Flynn’s Taxonomy

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

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

- Has SSE3
- 16 SSE registers

128 bit = 2 doubles = 4 singles

SSE3 Registers

- Different data types and associated 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)

- `addsd`: double precision

Compiler will use this for floating point
- on x86-64
- with proper flags if SSE/SSE2 is available
x86-64 FP Code Example

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

```asm
xorps %xmm1, %xmm1
xorl %ecx, %ecx
jmp .L8
.L10:
movslq %ecx,%rax
incl %ecx
movss (%rsi,%rax,4), %xmm0
mulss (%rdi,%rax,4), %xmm0
addss %xmm0, %xmm1
.L8:
  cmpl %edx, %ecx
  jl .L10
ret
```

```cpp
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:
    - R0 R1 R2 R3
  - We will use
    - LSB
    - 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)
  \[
  \text{\_mm\_load\_ps()} \leftrightarrow \text{movaps} \\
  \text{\_mm\_add\_ps()} \leftrightarrow \text{addps} \\
  \text{\_mm\_mul\_pd()} \leftrightarrow \text{mulpd} \\
  \ldots
  \]

- Multi instructions (map to several assembly instructions)
  \[
  \text{\_mm\_set\_ps()} \\
  \text{\_mm\_set1\_ps()} \\
  \ldots
  \]

- Macros and helpers
  \[
  \text{\_MM\_TRANSPOSE4\_PS()} \\
  \text{\_MM\_SHUFFLE()} \\
  \ldots
  \]

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 %xms</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 + Shuffling</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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE Instruction</th>
</tr>
</thead>
<tbody>
<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

\[ a = \_mm\_load\_ps(p); \] // p 16-byte aligned
\[ a = \_mm\_loadu\_ps(p); \] // p not aligned

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

load\_ps on unaligned pointer: seg fault

\[ a = \_mm\_loadl\_pi(a, p); \] // p 8-byte aligned
\[ a = \_mm\_loadh\_pi(a, p); \] // p 8-byte aligned
**Loads and Stores**

```
loads
memory

1.0

p

LSB

1.0 0 0 0

a

a = _mm_load_ss(p); // p 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

<table>
<thead>
<tr>
<th>LSB</th>
<th>1.0</th>
<th>2.0</th>
<th>3.0</th>
<th>4.0</th>
<th>a = _mm_set_ps(4.0, 3.0, 2.0, 1.0);</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>b = _mm_set1_ps(1.0);</td>
</tr>
<tr>
<td>LSB</td>
<td>1.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>c = _mm_set_ss(1.0);</td>
</tr>
<tr>
<td>LSB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>d = _mm_setzero_ps();</td>
</tr>
</tbody>
</table>

### 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>RISQRTPS</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>HADDPs</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

\[ a = \begin{bmatrix} 1.0 \ 2.0 \ 3.0 \ 4.0 \end{bmatrix} \quad b = \begin{bmatrix} 0.5 \ 1.5 \ 2.5 \ 3.5 \end{bmatrix} \]

\[ c = \begin{bmatrix} 1.5 \ 3.5 \ 5.5 \ 7.5 \end{bmatrix} \]

\[
c = \_\text{mm}_\text{add}_\text{ps}(a, b);
\]

*analogous:*

\[
c = \_\text{mm}_\text{sub}_\text{ps}(a, b);
\]

\[
c = \_\text{mm}_\text{mul}_\text{ps}(a, b);
\]

Example

```c
#include <ia32intrin.h>

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

void addindex_vec(float *x, int n) {
    __m128 index, x_vec;
    for (int i = 0; i < n; i+=4) {
        x_vec = \_\text{mm}_\text{load}_\text{ps}(x+i);
        index = \_\text{mm}_\text{set}_\text{ps}(i+3, i+2, i+1, i); // create vector with indexes
        x_vec = \_\text{mm}_\text{add}_\text{ps}(x_vec, index); // add the two
        \_\text{mm}_\text{store}_\text{ps}(x+i, x_vec); // store back
    }
}
```

Is this the best solution?

*No! \_\text{mm}_\text{set}_\text{ps} may be too expensive*
Example

```c
#include <ia32intrin.h>

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);
        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
    }
}
```

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

Code style helps with performance!

Arithmetic

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

\[ c = \text{\_mm\_max\_ps}(a, b); \]

Arithmetic

\[ c = \text{\_mm\_addsub\_ps}(a, b); \]
Arithmetic

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

\[ \begin{array}{c|cccc}
& 3.0 & 7.0 & 2.0 & 6.0 \\
\hline
\end{array} \]

\[ c = _{\text{mm}}\text{hadd}_{\text{ps}}(a, b); \]

analogous:

\[ c = _{\text{mm}}\text{hsub}_{\text{ps}}(a, b); \]

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 is even
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

```c
__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

![Diagram showing the computation process](image)

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>CMPLSS</td>
</tr>
<tr>
<td>_mm_cmple_ps</td>
<td>Less Than or Equal</td>
<td>CMPLEPS</td>
</tr>
<tr>
<td>_mm_cmgt_ss</td>
<td>Greater Than</td>
<td>CMPLTSS</td>
</tr>
<tr>
<td>_mm_cmgt_ps</td>
<td>Greater Than</td>
<td>CMPLTPS</td>
</tr>
<tr>
<td>_mm_cmgle_ss</td>
<td>Greater Than or Equal</td>
<td>CMPLSS</td>
</tr>
<tr>
<td>_mm_cmgle_ps</td>
<td>Greater Than or Equal</td>
<td>CMPLEPS</td>
</tr>
<tr>
<td>_mm_cmpgneq_ss</td>
<td>Not Equal</td>
<td>CMPNQEQSS</td>
</tr>
<tr>
<td>_mm_cmpgneq_ps</td>
<td>Not Equal</td>
<td>CMPNQEQPS</td>
</tr>
<tr>
<td>_mm_cmpltlt_ss</td>
<td>Not Less Than</td>
<td>CMPLTSS</td>
</tr>
<tr>
<td>_mm_cmpltlt_ps</td>
<td>Not Less Than</td>
<td>CMPLTPS</td>
</tr>
<tr>
<td>_mm_cmplele_ss</td>
<td>Not Less Than or Equal</td>
<td>CMPLSS</td>
</tr>
<tr>
<td>_mm_cmplele_ps</td>
<td>Not Less Than or Equal</td>
<td>CMPLEPS</td>
</tr>
<tr>
<td>_mm_cmgtgt_ss</td>
<td>Greater Than</td>
<td>CMPLTSS</td>
</tr>
<tr>
<td>_mm_cmgtgt_ps</td>
<td>Greater Than</td>
<td>CMPLTPS</td>
</tr>
<tr>
<td>_mm_cmgtneq_ss</td>
<td>Not Greater Than</td>
<td>CMPNLEQSS</td>
</tr>
<tr>
<td>_mm_cmgtneq_ps</td>
<td>Not Greater Than</td>
<td>CMPNLEQPSS</td>
</tr>
<tr>
<td>_mm_ucompteq_ss</td>
<td>Equal</td>
<td>UCMPTEQSS</td>
</tr>
<tr>
<td>_mm_ucompteq_ps</td>
<td>Equal</td>
<td>UCMPTEQPS</td>
</tr>
<tr>
<td>_mm_ucompltlt_ss</td>
<td>Not Less Than</td>
<td>UCMPLTEQSS</td>
</tr>
<tr>
<td>_mm_ucompltlt_ps</td>
<td>Not Less Than</td>
<td>UCMPTEQPS</td>
</tr>
<tr>
<td>_mm_ucomplele_ss</td>
<td>Not Less Than or Equal</td>
<td>UCMPTEQSS</td>
</tr>
<tr>
<td>_mm_ucomplele_ps</td>
<td>Not Less Than or Equal</td>
<td>UCMPTEQPS</td>
</tr>
<tr>
<td>_mm_ucomgtgt_ss</td>
<td>Greater Than</td>
<td>UCMPTEQSS</td>
</tr>
<tr>
<td>_mm_ucomgtgt_ps</td>
<td>Greater Than</td>
<td>UCMPTEQPS</td>
</tr>
<tr>
<td>_mm_ucomgtneq_ss</td>
<td>Not Greater Than</td>
<td>UCMPTEQSS</td>
</tr>
<tr>
<td>_mm_ucomgtneq_ps</td>
<td>Not Greater Than</td>
<td>UCMPTEQPS</td>
</tr>
</tbody>
</table>
Comparisons

Each field:
0xffffffff if true
0x0 if false

Return type: __m128

Example

```c
#include <xmmintrin.h>

void fcond(float *a, size_t n) {
    int i;

    __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);
        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(a+i, vr);
    }
}
```

```
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.;
    }
}
```
Vectorization

= 

Conversion

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_cvtss_si32</td>
<td>Convert to 32-bit integer</td>
<td>CVTSS2SI</td>
</tr>
<tr>
<td>_mm_cvtss_si64*</td>
<td>Convert to 64-bit integer</td>
<td>CVTSS2SI</td>
</tr>
<tr>
<td>_mm_cvtss_pi32</td>
<td>Convert to two 32-bit integers</td>
<td>CVTTPS2PI</td>
</tr>
<tr>
<td>_mm_cvtss_si32</td>
<td>Convert to 32-bit integer</td>
<td>CVTTS2SI</td>
</tr>
<tr>
<td>_mm_cvtss_si64*</td>
<td>Convert to 64-bit integer</td>
<td>CVTTS2SI</td>
</tr>
<tr>
<td>_mm_cvtss_pi32</td>
<td>Convert to two 32-bit integers</td>
<td>CVTTPS2PI</td>
</tr>
<tr>
<td>_mm_cvtss2_si32</td>
<td>Convert from 32-bit integer</td>
<td>CVTS2SS</td>
</tr>
<tr>
<td>_mm_cvtss2_si64*</td>
<td>Convert from 64-bit integer</td>
<td>CVTS2SS</td>
</tr>
<tr>
<td>_mm_cvtss2_pi32</td>
<td>Convert from two 32-bit integers</td>
<td>CVTTPS2PI</td>
</tr>
<tr>
<td>_mm_cvtss2_pi16</td>
<td>Convert from four 16-bit integers</td>
<td>composite</td>
</tr>
<tr>
<td>_mm_cvtss2_ps</td>
<td>Convert from four 16-bit integers</td>
<td>composite</td>
</tr>
<tr>
<td>_mm_cvtss2_ps</td>
<td>Convert from four 8-bit integers</td>
<td>composite</td>
</tr>
<tr>
<td>_mm_cvtss2_ps</td>
<td>Convert from four 32-bit integers</td>
<td>composite</td>
</tr>
<tr>
<td>_mm_cvtss2_ps</td>
<td>Convert to four 16-bit integers</td>
<td>composite</td>
</tr>
<tr>
<td>_mm_cvtss2_ps</td>
<td>Convert to four 8-bit integers</td>
<td>composite</td>
</tr>
<tr>
<td>_mm_cvtss2_ps</td>
<td>Extract</td>
<td>composite</td>
</tr>
</tbody>
</table>
Conversion

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

```
| 1.0 | 2.0 | 3.0 | 4.0 |
-|---|---|---|---|
| a |
```

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

Cast

```
__m128i _mm_castps_si128(__m128 a)
__m128i _mm_castsi128_ps(__m128i 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

\[
\text{\texttt{\_m128 \_mm_cvt\_pi2ps (\_m128 a, \_m64 b)}}
\]

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>SHUFP S</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 S</td>
</tr>
<tr>
<td>_mm_movehi_ps</td>
<td>Move High to Low</td>
<td>MOVH LPS</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>

**SSSE3**

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSSE3 Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_shuffle_epi8</td>
<td>Shuffle</td>
<td>PSHUFB</td>
</tr>
<tr>
<td>_mm_alignr_epi8</td>
<td>Shift</td>
<td>PALIGNR</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

\[
\begin{align*}
\text{c} &= \_\text{mm}_\text{unpacklo}_\text{ps}(a, b); \\
\text{c} &= \_\text{mm}_\text{unpackhi}_\text{ps}(a, b);
\end{align*}
\]

\[
\begin{align*}
c &= \_\text{mm}_\text{shuffle}_\text{ps}(a, b, _\text{MM}_\text{SHUFFLE}(l, k, j, i));
\end{align*}
\]

c0 = ai
c1 = aj
c2 = bk
c3 = bl
i,j,k,l \ in \ \{0,1,2,3\}
Example: Loading 4 Real Numbers from Arbitrary Memory Locations

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 \texttt{\_mm\_load\_ps()}
- Other possibility (but likely also yields 7 instructions)

\begin{verbatim}
__m128 \texttt{vf};
\texttt{vf} = \texttt{\_mm\_set\_ps(*p3, *p2, *p1, *p0)};
\end{verbatim}

- \texttt{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?):

\begin{verbatim}
__declspec(align(16)) float \texttt{g[4];}
__m128 \texttt{vf};
\texttt{g[0]} = \texttt{*p0;}
\texttt{g[1]} = \texttt{*p1;}
\texttt{g[2]} = \texttt{*p2;}
\texttt{g[3]} = \texttt{*p3;}
\texttt{vf} = \texttt{\_mm\_load\_ps(g)};
\end{verbatim}
Example: Storing 4 Real Numbers to Arbitrary Memory Locations

7 instructions, shorter critical path

Shuffle

\_m128i \_mm\_alignr\_epi8(\_m128i \textit{a}, \_m128i \textit{b}, \textbf{const} \textit{int} \textit{n})

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

\textit{Example}: View \_m128i as 4 32-bit ints; \textit{n} = 12

How to use this with floating point vectors?

\textit{Use with \_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+4+i));  // 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_castps_si128(_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
}

void shift(float *x, float *y, int n) {
    for (int i = 0; i < n-1; i++)
        y[i] = x[i+1];
    y[n-1] = 0;
}
```

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

Use with _mm_castsi128_ps to do the same for floating point
Shuffle

\_\_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: LSB 0x0 0x0 0x0 0x0 mask

LSB 1.0 2.0 3.0 4.0 a  

LSB 0.5 1.5 2.5 3.5 b

LSB 1.0 1.5 3.0 4.0 c

see also \_\_mm\_blend\_ps

Example (Continued From Before)

```c
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.;
    }
}

#include <xmmintrin.h>

void fcond(float *a, size_t n) {
    int i;
    \_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(ones, mones, vmask);
        vr = \_mm\_add\_ps(vt, vb);
    }
}
```
Shuffle

_macro_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>row0</th>
<th>LSB</th>
<th>row1</th>
<th>LSB</th>
<th>row2</th>
<th>LSB</th>
<th>row3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2.0</td>
<td>3.0</td>
<td>4.0</td>
<td>1.0</td>
<td>5.0</td>
<td>9.0</td>
<td>13.0</td>
</tr>
<tr>
<td>5.0</td>
<td>6.0</td>
<td>7.0</td>
<td>8.0</td>
<td>2.0</td>
<td>6.0</td>
<td>10.0</td>
<td>14.0</td>
</tr>
<tr>
<td>9.0</td>
<td>10.0</td>
<td>11.0</td>
<td>12.0</td>
<td>3.0</td>
<td>7.0</td>
<td>11.0</td>
<td>15.0</td>
</tr>
<tr>
<td>13.0</td>
<td>14.0</td>
<td>15.0</td>
<td>16.0</td>
<td>4.0</td>
<td>8.0</td>
<td>12.0</td>
<td>16.0</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!](https://eth-ai.ch/))
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];
}
```

```asm
    ;;;    a[i] = a[i] + b[i];
    movss xmm0, DWORD PTR [rcx+rax*4]
    addss xmm0, DWORD PTR [rdx+rax*4]
    movss DWORD PTR [rcx+rax*4], xmm0
```

**vectorized:**

```asm
    ;;;    a[i] = a[i] + b[i];
    movss xmm0, DWORD PTR [rcx+r11*4]
    addss xmm0, DWORD PTR [rdx+r11*4]
    movss DWORD PTR [rcx+r11*4], xmm0
    ...
    movups xmm0, XMMWORD PTR [rdx+r10*4]
    movups xmm1, XMMWORD PTR [16+rdx+r10*4]
    addps xmm0, XMMWORD PTR [rcx+r10*4]
    addps xmm1, XMMWORD PTR [16+rcx+r10*4]
    movaps XMMWORD PTR [rcx+r10*4], xmm0
    movaps XMMWORD PTR [16+rcx+r10*4], xmm1
```

why this?

why everything twice?

why movups and movaps?

unaligned

aligned
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:

```
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**

  ```
  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`)**

  ```
  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;
```

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.
- ...

```c
void myadd(float *a, float *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?

**Yes: Through versioning**

```
function

runtime check
a, b potentially aliased?

no

runtime check
a, b aligned?

yes

yes, yes

vectorized loop
aligned loads

yes, no

vectorized loop
aligned and unaligned loads
or peeling and aligned loads

no, yes

vectorized loop
unaligned loads
or peeling and aligned loads

no, no

unvectorized loop
```
Compiler Vectorization

- Read manual