

Example SSE Family: Floating Point

- Not drawn to scale
- From SSE3: Only additional instructions
- Every Core 2 has SSE3

Intel x86

<table>
<thead>
<tr>
<th>Register width</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 bit (only int)</td>
<td>x86-16 8086</td>
</tr>
<tr>
<td>2 doubles = 128 bit</td>
<td>x86-32 286</td>
</tr>
<tr>
<td>4 doubles = 256 bit</td>
<td>x86-64</td>
</tr>
<tr>
<td>8 doubles = 512 bit</td>
<td>SSE MMX</td>
</tr>
<tr>
<td>SSE</td>
<td>SSE2</td>
</tr>
<tr>
<td>SSE3</td>
<td>AVX</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX-512</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Intel x86</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86-16 8086</td>
<td></td>
</tr>
<tr>
<td>x86-32 286</td>
<td></td>
</tr>
<tr>
<td>x86-64</td>
<td>Pentium III</td>
</tr>
<tr>
<td>SSE</td>
<td>Pentium 4</td>
</tr>
<tr>
<td>SSE2</td>
<td>Pentium 4E</td>
</tr>
<tr>
<td>SSE3</td>
<td>Pentium 4F</td>
</tr>
<tr>
<td>AVX</td>
<td>Core 2 Duo</td>
</tr>
<tr>
<td>AVX2</td>
<td>Penryn</td>
</tr>
<tr>
<td>AVX-512</td>
<td>Core i7 (Nehalem)</td>
</tr>
<tr>
<td></td>
<td>Sandy Bridge</td>
</tr>
<tr>
<td></td>
<td>Haswell</td>
</tr>
<tr>
<td></td>
<td>Skylake-X</td>
</tr>
</tbody>
</table>

Intel x86 Processors

- MMX
- SSE
- SSE2
- SSE3
- AVX
- AVX2
- AVX-512

2 doubles = 128 bit
4 doubles = 256 bit
8 doubles = 512 bit

Time
### Core 2
- Has SSE3
- 16 SSE registers

128 bit = 2 doubles = 4 singles

<table>
<thead>
<tr>
<th>xmm0</th>
<th>xmm8</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmm1</td>
<td>xmm9</td>
</tr>
<tr>
<td>xmm2</td>
<td>xmm10</td>
</tr>
<tr>
<td>xmm3</td>
<td>xmm11</td>
</tr>
<tr>
<td>xmm4</td>
<td>xmm12</td>
</tr>
<tr>
<td>xmm5</td>
<td>xmm13</td>
</tr>
<tr>
<td>xmm6</td>
<td>xmm14</td>
</tr>
<tr>
<td>xmm7</td>
<td>xmm15</td>
</tr>
</tbody>
</table>

### SSE3 Registers
- Used for different data types and instructions
- Integer vectors:
  - 16-way byte
  - 8-way 2 bytes
  - 4-way 4 bytes
  - 2-way 8 bytes

- Floating point vectors:
  - 4-way single (since SSE)
  - 2-way double (since SSE2)

- Floating point scalars:
  - Single (since SSE)
  - Double (since SSE2)
SSE3 Instructions: Examples

- Single precision 4-way vector add: `addps %xmm0 %xmm1`

- Single precision scalar add: `addss %xmm0 %xmm1`

SSE3 Instruction Names

- `addps`: packed (vector)
  - single precision

- `addpd`: double precision

- `addss`: single slot (scalar)
  - Compiler will use this for floating point
    - on x86-64
    - with proper flags if SSE/SSE2 is available

- `addsd`: double precision scalar
x86-64 FP Code Example

- **Inner product of two vectors**
  - Single precision arithmetic
  - Compiled: not vectorized, uses SSE instructions

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

SSE: 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 single precision (4-way)
SIMD Extensions and SSE

- Overview: SSE family
- **SSE intrinsics**
- Compiler vectorization

References:
*Intel Intrinsics Guide*
(easy access to all instructions, also contains latency and throughput information!)

Intel icc compiler manual
Visual Studio manual

SSE Family: Floating Point

- Not drawn to scale
- From SSE2: Only additional instructions
- *Every Core 2 has SSE3*
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 on icc
  - Most intrinsics work with gcc and Visual Studio (VS)
  - Some language extensions are icc (or even VS) specific

Visual Conventions We Will Use

- Memory
  - Increasing address
  - Memory

- Registers
  - Commonly:
    - R3 R2 R1 R0
  - We will use
    - R0 R1 R2 R3
SSE Intrinsics (Focus Floating Point)

- Data types
  ```
  __m128  f;  // = {float f0, f1, f2, f3}
  __m128d d;  // = {double d0, d1}
  __m128i i;  // 16 8-bit, 8 16-bit, 4 32-bit, or 2 64-bit ints
  ```

- Instructions
  - Naming convention: `_mm_<intrin_op>_ <suffix>`
  - Example:
    ```
    // a is 16-byte aligned
    float a[4] = {1.0, 2.0, 3.0, 4.0};
    __m128 t = _mm_load_ps(a);
    ```
    ```
    LSB 1.0 2.0 3.0 4.0
    ```
    - Same result as
      ```
      __m128 t = _mm_set_ps(4.0, 3.0, 2.0, 1.0)
      ```
SSE Intrinsics

- Native instructions (one-to-one with assembly)
  \_mm_load_ps() ↔ movaps
  \_mm_add_ps() ↔ addps
  \_mm_mul_ps() ↔ mulpd
  ...
- Multi instructions (map to several assembly instructions)
  \_mm_set_ps()
  \_mm_set1_ps()
  ...
- Macros and helpers
  \_MM_TRANSPOSE4_PS()
  \_MM_SHUFFLE()
  ...

What Are the Main Issues?

- Alignment is important (128 bit = 16 byte)
- You need to code explicit loads and stores
- Overhead through shuffles
**SSE vs. AVX**

<table>
<thead>
<tr>
<th></th>
<th>SSE</th>
<th>AVX</th>
</tr>
</thead>
<tbody>
<tr>
<td>float, double</td>
<td>4-way, 2-way</td>
<td>8-way, 4-way</td>
</tr>
<tr>
<td>register</td>
<td>16 x 128 bits: %xmm0 - %xmm15</td>
<td>16 x 256 bits: %ymm0 - %ymm15 The lower halves are the %xmms</td>
</tr>
<tr>
<td>assembly ops</td>
<td>addps, mulpd, ...</td>
<td>vaddps, vmulpd</td>
</tr>
<tr>
<td>intrinsics data type</td>
<td>__m128, __m128d</td>
<td>__m256, __m256d</td>
</tr>
<tr>
<td>intrinsics instructions</td>
<td>_mm_load_ps, _mm_add_pd, ...</td>
<td>_mm256_load_ps, _mm256_add_pd</td>
</tr>
</tbody>
</table>

*Mixing SSE and AVX may incur penalties*
SSE Intrinsics

- Load and store
- Constants
- Arithmetic
- Comparison
- Conversion
- Shuffles

Loads and Stores

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_loadh_pi</td>
<td>Load high</td>
<td>MOVHPS reg, mem</td>
</tr>
<tr>
<td>_mm_loadl_pi</td>
<td>Load low</td>
<td>MOVLPS reg, mem</td>
</tr>
<tr>
<td>_mm_load_ss</td>
<td>Load the low value and clear the three high values</td>
<td>MOVSS</td>
</tr>
<tr>
<td>_mm_load1_ps</td>
<td>Load one value into all four words</td>
<td>MOVAPS</td>
</tr>
<tr>
<td>_mm_load_ps</td>
<td>Load four values, address aligned</td>
<td>MOVAPS</td>
</tr>
<tr>
<td>_mm_loadu_ps</td>
<td>Load four values, address unaligned</td>
<td>MOVUPS</td>
</tr>
<tr>
<td>_mm_loadr_ps</td>
<td>Load four values in reverse</td>
<td>MOVAPS + Shuffling</td>
</tr>
<tr>
<td>_mm_set_ss</td>
<td>Set the low value and clear the three high values</td>
<td>Composite</td>
</tr>
<tr>
<td>_mm_set1_ps</td>
<td>Set all four words with the same value</td>
<td>Composite</td>
</tr>
<tr>
<td>_mm_set_ps</td>
<td>Set four values, address aligned</td>
<td>Composite</td>
</tr>
<tr>
<td>_mm_setr_ps</td>
<td>Set four values, in reverse order</td>
<td>Composite</td>
</tr>
<tr>
<td>_mm_setzero_ps</td>
<td>Clear all four values</td>
<td>Composite</td>
</tr>
</tbody>
</table>
Loads and Stores

\[
\begin{array}{cccc}
1.0 & 2.0 & 3.0 & 4.0 \\
\text{memory} \\
1.0 & 2.0 & 3.0 & 4.0 \\
\text{LSB} \\
\end{array}
\]\n
\[
a = \_\_\text{mm}\_\text{load}\_\text{ps}(p); \quad // \text{p 16-byte aligned}
\]
\[
a = \_\_\text{mm}\_\text{loadu}\_\text{ps}(p); \quad // \text{p not aligned}
\]

Avoid (can be expensive) on recent Intel possibly no penalty

Load_ps on unaligned pointer: seg fault

\[
\begin{array}{cccc}
1.0 & 2.0 & 3.0 & 4.0 \\
\end{array}
\]

\[
\begin{array}{cccc}
1.0 & 2.0 & 1.0 & 2.0 \\
\text{memory} \\
1.0 & 2.0 & 1.0 & 2.0 \\
\text{LSB} \\
\end{array}
\]

\[
a = \_\_\text{mm}\_\text{loadl}\_\text{pi}(a, p); \quad // \text{p 8-byte aligned}
\]
\[
a = \_\_\text{mm}\_\text{loadh}\_\text{pi}(a, p); \quad // \text{p 8-byte aligned}
\]
Loads and Stores

\[ a = \_\text{mm\_load\_ss}(p); \quad // \quad p \text{ any alignment} \]

Stores Analogous to Loads

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_storeh_pi</td>
<td>Store high</td>
<td>MOVHPS mem, reg</td>
</tr>
<tr>
<td>_mm_storel_pi</td>
<td>Store low</td>
<td>MOVLPS mem, reg</td>
</tr>
<tr>
<td>_mm_store_ss</td>
<td>Store the low value</td>
<td>MOVSS</td>
</tr>
<tr>
<td>_mm_store1_ps</td>
<td>Store the low value across all four words, address aligned</td>
<td>Shuffling + MOVSS</td>
</tr>
<tr>
<td>_mm_store_ps</td>
<td>Store four values, address aligned</td>
<td>MOVAPS</td>
</tr>
<tr>
<td>_mm_storeu_ps</td>
<td>Store four values, address unaligned</td>
<td>MOVUPS</td>
</tr>
<tr>
<td>_mm_storer_ps</td>
<td>Store four values, in reverse order</td>
<td>MOVAPS + Shuffling</td>
</tr>
</tbody>
</table>
### Constants

- \(a = \text{_mm_set_ps}(4.0, 3.0, 2.0, 1.0);\)
- \(b = \text{_mm_set1_ps}(1.0);\)
- \(c = \text{_mm_set_ss}(1.0);\)
- \(d = \text{_mm_setzero_ps}();\)

### Arithmetic

#### SSE

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_add_ss</td>
<td>Addition</td>
<td>ADDSS</td>
</tr>
<tr>
<td>_mm_add_ps</td>
<td>Addition</td>
<td>ADDPS</td>
</tr>
<tr>
<td>_mm_sub_ss</td>
<td>Subtraction</td>
<td>SUBSS</td>
</tr>
<tr>
<td>_mm_sub_ps</td>
<td>Subtraction</td>
<td>SUBPS</td>
</tr>
<tr>
<td>_mm_mul_ss</td>
<td>Multiplication</td>
<td>MULSS</td>
</tr>
<tr>
<td>_mm_mul_ps</td>
<td>Multiplication</td>
<td>MULPS</td>
</tr>
<tr>
<td>_mm_div_ss</td>
<td>Division</td>
<td>DIVSS</td>
</tr>
<tr>
<td>_mm_div_ps</td>
<td>Division</td>
<td>DIVPS</td>
</tr>
<tr>
<td>_mm_sqrt_ss</td>
<td>Squared Root</td>
<td>SQRTSS</td>
</tr>
<tr>
<td>_mm_sqrt_ps</td>
<td>Squared Root</td>
<td>SQRTPS</td>
</tr>
<tr>
<td>_mm_rcp_ss</td>
<td>Reciprocal</td>
<td>RCPSS</td>
</tr>
<tr>
<td>_mm_rcp_ps</td>
<td>Reciprocal</td>
<td>RCPPS</td>
</tr>
<tr>
<td>_mm_rsqrt_ss</td>
<td>Reciprocal Squared Root</td>
<td>RISQRTSS</td>
</tr>
<tr>
<td>_mm_rsqrt_ps</td>
<td>Reciprocal Squared Root</td>
<td>RISQRTPS</td>
</tr>
<tr>
<td>_mm_min_ss</td>
<td>Computes Minimum</td>
<td>MINSS</td>
</tr>
<tr>
<td>_mm_min_ps</td>
<td>Computes Minimum</td>
<td>MINPS</td>
</tr>
<tr>
<td>_mm_max_ss</td>
<td>Computes Maximum</td>
<td>MAXSS</td>
</tr>
<tr>
<td>_mm_max_ps</td>
<td>Computes Maximum</td>
<td>MAXPS</td>
</tr>
</tbody>
</table>

#### SSE3

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE3 Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_addsub_ps</td>
<td>Subtract and add</td>
<td>ADDSUBPS</td>
</tr>
<tr>
<td>_mm_hadd_ps</td>
<td>Add</td>
<td>HADDP PS</td>
</tr>
<tr>
<td>_mm_hsub_ps</td>
<td>Subtracts</td>
<td>HSUBPS</td>
</tr>
</tbody>
</table>

#### SSE4

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Operation</th>
<th>Corresponding SSE4 Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_dp_ps</td>
<td>Single precision dot product</td>
<td>DPPS</td>
</tr>
</tbody>
</table>
Arithmetic

\[ \begin{array}{cccc}
1.0 & 2.0 & 3.0 & 4.0 \\
& + & + & + \\
0.5 & 1.5 & 2.5 & 3.5 \\
\end{array} \]

\[ c = _{\text{mm_add_ps}}(a, b); \]

**analogous:**

\[ c = _{\text{mm_sub_ps}}(a, b); \]
\[ c = _{\text{mm_mul_ps}}(a, b); \]

---

Example

```c
#include <ia32intrin.h>

// n a multiple of 4, x is 16-byte aligned
void addindex_vec(float *x, int n) {
    __m128 index, x_vec;
    for (int i = 0; i < n; i+=4) {
        x_vec = _mm_load_ps(x+i); // load 4 floats
        index = _mm_set_ps(i+3, i+2, i+1, i); // create vector with indexes
        x_vec = _mm_add_ps(x_vec, index); // add the two
        _mm_store_ps(x+i, x_vec); // store back
    }
}
```

Is this the best solution?

*No! _mm_set_ps may be too expensive*
Example

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

```c
#include <ia32intrin.h>

// n a multiple of 4, x is 16-byte aligned
void addindex_vec(float *x, int n) {
    __m128 x_vec, init, incr;
    ind = _mm_set_ps(3, 2, 1, 0);
    incr = _mm_set1_ps(4);
    for (int i = 0; i < n; i+=4) {
        x_vec = _mm_load_ps(x+i); // load 4 floats
        x_vec = _mm_add_ps(x_vec, ind); // add the two
        ind = _mm_add_ps(ind, incr); // update ind
        _mm_store_ps(x+i, x_vec); // store back
    }
}
```

Code style helps with performance!

Arithmetic

```
1.0 2.0 3.0 4.0  a
  +
1.5 2.0 3.0 4.0  c
  +
0.5       b
```

```
c = _mm_add_ss(a, b);
```
Arithmetic

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

Arithmetic

```
c = _mm_addsub_ps(a, b);
```
Arithmetic

\[ a = \begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 \end{bmatrix} \]
\[ b = \begin{bmatrix} 0.5 & 1.5 & 2.5 & 3.5 \end{bmatrix} \]
\[ c = \begin{bmatrix} 3.0 & 7.0 & 2.0 & 6.0 \end{bmatrix} \]

\[
c = \_\_m\_hadd\_ps(a, b);
\]

**analogous:**

\[
c = \_\_m\_hsub\_ps(a, b);
\]

Example

```c
#include <ia32intrin.h>

// n is even
void lp(float *x, float *y, int n) {
    for (int i = 0; i < n/2; i++)
        y[i] = (x[2*i] + x[2*i+1])/2;
}

// n a multiple of 8, x, y are 16-byte aligned
void lp_vec(float *x, int n) {
    __m128 half, v1, v2, avg;
    half = _mm_set1_ps(0.5); // set vector to all 0.5
    for(int i = 0; i < n/8; i++) {
        v1 = _mm_load_ps(x+i*8); // load first 4 floats
        v2 = _mm_load_ps(x+4+i*8); // load next 4 floats
        avg = _mm_hadd_ps(v1, v2); // add pairs of floats
        avg = _mm_mul_ps(avg, half); // multiply with 0.5
        _mm_store_ps(y+i*4, avg); // save result
    }
}
```
Arithmetic

\[
\text{\_m128 \_mm\_dp\_ps(\_m128 a, \_m128 b, const int mask)}
\]

(SSE4) 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.

**Example:** mask = 117 = 01110101

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

\[
\begin{array}{cccc}
0.5 & 3.0 & 7.5 & 14.0 \\
\hline
11.0 & 0 & 11.0 & 0 \\
\end{array}
\]

Comparisons

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_cmpeq_ss</td>
<td>Equal</td>
<td>CMPEQSS</td>
</tr>
<tr>
<td>_mm_cmpeq_ps</td>
<td>Equal</td>
<td>CMPEQPS</td>
</tr>
<tr>
<td>_mm_cmplt_ss</td>
<td>Less Than</td>
<td>CMPLTSS</td>
</tr>
<tr>
<td>_mm_cmplt_ps</td>
<td>Less Than</td>
<td>CMPLTPS</td>
</tr>
<tr>
<td>_mm_cmple_ss</td>
<td>Less Than or Equal</td>
<td>CMPELSS</td>
</tr>
<tr>
<td>_mm_cmple_ps</td>
<td>Less Than or Equal</td>
<td>CMPELPS</td>
</tr>
<tr>
<td>_mm_cmpgt_ss</td>
<td>Greater Than</td>
<td>CMPGTSS</td>
</tr>
<tr>
<td>_mm_cmpgt_ps</td>
<td>Greater Than</td>
<td>CMPGTPS</td>
</tr>
<tr>
<td>_mm_cmpge_ss</td>
<td>Greater Than or Equal</td>
<td>CMPGESS</td>
</tr>
<tr>
<td>_mm_cmpge_ps</td>
<td>Greater Than or Equal</td>
<td>CMPGEPSS</td>
</tr>
<tr>
<td>_mm_cmpeq_ss</td>
<td>Not Equal</td>
<td>CMPEQSS</td>
</tr>
<tr>
<td>_mm_cmpeq_ps</td>
<td>Not Equal</td>
<td>CMPEQPS</td>
</tr>
<tr>
<td>_mm_cmplt_ss</td>
<td>Not Less Than</td>
<td>CMPLTSS</td>
</tr>
<tr>
<td>_mm_cmplt_ps</td>
<td>Not Less Than</td>
<td>CMPLTPS</td>
</tr>
<tr>
<td>_mm_cmple_ss</td>
<td>Not Less Than or Equal</td>
<td>CMPELSS</td>
</tr>
<tr>
<td>_mm_cmple_ps</td>
<td>Not Less Than or Equal</td>
<td>CMPELPS</td>
</tr>
<tr>
<td>_mm_cmpgt_ss</td>
<td>Not Greater Than</td>
<td>CMPGTSS</td>
</tr>
<tr>
<td>_mm_cmpgt_ps</td>
<td>Not Greater Than</td>
<td>CMPGTPS</td>
</tr>
<tr>
<td>_mm_cmpge_ss</td>
<td>Not Greater Than or Equal</td>
<td>CMPGESS</td>
</tr>
<tr>
<td>_mm_cmpge_ps</td>
<td>Not Greater Than or Equal</td>
<td>CMPGEPSS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_cmpord_ss</td>
<td>Ordered</td>
<td>CMPORDSS</td>
</tr>
<tr>
<td>_mm_cmpord_ps</td>
<td>Ordered</td>
<td>CMPORDPS</td>
</tr>
<tr>
<td>_mm_cmpunord_ss</td>
<td>Unordered</td>
<td>CMPUNORDSS</td>
</tr>
<tr>
<td>_mm_cmpunord_ps</td>
<td>Unordered</td>
<td>CMPUNORDPS</td>
</tr>
<tr>
<td>_mm_comieq_ss</td>
<td>Equal</td>
<td>COMISS</td>
</tr>
<tr>
<td>_mm_comile_ss</td>
<td>Less Than or Equal</td>
<td>COMISS</td>
</tr>
<tr>
<td>_mm_comile_ps</td>
<td>Less Than or Equal</td>
<td>COMISS</td>
</tr>
<tr>
<td>_mm_comigt_ss</td>
<td>Greater Than</td>
<td>COMISS</td>
</tr>
<tr>
<td>_mm_comigt_ps</td>
<td>Greater Than</td>
<td>COMISS</td>
</tr>
<tr>
<td>_mm_comineq_ss</td>
<td>Not Equal</td>
<td>COMISS</td>
</tr>
<tr>
<td>_mm_comineq_ps</td>
<td>Not Equal</td>
<td>COMISS</td>
</tr>
<tr>
<td>_mm_ucomieq_ss</td>
<td>Equal</td>
<td>UCOMISS</td>
</tr>
<tr>
<td>_mm_ucomile_ss</td>
<td>Less Than</td>
<td>UCOMISS</td>
</tr>
<tr>
<td>_mm_ucomile_ps</td>
<td>Less Than</td>
<td>UCOMISS</td>
</tr>
<tr>
<td>_mm_ucomigt_ss</td>
<td>Greater Than</td>
<td>UCOMISS</td>
</tr>
<tr>
<td>_mm_ucomigt_ps</td>
<td>Greater Than</td>
<td>UCOMISS</td>
</tr>
<tr>
<td>_mm_ucomineq_ss</td>
<td>Not Equal</td>
<td>UCOMISS</td>
</tr>
<tr>
<td>_mm_ucomineq_ps</td>
<td>Not Equal</td>
<td>UCOMISS</td>
</tr>
</tbody>
</table>
Comparisons

<table>
<thead>
<tr>
<th>a</th>
<th>1.0</th>
<th>2.0</th>
<th>3.0</th>
<th>4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>1.0</td>
<td>1.5</td>
<td>3.0</td>
<td>3.5</td>
</tr>
<tr>
<td>c</td>
<td>0x0</td>
<td>0x0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

c = \_mm_cmpeq_ps(a, b);

**analogous:**

c = \_mm_cmple_ps(a, b);

c = \_mm_cmpgt_ps(a, b);

c = \_mm_cmplt_ps(a, b);

etc.

Each field:

- 0xffffffff if true
- 0x0 if false

Return type: \_m128

Example

```c
#include <xmmintrin.h>

void fcond(float *x, size_t n) {
  int i;
  for(i = 0; i < n; i++) {
    if(x[i] > 0.5)
      x[i] += 1.;
    else x[i] -= 1.;
  }
}

void fcond(float *x, size_t n) {
  __m128 vt, vmask, vp, vm, vr, ones, mones, thresholds;
  ones = \_mm_set1_ps(1.);
  mones = \_mm_set1_ps(-1.);
  thresholds = \_mm_set1_ps(0.5);
  for(i = 0; i < n; i+=4) {
    vt = \_mm_load_ps(x+i);
    vmask = \_mm_cmplt_ps(vt, thresholds);
    vp = \_mm_and_ps(vmask, ones);
    vm = \_mm_andnot_ps(vmask, mones);
    vr = \_mm_add_ps(vt, \_mm_or_ps(vp, vm));
    \_mm_store_ps(x+i, vr);
  }
}
```
**Conversion**

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_cvtsi32_si32</td>
<td>Convert to 32-bit integer</td>
<td>CVTSI2SS</td>
</tr>
<tr>
<td>_mm_cvtsi64_si64*</td>
<td>Convert to 64-bit integer</td>
<td>CVTSI2SS</td>
</tr>
<tr>
<td>_mm_cvtps_pi32</td>
<td>Convert to two 32-bit integers</td>
<td>CVTTPS2PI</td>
</tr>
<tr>
<td>_mm_cvttss_si32</td>
<td>Convert to 32-bit integer</td>
<td>CVTTSS2SI</td>
</tr>
<tr>
<td>_mm_cvttss_si64*</td>
<td>Convert to 64-bit integer</td>
<td>CVTTSS2SI</td>
</tr>
<tr>
<td>_mm_cvtpi32_ps</td>
<td>Convert from two 32-bit integers</td>
<td>CVTTPI2PS</td>
</tr>
<tr>
<td>_mm_cvtpi16_ps</td>
<td>Convert from four 16-bit integers</td>
<td>composite</td>
</tr>
<tr>
<td>_mm_cvtpu16_ps</td>
<td>Convert from four 16-bit integers</td>
<td>composite</td>
</tr>
<tr>
<td>_mm_cvtpi8_ps</td>
<td>Convert from four 8-bit integers</td>
<td>composite</td>
</tr>
<tr>
<td>_mm_cvtpu8_ps</td>
<td>Convert from four 8-bit integers</td>
<td>composite</td>
</tr>
<tr>
<td>_mm_cvtps_pi16</td>
<td>Convert to four 16-bit integers</td>
<td>composite</td>
</tr>
<tr>
<td>_mm_cvtps_pi8</td>
<td>Convert to four 8-bit integers</td>
<td>composite</td>
</tr>
<tr>
<td>_mm_cvtsi_f32</td>
<td>Extract</td>
<td>composite</td>
</tr>
</tbody>
</table>

*Picture: www.druckundbestell.de*
Conversion

```c
float _mm_cvtss_f32(__m128 a)
```

<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>

```
f = _mm_cvtss_f32(a);
```

Cast

interpreted as

```
__m128i __m_castps_si128(__m128 a)
```

Reinterprets the four single precision floating point values in a as four 32-bit integers, and vice versa.

*No conversion is performed. Does not map to any assembly instructions.*

Makes integer shuffle instructions usable for floating point.
Actual Conversion

```c
__m128 _mm_cvt_pi2ps (__m128 a, __m64 b)
```

convert

ints  floats

Shuffles

### SSE

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_shuffle_ps</td>
<td>Shuffle</td>
<td>SHUFPS</td>
</tr>
<tr>
<td>_mm_unpackhi_ps</td>
<td>Unpack High</td>
<td>UNPCKHPS</td>
</tr>
<tr>
<td>_mm_unpacklo_ps</td>
<td>Unpack Low</td>
<td>UNPCKLPS</td>
</tr>
<tr>
<td>_mm_move_ss</td>
<td>Set low word, pass in three high values</td>
<td>MOVS</td>
</tr>
<tr>
<td>_mm_movehl_ps</td>
<td>Move High to Low</td>
<td>MOVHLPS</td>
</tr>
<tr>
<td>_mm_movelh_ps</td>
<td>Move Low to High</td>
<td>MOVLHPS</td>
</tr>
<tr>
<td>_mm_movemask_ps</td>
<td>Create four-bit mask</td>
<td>MOVMSKPS</td>
</tr>
</tbody>
</table>

### SSE3

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE3 Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_movehdup_ps</td>
<td>Duplicates</td>
<td>MOVSHDUP</td>
</tr>
<tr>
<td>_mm_moveldup_ps</td>
<td>Duplicates</td>
<td>MOVSLDUP</td>
</tr>
</tbody>
</table>

### SSE4

<table>
<thead>
<tr>
<th>Intrinsic Syntax</th>
<th>Operation</th>
<th>Corresponding SSE4 Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>__m128 _mm_blend_ps(__m128 v1, __m128 v2, const int mask)</td>
<td>Selects float single precision data from 2 sources using constant mask</td>
<td>BLENDPS</td>
</tr>
<tr>
<td>__m128 _mm_blendv_ps(__m128 v1, __m128 v2, __m128 v3)</td>
<td>Selects float single precision data from 2 sources using variable mask</td>
<td>BLENDVPS</td>
</tr>
<tr>
<td>__m128 _mm_insert_ps(__m128 dst, __m128 src, const int ndx)</td>
<td>Insert single precision float into packed single precision array element selected by index.</td>
<td>INSERTPS</td>
</tr>
<tr>
<td>int _mm_extract_ps(__m128 src, const int ndx)</td>
<td>Extract single precision float from packed single precision array selected by index.</td>
<td>EXTRACTPS</td>
</tr>
</tbody>
</table>
Shuffles

\[
c = \_\_mm\_unpacklo\_ps(a, b);
\]

\[
c = \_\_mm\_unpackhi\_ps(a, b);
\]

\[
c = \_\_mm\_shuffle\_ps(a, b, \_\_MM\_SHUFFLE(l, k, j, i));
\]

Helper macro to create mask:

\[
c0 = a_i \\
c1 = a_j \\
c2 = b_k \\
c3 = b_l \\
i,j,k,l \in \{0,1,2,3\}
\]
Example: Loading 4 Real Numbers from Arbitrary Memory Locations

1.0
2.0
3.0
4.0

p0
p1
p2
p3

LSB

7 instructions, this is one good way of doing it

Code For Previous Slide

```c
#include <ia32intrin.h>

__m128 LoadArbitrary(float *p0, float *p1, float *p2, float *p3) {
    __m128 a, b, c, d, e, f;
    a = __mm_load_ss(p0);
    b = __mm_load_ss(p1);
    c = __mm_load_ss(p2);
    d = __mm_load_ss(p3);
    e = __mm_shuffle_ps(a, b, _MM_SHUFFLE(1,0,2,0)); //only zeros are important
    f = __mm_shuffle_ps(c, d, _MM_SHUFFLE(1,0,2,0)); //only zeros are important
    return __mm_shuffle_ps(e, f, _MM_SHUFFLE(2,0,2,0));
}
```
Example: Loading 4 Real Numbers from Arbitrary Memory Locations (cont’d)

- Whenever possible avoid the previous situation
- Restructure algorithm and use the aligned \_mm_load_ps()
- Other possibility (but likely also yields 7 instructions)

```
__m128 vf;
vf = __mm_set_ps(*p3, *p2, *p1, *p0);
```

- SSE4: \_mm_insert_epi32 together with \_mm_castsi128_ps
  - Not clear whether better

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

- Do not do this (why?):

```
__declspec(align(16)) float g[4];
__m128 vf;
g[0] = *p0;
g[1] = *p1;
g[2] = *p2;
g[3] = *p3;
vf = __mm_load_ps(g);
```
Example: Storing 4 Real Numbers to Arbitrary Memory Locations

7 instructions, shorter critical path

Shuffle

\[
\_\text{m128i } \_\text{mm_alignr_epi8}(\_\text{m128i } a, \_\text{m128i } b, \text{const int } n)
\]

Concatenate a and b and extract byte-aligned result shifted to the right by n bytes

*Example*: View \_\text{m128i} as 4 32-bit ints; \( n = 12 \)

*How to use this with floating point vectors?*

*Use with \_\text{mm_castsi128_ps}!*
Example

```c
#include <ia32intrin.h>

// n a multiple of 4, x, y are 16-byte aligned
void shift_vec(float *x, float *y, int n) {
    __m128 f;
    __m128i i1, i2, i3;
    i1 = _mm_castps_si128(_mm_load_ps(x)); // load first 4 floats and cast to int
    for (int i = 0; i < n-8; i = i + 4) {
        i2 = _mm_castps_si128(_mm_load_ps(x+i+4)); // load next 4 floats and cast to int
        f = _mm_castsi128_ps(_mm_alignr_epi8(i2,i1,4)); // shift and extract and cast back
        _mm_store_ps(y+i,f); // store it
        i1 = i2; // make 2nd element 1st
    }
    // we are at the last 4
    i2 = _mm_castsi128_ps(_mm_setzero_ps()); // set the second vector to 0 and cast to int
    f = _mm_castsi128_ps(_mm_alignr_epi8(i2,i1,4)); // shift and extract and cast back
    _mm_store_ps(y+n-4,f); // store it
}
```

Shuffle

```c
__m128i _mm_shuffle_epi8(__m128i a, __m128i mask)
```

Result is filled in each position by any element of a or with 0, as specified by mask

**Example:** View __m128i as 4 32-bit ints

![Example diagram](image)

**Use with _mm_castsi128_ps to do the same for floating point**
Shuffle

```c
__m128 __mm_blendv_ps(__m128 a, __m128 b, __m128 mask)
```

(SSE4) Result is filled in each position by an element of a or b in the same position as specified by mask

**Example:**

<table>
<thead>
<tr>
<th>mask</th>
<th>0x0</th>
<th>0x0</th>
<th>0x0</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1.0</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>b</td>
<td>0.5</td>
<td>1.5</td>
<td>2.5</td>
</tr>
<tr>
<td>c</td>
<td>1.0</td>
<td>1.5</td>
<td>3.0</td>
</tr>
</tbody>
</table>

see also __mm_blend_ps

---

Example (Continued From Before)

```c
#include <xmmintrin.h>

void fcond(float *x, size_t n) {
    int i;
    for(i = 0; i < n; i++) {
        if(x[i] > 0.5)
            x[i] += 1.;
        else
            x[i] -= 1.;
    }
}

void fcond(float *a, size_t n) {
    __m128 vt, vmask, vp, vm, vr, ones, mones, thresholds;
    ones = __mm_set1_ps(1.);
    mones = __mm_set1_ps(-1.);
    thresholds = __mm_set1_ps(0.5);
    for(i = 0; i < n; i+=4) {
        vt = __mm_load_ps(a+i);
        vmask = __mm_cmpgt_ps(vt, thresholds);
        vb = __mm_blendv_ps(mones, ones, vmask);
        vr = __mm_add_ps(vt, vb);
        __mm_store_ps(a+i, vr);
    }
}
```
Shuffle

(shader_function)

_MBM_TRANSPOSE4_PS(row0, row1, row2, row3)_

**Macro for 4 x 4 matrix transposition:** The arguments row0,…, row3 are __m128 values each containing a row of a 4 x 4 matrix. After execution, row0,.., row 3 contain the columns of that matrix.

<table>
<thead>
<tr>
<th>LSB</th>
<th>1.0</th>
<th>2.0</th>
<th>3.0</th>
<th>4.0</th>
<th>row0</th>
<th>LSB</th>
<th>1.0</th>
<th>5.0</th>
<th>9.0</th>
<th>13.0</th>
<th>row0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB</td>
<td>5.0</td>
<td>6.0</td>
<td>7.0</td>
<td>8.0</td>
<td>row1</td>
<td>LSB</td>
<td>2.0</td>
<td>6.0</td>
<td>10.0</td>
<td>14.0</td>
<td>row1</td>
</tr>
<tr>
<td>LSB</td>
<td>9.0</td>
<td>10.0</td>
<td>11.0</td>
<td>12.0</td>
<td>row2</td>
<td>LSB</td>
<td>3.0</td>
<td>7.0</td>
<td>11.0</td>
<td>15.0</td>
<td>row2</td>
</tr>
<tr>
<td>LSB</td>
<td>13.0</td>
<td>14.0</td>
<td>15.0</td>
<td>16.0</td>
<td>row3</td>
<td>LSB</td>
<td>4.0</td>
<td>8.0</td>
<td>12.0</td>
<td>16.0</td>
<td>row3</td>
</tr>
</tbody>
</table>

**In SSE:** 8 shuffles (4_mm_unpacklo_ps, 4_mm_unpackhi_ps)

---

**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_ps has higher throughput than hadd_ps
- Be aware of available instructions ([intrinsics guide!])
SIMD Extensions and SSE

- SSE 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: mulps, addps, xxxps
- Generate assembly with source code annotation:
  - Visual Studio + icc: /Fas
  - icc on Linux/Mac: -S

Example

unvectorized: /Qvec-

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

vectorized:

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

why this?

why everything twice?

why movups and movaps?

unaligned
aligned
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 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`, `-falias`
  - `-fargument-noalias`, `-falias-args` (function arguments only)

- **For one loop: pragma**

```c
void add(float *a, float *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(float *restrict a, float *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(float *a, float *b, float *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(float *a, float *b, float *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(float *a, float *b, float *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

```c
float *x = (float *) malloc(1024*sizeof(float));
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:

```c
float *x = (float *) malloc(1024*sizeof(float));
int i;
peel = x & 0x0f; /* x mod 16 */
if (peel != 0) {
  peel = 16 - peel;
  /* initial segment */
  for (i = 0; i < peel; i++)
    x[i] = 1;
}
/* 16-byte aligned access */
for (i = peel; i < 1024; i++)
x[i] = 1;
```

Ensuring Alignment

- Align arrays to 16-byte boundaries (see earlier discussion)
- If compiler cannot analyze:
  - Use pragma for loops
    ```c
    float *x = (float *) malloc(1024*sizeof(float));
    int i;
    #pragma vector aligned
    for (i = 0; i < 1024; i++)
      x[i] = 1;
    ```
  - For specific arrays:
    ```c
    __assume_aligned(a, 16);
    ```
More Tips (icc 14.0)  
https://software.intel.com/en-us/node/512631

- 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 16 byte boundaries (for SSE instructions).

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

- ...

...
Compiler Vectorization

- Read manual