How to Write Fast Numerical Code
Spring 2015

Lecture: SIMD extensions, SSE, compiler vectorization

Instructor: Markus Püschel
TA: Gagandeep Singh, Daniele Spampinato, Alen Stojanov

Planning

- Currently: work on project (submit update 2.4.)
- No class on Sechseläuten (13.4.)
- Exam 15.4.
- First one-on-one meetings: late April
Flynn’s Taxonomy

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

SIMD Extensions and SSE

- Overview: SSE family
- 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**: Relative easy to design; chip designers have enough transistors to play with

---

**MMX:** Multimedia extension

**SSE:** Streaming SIMD extension

**AVX:** Advanced vector extensions

<table>
<thead>
<tr>
<th>Intel x86</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86-16</td>
<td>8086</td>
</tr>
<tr>
<td></td>
<td>286</td>
</tr>
<tr>
<td>x86-32</td>
<td>386</td>
</tr>
<tr>
<td>MMX</td>
<td>486</td>
</tr>
<tr>
<td>SSE</td>
<td>Pentium</td>
</tr>
<tr>
<td>SSE2</td>
<td>Pentium MMX</td>
</tr>
<tr>
<td>SSE3</td>
<td>Pentium III</td>
</tr>
<tr>
<td>x86-64 / em64t</td>
<td>Pentium 4</td>
</tr>
<tr>
<td>SSE4</td>
<td>Pentium 4E</td>
</tr>
<tr>
<td>AVX</td>
<td>Core 2 Duo</td>
</tr>
<tr>
<td>AVX2</td>
<td>Penryn</td>
</tr>
<tr>
<td></td>
<td>Core i7 (Nehalem)</td>
</tr>
<tr>
<td></td>
<td>Core i7</td>
</tr>
<tr>
<td></td>
<td>Sandy Bridge</td>
</tr>
<tr>
<td></td>
<td>Haswell</td>
</tr>
</tbody>
</table>
Overview Floating-Point Vector ISAs

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Name</th>
<th>v-way</th>
<th>Precision</th>
<th>Introduced with</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>SSE</td>
<td>4-way</td>
<td>single</td>
<td>Pentium III</td>
</tr>
<tr>
<td></td>
<td>SSE2</td>
<td>+</td>
<td>double</td>
<td>Pentium 4</td>
</tr>
<tr>
<td></td>
<td>SSE3</td>
<td></td>
<td></td>
<td>Pentium 4 (Prescott)</td>
</tr>
<tr>
<td></td>
<td>SSSE3</td>
<td></td>
<td></td>
<td>Core Duo</td>
</tr>
<tr>
<td></td>
<td>SSE4</td>
<td>8-way</td>
<td>single</td>
<td>Core2 Extreme (Penryn)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4-way</td>
<td>double</td>
<td>Core i7 (Sandybridge)</td>
</tr>
<tr>
<td>Intel</td>
<td>IPX</td>
<td>2-way</td>
<td>single</td>
<td>Itanium</td>
</tr>
<tr>
<td>Intel</td>
<td>LRB</td>
<td>16-way</td>
<td>single</td>
<td>Larrabee</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8-way</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>AMD</td>
<td>3DNow!</td>
<td>2-way</td>
<td>single</td>
<td>K6</td>
</tr>
<tr>
<td></td>
<td>Enhanced 3DNow!</td>
<td></td>
<td></td>
<td>K7</td>
</tr>
<tr>
<td></td>
<td>3DNow! Professional</td>
<td>+</td>
<td>single</td>
<td>Athlon XP</td>
</tr>
<tr>
<td></td>
<td>AMD64</td>
<td></td>
<td>double</td>
<td>Opteron</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>4-way</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>Motorola</td>
<td>Altivec</td>
<td>4-way</td>
<td>single</td>
<td>MPC 7400 G4</td>
</tr>
<tr>
<td>IBM</td>
<td>VMX</td>
<td>4-way</td>
<td>single</td>
<td>PowerPC 970 G5</td>
</tr>
<tr>
<td></td>
<td>SPU</td>
<td>+</td>
<td>double</td>
<td>Cell BE</td>
</tr>
<tr>
<td>IBM</td>
<td>Double FPU</td>
<td>2-way</td>
<td>double</td>
<td>PowerPC 440 FP2</td>
</tr>
</tbody>
</table>

Within an extension family, newer generations add features to older ones
Convergence: 3DNow! Professional = 3DNow! + SSE; VMX = Altivec;
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

- 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:** \texttt{addps} \%xmm0 \%xmm1

  ![Diagram of 4-way vector add]

- **Single precision scalar add:** \texttt{addss} \%xmm0 \%xmm1

  ![Diagram of scalar add]

### SSE3 Instruction Names

- **addps**
  - packed (vector)
  - single precision

- **addpd**
  - double precision

- **addss**
  - single slot (scalar)

- **addsd**

  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

```
ipf:
  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
  movaps %xmm1, %xmm0
  ret
```

```
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;
}
```

From Core 2 Manual

Latency, throughput

<table>
<thead>
<tr>
<th></th>
<th>Single-precision (SP) FP MUL</th>
<th>Double-precision FP MUL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.1</td>
<td>4.1</td>
</tr>
<tr>
<td></td>
<td>5.1</td>
<td>5.1</td>
</tr>
<tr>
<td>SSE based FP x87 FP</td>
<td>5.2</td>
<td>5.2</td>
</tr>
<tr>
<td></td>
<td>1.1</td>
<td>1.1</td>
</tr>
</tbody>
</table>

SSE based FP x87 FP

FP shuffle
DIV/SQRT

Issue port 0; Whiteport port 0
FP shuffle does not handle QW shuffle.
Summary

- On Core 2 there are two different (unvectorized) floating points
  - x87: obsolete, is default on x86-32
  - SSE based: uses only one slot, is default on x86-64

- SIMD vector floating point instructions
  - 4-way single precision: since SSE
  - 2-way double precision: since SSE2
  - SSE vector add and mult are fully pipelined (1 per cycle): possible gain 4x and 2x, respectively
  - Starting with Sandybridge, AVX was introduced: 8-way single, 4-way double

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 (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*
SSE Family Intrinsics

- Assembly coded C functions
- Expanded inline upon compilation: no overhead
- Like writing assembly inside C
- Floating point:
  - Intrinsics for math functions: log, sin, ...
  - Intrinsics for SSE

- Our introduction is based onicc
  - Most intrinsics work with gcc and Visual Studio (VS)
  - Some language extensions are icc (or even VS) specific

Header files

- SSE: xmmintrin.h
- SSE2: emmintrin.h
- SSE3: pmmintrin.h
- SSSE3: tmmintrin.h
- SSE4: smmintrin.h and nmmintrin.h

or ia32intrin.h
Visual Conventions We Will Use

- **Memory**
  - Increasing address

- **Registers**
  - Before (and common)
  - Now we will use

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`
SSE Intrinsics (Focus Floating Point)

- **Instructions**
  - Naming convention: `_mm_<intrin_op><suffix>`
  - Example:
    ```c
    // 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
    ```c
    __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()
  `- mm_add_ps()
  `- mm_mul_ps()
  `- ...

- **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 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>MOVS + 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>

```c
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
```
How to Align

- __m128, __m128d, __m128i are 16-byte aligned

- Arrays:
  
  ```
  __declspec(align(16)) float g[4];
  ```

- Dynamic allocation
  - __mm_malloc() and __mm_free()
  - Write your own malloc that returns 16-byte aligned addresses
  - Some malloc's already guarantee 16-byte alignment

 Loads and Stores

```
  a = __mm_loadl_pi(a, p); // p 8-byte aligned
  a = __mm_loadh_pi(a, p); // p 8-byte aligned
```
Loads and Stores

\[
\begin{array}{c}
\text{p} \\
\downarrow \\
\text{memory}
\end{array}
\quad
\begin{array}{c}
1.0 \\
\downarrow \\
\text{a}
\end{array}
\]

<table>
<thead>
<tr>
<th>LSB</th>
<th>1.0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>__</td>
<td></td>
</tr>
<tr>
<td>a set to zero</td>
<td></td>
</tr>
</tbody>
</table>

\[a = \_mm\_load\_ss(p); // p any alignment\]

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>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( _mm_set_ps(4.0, 3.0, 2.0, 1.0) );</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LSB</th>
<th>1.0</th>
<th>1.0</th>
<th>1.0</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b )</td>
<td>( _mm_set1_ps(1.0) );</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LSB</th>
<th>1.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c )</td>
<td>( _mm_set_ss(1.0) );</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LSB</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d )</td>
<td>( _mm_setzero_ps() );</td>
<td></td>
<td></td>
<td></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_sqr_t _ss</td>
<td>Squared Root</td>
<td>SQRTSS</td>
</tr>
<tr>
<td>_mm_sqr_t _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_rsq_t _ss</td>
<td>Reciprocal Squared Root</td>
<td>RSQRTPS</td>
</tr>
<tr>
<td>_mm_rsq_t _ps</td>
<td>Reciprocal Squared Root</td>
<td>RSQRTPS</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

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

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

**analogous:**

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

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

Example

```c
#include <immintrin.h>

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;
}

#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);
        x_vec = _mm_add_ps(x_vec, ind);
        ind = _mm_add_ps(ind, incr);
        _mm_store_ps(x+i, x_vec);
    }
}
```

How does the code style differ from scalar code?

*Intrinsics force scalar replacement!*

Arithmetic

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

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

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

Arithmetic

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

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

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

\[ c = \text{mm\_hadd\_ps}(a, b); \]

\textit{analogous:}

\[ c = \text{mm\_hsub\_ps}(a, b); \]

Example

```c
// 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

![Diagram showing the computation of mm_dp_ps using SSE4](image)

### Comparisons

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_cmpneq_ss</td>
<td>Not Equal</td>
<td>CMPNEQSS</td>
</tr>
<tr>
<td>_mm_cmpneq_ps</td>
<td>Not Equal</td>
<td>CMPNEQPS</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_clmsle_ss</td>
<td>Less Than or Equal</td>
<td>CMPLESS</td>
</tr>
<tr>
<td>_mm_clmsle_ps</td>
<td>Less Than or Equal</td>
<td>CMPLEPS</td>
</tr>
<tr>
<td>_mm_cmgte_ss</td>
<td>Greater Than or Equal</td>
<td>CMPGTSS</td>
</tr>
<tr>
<td>_mm_cmgte_ps</td>
<td>Greater Than or Equal</td>
<td>CMPGTPE</td>
</tr>
<tr>
<td>_mm_cmneq_ss</td>
<td>Not Equal</td>
<td>CMPNEQSS</td>
</tr>
<tr>
<td>_mm_cmneq_ps</td>
<td>Not Equal</td>
<td>CMPNEQPS</td>
</tr>
<tr>
<td>_mm_cmpltlt_ss</td>
<td>Not Less Than</td>
<td>CMPNLTSS</td>
</tr>
<tr>
<td>_mm_cmpltlt_ps</td>
<td>Not Less Than</td>
<td>CMPNLTPS</td>
</tr>
<tr>
<td>_mm_cmpltle_ss</td>
<td>Not Less Than or Equal</td>
<td>CMPPLTSS</td>
</tr>
<tr>
<td>_mm_cmpltle_ps</td>
<td>Not Less Than or Equal</td>
<td>CMPPLTPE</td>
</tr>
<tr>
<td>_mm_cmpltge_ss</td>
<td>Greater Than or Equal</td>
<td>CMPGTLESS</td>
</tr>
<tr>
<td>_mm_cmpltge_ps</td>
<td>Greater Than or Equal</td>
<td>CMPGTLEPS</td>
</tr>
<tr>
<td>_mm_cmpltneq_ss</td>
<td>Not Greater Than or Equal</td>
<td>CMPNLEQSS</td>
</tr>
<tr>
<td>_mm_cmpltneq_ps</td>
<td>Not Greater Than or Equal</td>
<td>CMPNLEQPS</td>
</tr>
<tr>
<td>_mm_cmpltltlt_ss</td>
<td>Not Less Than Less Than</td>
<td>CMPNLTLSS</td>
</tr>
<tr>
<td>_mm_cmpltltlt_ps</td>
<td>Not Less Than Less Than</td>
<td>CMPNLTLPS</td>
</tr>
<tr>
<td>_mm_cmpltlteq_ss</td>
<td>Not Less Than or Equal Less Than</td>
<td>CMPNLTLEQSS</td>
</tr>
<tr>
<td>_mm_cmpltlteq_ps</td>
<td>Not Less Than or Equal Less Than</td>
<td>CMPNLTLEQPS</td>
</tr>
<tr>
<td>_mm_cmpltlneq_ss</td>
<td>Not Greater Than or Equal Less Than</td>
<td>CMPNLTLNEQSS</td>
</tr>
<tr>
<td>_mm_cmpltlneq_ps</td>
<td>Not Greater Than or Equal Less Than</td>
<td>CMPNLTLNEQPS</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_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>CMPLESS</td>
</tr>
<tr>
<td>_mm_cmple_ps</td>
<td>Less Than or Equal</td>
<td>CMPLEPS</td>
</tr>
<tr>
<td>_mm_cmgte_ss</td>
<td>Greater Than</td>
<td>CMPGTSS</td>
</tr>
<tr>
<td>_mm_cmgte_ps</td>
<td>Greater Than</td>
<td>CMPGTPE</td>
</tr>
<tr>
<td>_mm_cmneq_ss</td>
<td>Not Equal</td>
<td>CMPNEQSS</td>
</tr>
<tr>
<td>_mm_cmneq_ps</td>
<td>Not Equal</td>
<td>CMPNEQPS</td>
</tr>
<tr>
<td>_mm_cmpltlt_ss</td>
<td>Not Less Than</td>
<td>CMPNLTSS</td>
</tr>
<tr>
<td>_mm_cmpltlt_ps</td>
<td>Not Less Than</td>
<td>CMPNLTPS</td>
</tr>
<tr>
<td>_mm_cmpltle_ss</td>
<td>Not Less Than or Equal</td>
<td>CMPPLTSS</td>
</tr>
<tr>
<td>_mm_cmpltle_ps</td>
<td>Not Less Than or Equal</td>
<td>CMPPLTPE</td>
</tr>
<tr>
<td>_mm_cmpltge_ss</td>
<td>Greater Than</td>
<td>CMPGTSS</td>
</tr>
<tr>
<td>_mm_cmpltge_ps</td>
<td>Greater Than</td>
<td>CMPGTPE</td>
</tr>
<tr>
<td>_mm_cmpltneq_ss</td>
<td>Not Greater Than</td>
<td>CMPNLEQSS</td>
</tr>
<tr>
<td>_mm_cmpltneq_ps</td>
<td>Not Greater Than</td>
<td>CMPNLEQPS</td>
</tr>
<tr>
<td>_mm_cmpltltlt_ss</td>
<td>Not Less Than Less Than</td>
<td>CMPNLTLSS</td>
</tr>
<tr>
<td>_mm_cmpltltlt_ps</td>
<td>Not Less Than Less Than</td>
<td>CMPNLTLPS</td>
</tr>
<tr>
<td>_mm_cmpltlteq_ss</td>
<td>Not Less Than or Equal Less Than</td>
<td>CMPNLTLEQSS</td>
</tr>
<tr>
<td>_mm_cmpltlteq_ps</td>
<td>Not Less Than or Equal Less Than</td>
<td>CMPNLTLEQPS</td>
</tr>
<tr>
<td>_mm_cmpltlneq_ss</td>
<td>Not Greater Than or Equal Less Than</td>
<td>CMPNLTLNEQSS</td>
</tr>
<tr>
<td>_mm_cmpltlneq_ps</td>
<td>Not Greater Than or Equal Less Than</td>
<td>CMPNLTLNEQPS</td>
</tr>
</tbody>
</table>
Comparisons

<table>
<thead>
<tr>
<th>LSB</th>
<th>1.0</th>
<th>2.0</th>
<th>3.0</th>
<th>4.0</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>≥</td>
<td>≥</td>
<td>≥</td>
<td>≥</td>
<td>b</td>
</tr>
<tr>
<td>LSB</td>
<td></td>
<td>0x0</td>
<td></td>
<td>0x0</td>
<td>0x0</td>
</tr>
</tbody>
</table>

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

Return type: __m128

```c

// Comparisons

c = _mm_cmpeq_ps(a, b);

// analogous:

c = _mm_cmple_ps(a, b);
c = _mm_cmpgt_ps(a, b);
c = _mm_cmpge_ps(a, b);
c = _mm_cmplt_ps(a, b);
c = _mm_cmpneq_ps(a, b);

etc.
```

Example

```c

// Example

#include <xmmintrin.h>

void fcond(float *x, size_t n) {
    int i;
    __m128 vt, vr, vtp1, vtm1, vmask, ones, thresholds;
    ones = _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);
        vtp1 = _mm_add_ps(vt, ones);
        vtm1 = _mm_sub_ps(vt, ones);
        vr = _mm_or_ps(_mm_and_ps(vmask, vtp1), _mm_andnot_ps(vmask, vtm1));
        _mm_store_ps(a+i, vr);
    }
}
```

void fcond(float *a, size_t n) {
    int i;
    __m128 vt, vr, vtp1, vtm1, vmask, ones, thresholds;
    ones = _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);
        vtp1 = _mm_add_ps(vt, ones);
        vtm1 = _mm_sub_ps(vt, ones);
        vr = _mm_or_ps(_mm_and_ps(vmask, vtp1), _mm_andnot_ps(vmask, vtm1));
        _mm_store_ps(a+i, vr);
    }
}
Vectorization

Picture: www.druckundbestell.de

Conversion

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_cvtss_s_32</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_cvtps_pi32</td>
<td>Convert to two 32-bit integers</td>
<td>CVTPS2PI</td>
</tr>
<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_cvtps_pi32</td>
<td>Convert to two 32-bit integers</td>
<td>CVTPS2PI</td>
</tr>
<tr>
<td>_mm_cvtsi32_ss</td>
<td>Convert from 32-bit integer</td>
<td>CVTS2SS</td>
</tr>
<tr>
<td>_mm_cvtsi64_ss*</td>
<td>Convert from 64-bit integer</td>
<td>CVTS2SS</td>
</tr>
<tr>
<td>_mm_cvtpi32_ps</td>
<td>Convert from two 32-bit integers</td>
<td>CVTTPS2PS</td>
</tr>
<tr>
<td>_mm_cvtpi6_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_cvtpu8_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_cvtpi32x2_ps</td>
<td>Convert from four 32-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_cvtss_f32</td>
<td>Extract</td>
<td>composite</td>
</tr>
</tbody>
</table>
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>f</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Cast

```c
__m128i _mm_castsi128_ps(__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._

```c
__m128i _mm_castsi128_ps(__m128i a)
```

Makes integer shuffle instructions usable for floating point.
### 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_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_movelupd_ps</td>
<td>Duplicates</td>
<td>MOVSLDUP</td>
</tr>
</tbody>
</table>

**SSE4**

**SSSE3**

<table>
<thead>
<tr>
<th>Intrinsic Syntax</th>
<th>Operation</th>
<th>Corresponding SSSE3 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>

### Examples

- **Example 1**
  
  ```
  c = _mm_unpacklo_ps(a, b);
  ```

- **Example 2**
  
  ```
  c = _mm_unpackhi_ps(a, b);
  ```

- **Example 3**
  
  ```
  c = _mm_shuffle_ps(a, b, c);
  ```
Shuffles

\[ c = \text{_mm_shuffle_ps}(a, b, \_MM_SHUFFLE(l, k, j, i)); \]

helper macro to create mask

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

\[
\begin{array}{cccc}
c0 & c1 & c2 & c3 \\
\text{LSB} & & & c \\
\end{array}
\]

any element of \( a \)

any element of \( b \)

\[
c0 = ai \\
c1 = aj \\
c2 = bk \\
c3 = bl \\
\]

\[
i,j,k,l \text{ in } \{0,1,2,3\}
\]

Example: Loading 4 Real Numbers from Arbitrary Memory Locations

\[
\text{memory}
\]

\[
\begin{array}{cccc}
p0 & p1 & p2 & p3 \\
1.0 & 2.0 & 3.0 & 4.0 \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{LSB} & 1.0 & 0 & 0 & 0 \\
\text{LSB} & 2.0 & 0 & 0 & 0 \\
\text{LSB} & 3.0 & 0 & 0 & 0 \\
\text{LSB} & 4.0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{LSB} & 1.0 & 2.0 & 0 & 0 \\
\text{LSB} & 3.0 & 0 & 4.0 & 0 \\
\text{LSB} & 1.0 & 2.0 & 3.0 & 4.0 \\
\end{array}
\]

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)
  ```c
  __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?):

```c
__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

```
1.0 2.0 3.0 4.0
```

7 instructions, shorter critical path
**Shuffle**

\[
\text{__m128i \_mm_alignr_epi8(__m128i a, __m128i b, const int n)}
\]

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; \(n = 12\)

![Diagram showing byte alignment and extraction](image)

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(__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

__m128i __m_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

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
```

Use with __m_castsi128_ps to do the same for floating point

Shuffle

__m128 __m_blendsv_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:  

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2.0</td>
<td>3.0</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>1.5</td>
<td>3.0</td>
<td>4.0</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>1.5</td>
<td>2.5</td>
<td>3.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>1.5</td>
<td>3.0</td>
<td>4.0</td>
</tr>
</tbody>
</table>
```

see also __m_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] += 1.0;
        else
            x[i] -= 1.0;
    }
}
```

```c
#include <smmintrin.h>

void fcond(float *a, size_t n) {
    int i;
    __m128 vt, vr, vtp1, vtm1, vmask, ones, thresholds;
    ones = _mm_set1_ps(1.0);
    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);
        vtp1 = _mm_add_ps(vt, ones);
        vtm1 = _mm_sub_ps(vt, ones);
        vr = _mm_blendv_ps(vtm1, vtp1, vmask);  
        _mm_store_ps(a+i, vr);
    }
}
```

Shuffle

```
_MM_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.

```
LSB  1.0  2.0  3.0  4.0  row\theta  
LSB  5.0  6.0  7.0  8.0  row1  
LSB  9.0 10.0 11.0 12.0  row2  
LSB 13.0 14.0 15.0 16.0  row3  
```

**In SSE**: 8 shuffles (4 _mm_unpacklo_ps, 4 _mm_unpackhi_ps)
Vectorization With Intrinsics: Key Points

- Use aligned loads and stores
- Minimize overhead (shuffle instructions) = maximize vectorization efficiency

Definition: Vectorization efficiency

\[
\begin{array}{c|c}
\text{Op count of scalar (unvectorized) code} & \text{Op count of vectorized code} \\
\end{array}
\]


Ideally: Efficiency = v for v-way vector instructions

- assumes no vector instruction does more than v scalar ops
- assumes every vector instruction has the same cost (not true: see hadd for example)

Vectorization Efficiency: Example I

```c
#include <ia32intrin.h>

// n is even
void lp(float *x, float *y, int n) {
    for (int i = 0; i < n/8; 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
    }
}
```
Vectorization Efficiency: Example 2

- 4 x 4 matrix-vector multiplication
- **Blackboard**

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

SIMD Extensions and SSE

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

References:
- Intel icc manual (look for auto vectorization)
Compiler Vectorization

- Compiler flags
- Aliasing
- Proper code style
- Alignment

Compiler Flags (icc 12.0)

<table>
<thead>
<tr>
<th>Linux* OS and Mac OS* X</th>
<th>Windows* OS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-vec</td>
<td>/Qvec</td>
<td>Enables or disables vectorization and transformations enabled for vectorization. Vectorization is enabled by default. To disable, use -no-vec (Linux* and MacOS* X) or /Qvec- (Windows*) option. Supported on IA-32 and Intel® 64 architectures only.</td>
</tr>
<tr>
<td>-no-vec</td>
<td>/Qvec</td>
<td></td>
</tr>
<tr>
<td>-vec-report</td>
<td>/Qvec-report</td>
<td>Controls the diagnostic messages from the vectorizer. See Vectorization Report.</td>
</tr>
<tr>
<td>-simd</td>
<td>/Qsimd</td>
<td>Controls user-mandated (SIMD) vectorization. User-mandated (SIMD) vectorization is enabled by default. Use the -no-simd (Linux* or MacOS* X) or /Qsimd- (Windows*) option to disable SIMD transformations for vectorization.</td>
</tr>
<tr>
<td>-no-simd</td>
<td>/Qsimd</td>
<td></td>
</tr>
</tbody>
</table>

Architecture flags:

Linux: -xHost  -mHost
Windows: /QxHost  /Qarch:Host

Host in {SSE2, SSE3, SSSE3, SSE4.1, SSE4.2}

Default: -mSSE2, /Qarch:SSE2
How Do I Know the Compiler Vectorized?

- `vec-report (previous slide)`
- 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];
}
```

```assembly
;;;    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:**

```assembly
;;;    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
```

- why this?
- why everything twice?
- why movups and movaps?

```assembly
... 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
```

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`, `/Oa`
  - `-fargument-noalias`, `/Qalias-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

- There should not be any backward loop-carried dependencies
  - Need to allow consecutive iterations of original loop to be executed together within new unrolled, vectorized loop.

- Vectorizable or not?

```c
#pragma ivdep
for (i=1; i<n; i++) {
    a[i] = 2*b[i];
    d[i] = c[i] = a[i-1];
}
```

```c
#pragma ivdep
for (i=1; i<n; i++) {
    d[i] = c[i] + a[i-1];
    a[i] = 2*b[i];
}
```
Proper Code Style

- No special operators and no function calls, unless inlined, either manually or automatically by the compiler, or they are SIMD (vectorized) functions.

```c
#pragma ivdep
for (i=1; i<n; i++) {
    a[i] = foo(b[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);
    ```
Assume:
- No aliasing information
- No alignment information

Can compiler vectorize?

Yes: Through versioning

Compiler Vectorization

- Read manual