SIMD Instructions

18-613: Foundations of Computer Systems
4th Lecture, Feb 19, 2019

Instructor:
Franz Franchetti

Based on ETH 263-2800-00L:
“Design of Parallel and High-Performance Computing” by Markus Püschel:
SIMD Vector Instructions in a Nutshell

■ What are these instructions?
  ▪ Extension of the ISA. Data types and instructions for parallel computation on short (2-16) vectors of integers and floats

  ![4-way SIMD example]

■ Why are they here?
  ▪ Useful: Many applications (e.g., multimedia) feature the required fine grain parallelism – code potentially faster
  ▪ Doable: Chip designers have enough transistors available, easy to implement

**SIMD** = *Single Instruction Multiple Data*
Other SIMD/Vector Hardware

- Original SIMD machines (CM-2,...)
  - Don’t really have anything in common with SIMD vector extension

- Vector Computers (NEC SX6, Earth simulator, SX-Aurora)
  - Vector lengths of up to 128
  - High bandwidth memory, no memory hierarchy
  - Pipelined vector operations

- Very long instruction word (VLIW) architectures
  - Explicit parallelism
  - More flexible
  - No data reorganization necessary

- SIMT (Nvidia)
  - SIMD lane == tread
  - Predicated execution, scheduling,...
Evolution of Intel Vector Instructions

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

### Intel x86

- **x86-16**
  - 8086
- **x86-32**
  - 286
  - 386
  - 486
  - Pentium
  - Pentium MMX
- **x86-64**
  - Pentium III
  - Pentium 4
  - Pentium 4E
  - Pentium 4F
  - Core 2 Duo
  - Penryn
  - Core i7 (Nehalem)
  - Sandy Bridge
  - Haswell
  - Skylake-X

**Register width**
- 64 bit (only int)
- 128 bit
- 256 bit
- 512 bit

**Time**
Intel SSE/AVX: Floating Point

- **SSE**
  - 128-bit
  - 4x float

- **SSE2**
  - 128-bit
  - 4x float

- **SSE3**
  - 128-bit
  - 4x float

- **SSSE3**
  - 128-bit
  - 2x double

- **SSE4.1/4.2**
  - 256-bit
  - 8x float
  - 4x double

- **AVX**
  - 512-bit
  - 16x float
  - 8x double

- **AVX2, FMA3**
  - 256-bit
  - 8x float
  - 4x double

- **AVX512, KNC**
  - 512-bit
  - 16x float
  - 8x double
Intel MMX/SSE/AVX: Integer

- MMX: 64-bit
- SSE: 8/16/32/64 bit int
- SSE2: 128 bit
- SSE3: 8/16/32/64/128 bit int
- SSSE3: 128 bit
- SSE4.1/4.2: 256 bit
- AVX: 8/16/32/64/128 bit int
- AVX2, FMA3: 512 bit
- AVX512, KNC: 16x float, 8x double
- MMX: 64-bit
- SSE: 8/16/32/64 bit int
- SSE2: 128 bit
- SSE3: 8/16/32/64/128 bit int
- SSSE3: 128 bit
- SSE4.1/4.2: 256 bit
- AVX: 8/16/32/64/128 bit int
- AVX2, FMA3: 512 bit
- AVX512, KNC: 16x float, 8x double
- MMX: 64-bit
- SSE: 8/16/32/64 bit int
- SSE2: 128 bit
- SSE3: 8/16/32/64/128 bit int
- SSSE3: 128 bit
- SSE4.1/4.2: 256 bit
- AVX: 8/16/32/64/128 bit int
- AVX2, FMA3: 512 bit
- AVX512, KNC: 16x float, 8x double
- MMX: 64-bit
- SSE: 8/16/32/64 bit int
- SSE2: 128 bit
- SSE3: 8/16/32/64/128 bit int
- SSSE3: 128 bit
- SSE4.1/4.2: 256 bit
- AVX: 8/16/32/64/128 bit int
- AVX2, FMA3: 512 bit
- AVX512, KNC: 16x float, 8x double
- MMX: 64-bit
- SSE: 8/16/32/64 bit int
- SSE2: 128 bit
- SSE3: 8/16/32/64/128 bit int
- SSSE3: 128 bit
- SSE4.1/4.2: 256 bit
- AVX: 8/16/32/64/128 bit int
- AVX2, FMA3: 512 bit
- AVX512, KNC: 16x float, 8x double
- MMX: 64-bit
- SSE: 8/16/32/64 bit int
- SSE2: 128 bit
- SSE3: 8/16/32/64/128 bit int
- SSSE3: 128 bit
- SSE4.1/4.2: 256 bit
- AVX: 8/16/32/64/128 bit int
- AVX2, FMA3: 512 bit
- AVX512, KNC: 16x float, 8x double
- MMX: 64-bit
- SSE: 8/16/32/64 bit int
- SSE2: 128 bit
- SSE3: 8/16/32/64/128 bit int
- SSSE3: 128 bit
- SSE4.1/4.2: 256 bit
- AVX: 8/16/32/64/128 bit int
- AVX2, FMA3: 512 bit
- AVX512, KNC: 16x float, 8x double
- MMX: 64-bit
- SSE: 8/16/32/64 bit int
- SSE2: 128 bit
- SSE3: 8/16/32/64/128 bit int
- SSSE3: 128 bit
- SSE4.1/4.2: 256 bit
- AVX: 8/16/32/64/128 bit int
- AVX2, FMA3: 512 bit
- AVX512, KNC: 16x float, 8x double
- MMX: 64-bit
- SSE: 8/16/32/64 bit int
- SSE2: 128 bit
- SSE3: 8/16/32/64/128 bit int
- SSSE3: 128 bit
- SSE4.1/4.2: 256 bit
- AVX: 8/16/32/64/128 bit int
- AVX2, FMA3: 512 bit
- AVX512, KNC: 16x float, 8x double
- MMX: 64-bit
- SSE: 8/16/32/64 bit int
- SSE2: 128 bit
- SSE3: 8/16/32/64/128 bit int
- SSSE3: 128 bit
- SSE4.1/4.2: 256 bit
- AVX: 8/16/32/64/128 bit int
- AVX2, FMA3: 512 bit
- AVX512, KNC: 16x float, 8x double
- MMX: 64-bit
Other Types of Special Instructions

- Cryptography
- Bit-manipulation
- Cache/TLB control
- Debug support
- bounds checks
- Hardware random number generator
- String operations
- Half precision (FP16) support
- Machine control register access
- Galois field arithmetic
### SSE3/4/AVX128 XMM Register File

**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>

Since SSE2 also vectors of 8/16/32/64/128-bit integers
AVX Register File (Sandy Bridge and Later)

256 bit = 4 doubles = 8 singles

Since AVX2 also vectors of 8/16/32/64/128/256-bit integers
### AVX512 Register File

512 bit = 16 doubles = 32 singles

<table>
<thead>
<tr>
<th>%zmm0</th>
<th>%zmm8</th>
<th>%zmm16</th>
<th>%zmm24</th>
</tr>
</thead>
<tbody>
<tr>
<td>%zmm1</td>
<td>%zmm9</td>
<td>%zmm17</td>
<td>%zmm25</td>
</tr>
<tr>
<td>%zmm2</td>
<td>%zmm10</td>
<td>%zmm18</td>
<td>%zmm26</td>
</tr>
<tr>
<td>%zmm3</td>
<td>%zmm11</td>
<td>%zmm19</td>
<td>%zmm27</td>
</tr>
<tr>
<td>%zmm4</td>
<td>%zmm12</td>
<td>%zmm20</td>
<td>%zmm28</td>
</tr>
<tr>
<td>%zmm5</td>
<td>%zmm13</td>
<td>%zmm21</td>
<td>%zmm29</td>
</tr>
<tr>
<td>%zmm6</td>
<td>%zmm14</td>
<td>%zmm22</td>
<td>%zmm30</td>
</tr>
<tr>
<td>%zmm7</td>
<td>%zmm15</td>
<td>%zmm23</td>
<td>%zmm31</td>
</tr>
</tbody>
</table>

Also vectors of 8/16/32/64/128/256/512-bit integers
XMM/YMM/ZMM Relationship
Detail: XMM/SSE2/3/4 Register

- 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`

With x86-64 all FPU operations are done in xmm scalar
AVX Instructions: Examples

- Double precision 4-way vector add: `vaddpd %ymm0, %ymm1, %ymm2`

Since AVX: Intel has 3-operand instructions and added AVX-style SSE instructions
Instruction Names

packed (vector)

\textbf{addps}

\textbf{vaddpd}

\textbf{single precision}

single slot (scalar)

\textbf{addss}

\textbf{addsd}

3 op \quad \text{double precision}

\textit{Compiler will use this for floating point}

\textit{SSE vs AVX: register operand decides}

\textbullet \quad \textit{Set arch to AVX even for SSE code on SandyBridge and newer}
### Inner product of two vectors

- Single precision arithmetic
- Compiled: not vectorized, uses SSE instructions

```c
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;
}
```

```assembly
# result = 0.0
xorps %xmm1, %xmm1  # i = 0
xorl %ecx, %ecx     # goto middle
jmp .L8             # loop:
.L10:
    movslq %ecx,%rax # icpy = i
    incl %ecx        # i++
    movss (%rsi,%rax,4), %xmm0 # t = y[icpy]
    mulss (%rdi,%rax,4), %xmm0 # t *= x[icpy]
    addss %xmm0, %xmm1 # result += t
.L8:
    cmpl %edx, %ecx   # i:n
    jl .L10           # if < goto loop
    movaps %xmm1, %xmm0 # return result
    ret
```

- Single precision arithmetic
- Compiled: not vectorized, uses SSE instructions
SSE/AVX: How to Take Advantage?

- Necessary: fine grain parallelism
- Options (ordered by effort):
  - Use vectorized libraries (easy, not always available)
  - Compiler vectorization (good option)
  - Use intrinsics (this lecture)
  - Write assembly
- We will focus on 4-way (SSE single and AVX double)
Vector Instructions: Language Extension

■ Data types
  ▪ __m128 f; // = {float f3, f2, f1, f0}
  ▪ __m128d d; // = {double d1, d0}
  ▪ __m128i i; // = {int i3, i2, i1, i0}
  ▪ __m256 f; // = {float f7, ..., f1, f0}
  ▪ ...
  ▪ __m512 f; // = {float f15, ..., f1, f0}
  ▪ ...

■ Intrinsics
  ▪ Native instructions: _mm_add_ps(), _mm256_mul_pd(),...
  ▪ Multi-instruction: _mm_setr_ps(), _mm512_set1_pd(),...

■ Macros
  ▪ Block operations: __MM_TRANSPOSE4_PS(), ...
  ▪ Helper: __MM_SHUFFLE(), __MM_GET_EXCEPTION_MASK()
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

Intel® 64 and IA-32 Architectures
Software Developer's Manual

Combined Volumes:
1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4

NOTE: This document contains all four volumes of the Intel 64 and IA-32 Architectures Software Developer's Manual: Basic Architecture, Order Number 253665; Instruction Set Reference A-Z, Order Number 325383; System Programming Guide, Order Number 325384; Model-Specific Registers, Order Number 335592. Refer to all four volumes when evaluating your design needs.

About 5,000 pages

Order Number: 325462-069US
January 2019
Visual Conventions We Will Use

- **Memory**
  - Increasing address
  - Memory

- **Registers**
  - Commonly:
    - R3 R2 R1 R0
  - We will use:
    - R0 R1 R2 R3
SSE4/AVX2 Data Types

- **SSE4 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
  ```

- **AVX2 data types**

  ```
  __m256  f;   // = {float f0, f1, ..., f7}
  __m256d d;   // = {double d0, d1, d2, d3}
  __m128i i;   // 32 8-bit, 16 16-bit, 8 32-bit, or 4 64-bit ints
  ```

SSE4==AVX128 and AVX2 can be mixed if done carefully
# SSE4/AVX128 4 x float Intrinsics

## 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);
```

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

- Same result as

```c
__m128 t = _mm_set_ps(4.0, 3.0, 2.0, 1.0)
```
AVX2 4 x double Intrinsics

- Instructions
  - Naming convention: _mm256_<intrin_op>_<suffix>
  - Example:

    ```c
    // a is 32-byte aligned
da[4] = {1.0, 2.0, 3.0, 4.0};
    __m256d t = _mm256_load_pd(a);
    ```

    LSB
    - 1.0 2.0 3.0 4.0

    - Same result as

    ```c
    __m256d t = _mm256_set_pd(4.0, 3.0, 2.0, 1.0)
    ```

    p: packed
d: double precision
SSE4/AVX128 and AVX2 Intrinsics

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

```c
a = _mm_load_ps(p);  // p 16-byte aligned

a = _mm_loadu_ps(p); // p not aligned

a = _mm256_load_pd(p);  // p 32-byte aligned

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

load_ps/pd on unaligned pointer: seg fault

*avoid unaligned (can be expensive) on recent Intel possibly no penalty*
# Example: Load and Set in SSE4/AVX128 and AVX2

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE/AVX Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm256_broadcast_pd</td>
<td>Broadcast 128 bits from memory to all elements of dst</td>
<td>_mm256_broadcast_pd</td>
</tr>
<tr>
<td>_mm_i32gather_epi32</td>
<td>Gather 32-bit integers from memory using 32-bit indices</td>
<td>_mm_i32gather_epi32</td>
</tr>
<tr>
<td>_mm_load_ss</td>
<td>Load the low value and clear the three high values</td>
<td>_mm_load_ss</td>
</tr>
<tr>
<td>_mm256_loadu2_m128i</td>
<td>Load two 128-bit values from memory, and combine them into dst</td>
<td>_mm256_loadu2_m128i</td>
</tr>
<tr>
<td>_mm256_load_ps</td>
<td>Load eight values, address aligned</td>
<td>_mm256_load_ps</td>
</tr>
<tr>
<td>_mm_loadu_ps</td>
<td>Load four values, address unaligned</td>
<td>_mm_loadu_ps</td>
</tr>
<tr>
<td>_mm_maskload_pd</td>
<td>Load packed double-precision elements from memory using mask</td>
<td>_mm_maskload_pd</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE/AVX 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</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>
<tr>
<td>_mm256_set1_pd</td>
<td>Set all four words with the same value</td>
<td>Composite</td>
</tr>
<tr>
<td>_mm256_set_ps</td>
<td>Set eight values</td>
<td>Composite</td>
</tr>
<tr>
<td>_mm256_set1_epi64x</td>
<td>Broadcast 64-bit integer a to all elements of dst</td>
<td>Composite</td>
</tr>
<tr>
<td>_mm256i_setzero_si256</td>
<td>Clear all 256 bits</td>
<td>Composite</td>
</tr>
</tbody>
</table>
# Stores Less Powerful than Loads

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE/AVX Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_mm256_store_pd</code></td>
<td>Store 4 doubles to aligned memory</td>
<td><code>vmovapd</code></td>
</tr>
<tr>
<td><code>_mm_store_pd</code></td>
<td>Store 2 doubles to aligned memory</td>
<td><code>movapd</code></td>
</tr>
<tr>
<td><code>_mm256_maskstore_ps</code></td>
<td>Store single-precision elements from a into memory using mask</td>
<td><code>vmaskmovps</code></td>
</tr>
<tr>
<td><code>_mm256_stream_si256</code></td>
<td>Non-temporal store</td>
<td><code>vmovntdq</code></td>
</tr>
<tr>
<td><code>_mm256_storeu2_m128d</code></td>
<td>Store the high and low 128-bit into memory two different locations</td>
<td><code>composite</code></td>
</tr>
<tr>
<td><code>_mm_storel_epi64</code></td>
<td>Store 64 bit of XMM register to memory</td>
<td><code>movq</code></td>
</tr>
<tr>
<td><code>_mm_store1_pd</code></td>
<td>Store lowest double to memory</td>
<td><code>composite</code></td>
</tr>
</tbody>
</table>
Constants

\[
\begin{align*}
\text{a} & = \_\text{mm\_set\_ps}(4.0, 3.0, 2.0, 1.0); \\
\text{b} & = \_\text{mm\_set1\_ps}(1.0); \\
\text{c} & = \_\text{mm\_set\_ss}(1.0); \\
\text{d} & = \_\text{mm\_setzero\_ps}();
\end{align*}
\]
## Arithmetic

### SSE/SSE2

<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>RSQRTSS</td>
</tr>
<tr>
<td>_mm_rsqrt_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 SSE 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>

### AVX

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Operation</th>
<th>Corresponding SSE4 Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm256_add_pd</td>
<td>Addition</td>
<td>vaddpd</td>
</tr>
<tr>
<td>_mm256_addsub_pd</td>
<td>Subtract and add</td>
<td>vaddsubpd</td>
</tr>
<tr>
<td>_mm256_hadd_pd</td>
<td>Horizontal add</td>
<td>vhadddp</td>
</tr>
</tbody>
</table>
| _mm256_fmaddsub_pd | Fused multiply add | vfmadddsub132pd
|                 |                         | vfmaddsub213pd
|                 |                         | vfmaddsub231pd
| _mm256_fmsub_pd | Fused multiply subtract | vfmsub132pd
|                 |                         | vfmsub213pd
|                 |                         | vfmsub231pd
| _mm256_div_pd   | Division                | vdivpd                        |
| _mm256_sub_pd   | Subtraction             | vsubpd                        |
| _mm256_dp_ps    | Dot product             | vdpps                         |
Arithmetic

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

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

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

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

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

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

\[ c = _\text{mm\_mul\_pd}(a, b); \]
Example: AVX

```c
#include <immintrin.h>

// n a multiple of 4, x is 32-byte aligned
void addindex_vec(double *x, int n) {
  __m256d index, x_vec;
  for (int i = 0; i < n; i+=4) {
    x_vec = _mm256_load_pd(x+i);              // load 4 double
    index = _mm256_set_pd(i+3, i+2, i+1, i);  // create vector with indexes
    x_vec = _mm256_add_pd(x_vec, index);      // add the two
    _mm256_store_pd(x+i, x_vec);              // store back
  }
}
```

```c
void addindex(double *x, int n) {
  for (int i = 0; i < n; i++)
    x[i] = x[i] + i;
}
```
Example: SSE4

```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
    }
}
```

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

\[c = \_\text{mm\_hsub\_ps}(a, b)\]
Comparisons: SSE4/AVX128

<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>CMPELESS</td>
</tr>
<tr>
<td>_mm_cmple_ps</td>
<td>Less Than or Equal</td>
<td>CMPELES</td>
</tr>
<tr>
<td>_mm_cmpgt_ss</td>
<td>Greater Than</td>
<td>CMPLTSS</td>
</tr>
<tr>
<td>_mm_cmpgt_ps</td>
<td>Greater Than</td>
<td>CMPLTPS</td>
</tr>
<tr>
<td>_mm_cmpge_ss</td>
<td>Greater Than or Equal</td>
<td>CMPELESS</td>
</tr>
<tr>
<td>_mm_cmpge_ps</td>
<td>Greater Than or Equal</td>
<td>CMPELES</td>
</tr>
<tr>
<td>_mm_cmpneq_ss</td>
<td>Not Equal</td>
<td>CMPEQSS</td>
</tr>
<tr>
<td>_mm_cmpneq_ps</td>
<td>Not Equal</td>
<td>CMPEQPS</td>
</tr>
<tr>
<td>_mm_cmpltl_ss</td>
<td>Not Less Than</td>
<td>CMPLTSS</td>
</tr>
<tr>
<td>_mm_cmpltl_ps</td>
<td>Not Less Than</td>
<td>CMPLTLPSP</td>
</tr>
<tr>
<td>_mm_cmplne_ss</td>
<td>Not Less Than or Equal</td>
<td>CMPLNELESS</td>
</tr>
<tr>
<td>_mm_cmplne_ps</td>
<td>Not Less Than or Equal</td>
<td>CMPLNLEPS</td>
</tr>
<tr>
<td>_mm_cmpngt_ss</td>
<td>Not Greater Than</td>
<td>CMPLTSS</td>
</tr>
<tr>
<td>_mm_cmpngt_ps</td>
<td>Not Greater Than</td>
<td>CMPLTPS</td>
</tr>
<tr>
<td>_mm_cmpnge_ss</td>
<td>Not Greater Than or Equal</td>
<td>CMPLNELESS</td>
</tr>
<tr>
<td>_mm_cmpnge_ps</td>
<td>Not Greater Than or Equal</td>
<td>CMPLNLEPS</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</td>
<td>COMISS</td>
</tr>
<tr>
<td>_mm_comigt_ss</td>
<td>Greater Than</td>
<td>COMISS</td>
</tr>
<tr>
<td>_mm_comige_ss</td>
<td>Greater Than or Equal</td>
<td>COMISS</td>
</tr>
<tr>
<td>_mm_comineq_ss</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_ucomilt_ss</td>
<td>Less Than</td>
<td>UCOMISS</td>
</tr>
<tr>
<td>_mm_ucomile_ss</td>
<td>Less Than or Equal</td>
<td>UCOMISS</td>
</tr>
<tr>
<td>_mm_ucomigt_ss</td>
<td>Greater Than</td>
<td>UCOMISS</td>
</tr>
<tr>
<td>_mm_ucomige_ss</td>
<td>Greater Than or Equal</td>
<td>UCOMISS</td>
</tr>
<tr>
<td>_mm_ucomineq_ss</td>
<td>Not Equal</td>
<td>UCOMISS</td>
</tr>
</tbody>
</table>
Comparisons: AVX

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_cmp_pd</td>
<td>Controlled by imm8</td>
<td>vcmppd</td>
</tr>
<tr>
<td>_mm256_cmp_pd</td>
<td>Controlled by imm8</td>
<td>vcmppd</td>
</tr>
</tbody>
</table>

\[imm8:\]

<table>
<thead>
<tr>
<th>Value</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>_CMP_EQ_OQ</td>
</tr>
<tr>
<td>1</td>
<td>_CMP_LT_OS</td>
</tr>
<tr>
<td>2</td>
<td>_CMP_LE_OS</td>
</tr>
<tr>
<td>3</td>
<td>_CMP_UNORD_Q</td>
</tr>
<tr>
<td>4</td>
<td>_CMP_NEQ_UQ</td>
</tr>
<tr>
<td>5</td>
<td>_CMP_NLT_US</td>
</tr>
<tr>
<td>6</td>
<td>_CMP_NLE_US</td>
</tr>
<tr>
<td>7</td>
<td>_CMP_ORD_Q</td>
</tr>
<tr>
<td>8</td>
<td>_CMP_EQ_UQ</td>
</tr>
<tr>
<td>9</td>
<td>_CMP_NGE_US</td>
</tr>
<tr>
<td>10</td>
<td>_CMP_NGT_US</td>
</tr>
<tr>
<td>11</td>
<td>_CMP_FALSE_OQ</td>
</tr>
<tr>
<td>12</td>
<td>_CMP_EQ_US</td>
</tr>
<tr>
<td>13</td>
<td>_CMP_TRUE_UQ</td>
</tr>
<tr>
<td>16</td>
<td>_CMP_EQ_OS</td>
</tr>
<tr>
<td>17</td>
<td>_CMP_LT_OQ</td>
</tr>
<tr>
<td>18</td>
<td>_CMP_LE_OQ</td>
</tr>
<tr>
<td>19</td>
<td>_CMP_UNORD_S</td>
</tr>
<tr>
<td>20</td>
<td>_CMP_NEQ_US</td>
</tr>
<tr>
<td>21</td>
<td>_CMP_NLT_UQ</td>
</tr>
<tr>
<td>22</td>
<td>_CMP_NLE_UQ</td>
</tr>
<tr>
<td>23</td>
<td>_CMP_ORD_S</td>
</tr>
<tr>
<td>24</td>
<td>_CMP_EQ_US</td>
</tr>
<tr>
<td>25</td>
<td>_CMP_NGE_UQ</td>
</tr>
<tr>
<td>26</td>
<td>_CMP_NGT_UQ</td>
</tr>
<tr>
<td>27</td>
<td>_CMP_FALSE_OQ</td>
</tr>
<tr>
<td>28</td>
<td>_CMP_EQ_US</td>
</tr>
<tr>
<td>29</td>
<td>_CMP_TRUE_UQ</td>
</tr>
<tr>
<td>30</td>
<td>_CMP_GT_OQ</td>
</tr>
<tr>
<td>31</td>
<td>_CMP_TRUE_US</td>
</tr>
</tbody>
</table>
Comparison Example: SSE4

<table>
<thead>
<tr>
<th></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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1.0</td>
<td>1.5</td>
<td>3.0</td>
<td>3.5</td>
</tr>
</tbody>
</table>

\[
c = _mm_cmpeq_ps(a, b);
\]

**analogous:**

\[
c = _mm_cmple_ps(a, b);
\]

\[
c = _mm_cmpge_ps(a, b);
\]

\[
c = _mm_cmpgt_ps(a, b);
\]

\[
c = _mm_cmplt_ps(a, b);
\]

\[
\text{etc.}
\]

Each field:

- 0xffffffff if true
- 0x0 if false

Return type: __m128
SSE4 Code Example

```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.;
    }
}
```

```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);
    }
}
```
# Shuffles: SSE

**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>MOVSS</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>MOVS LDUP</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>PSHUF B</td>
</tr>
<tr>
<td>_mm_alignr_epi8</td>
<td>Shift</td>
<td>PALIGN R</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: AVX

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__m128 _mm256_extractf128_ps (__m256 a, const int imm8)</code></td>
<td>Extracts single 128-bit float from <code>__m256</code></td>
</tr>
<tr>
<td><code>__m128i _mm256_extractf128_si256 (__m256i a, const int imm8)</code></td>
<td>Extracts single 128-bit integer from <code>__m256i</code></td>
</tr>
<tr>
<td><code>__m128i _mm256_extracti128_si256 (__m256i a, const int imm8)</code></td>
<td>Extracts single 128-bit integer from <code>__m256i</code></td>
</tr>
<tr>
<td><code>__m256i _mm256_insert_epi16 (__m256i a, __int16 i, const int index)</code></td>
<td>Inserts 16-bit integer into <code>__m256i</code></td>
</tr>
<tr>
<td><code>__m256i _mm256_insert_epi32 (__m256i a, __int32 i, const int index)</code></td>
<td>Inserts 32-bit integer into <code>__m256i</code></td>
</tr>
<tr>
<td><code>__m256i _mm256_insert_epi64 (__m256i a, __int64 i, const int index)</code></td>
<td>Inserts 64-bit integer into <code>__m256i</code></td>
</tr>
<tr>
<td><code>__m256i _mm256_insert_epi8 (__m256i a, __int8 i, const int index)</code></td>
<td>Inserts 8-bit integer into <code>__m256i</code></td>
</tr>
<tr>
<td><code>__m256d _mm256_insertf128_pd (__m256d a, __m128d b, int imm8)</code></td>
<td>Inserts double 128-bit float into <code>__m256d</code></td>
</tr>
<tr>
<td><code>__m256 _mm256_insertf128_ps (__m256 a, __m128 b, int imm8)</code></td>
<td>Inserts single 128-bit float into <code>__m256</code></td>
</tr>
<tr>
<td><code>__m256i _mm256_inserti128_si256 (__m256i a, __m128i b, int imm8)</code></td>
<td>Inserts single 128-bit integer into <code>__m256i</code></td>
</tr>
<tr>
<td><code>__m128d _mm_permute_pd (__m128d a, int imm8)</code></td>
<td>Permutes double 128-bit data</td>
</tr>
<tr>
<td><code>__m256d _mm256_permute_pd (__m256d a, int imm8)</code></td>
<td>Permutes double 256-bit data</td>
</tr>
<tr>
<td><code>__m128 _mm_permute_ps (__m128 a, int imm8)</code></td>
<td>Permutes single 128-bit data</td>
</tr>
<tr>
<td><code>__m256 _mm_permute_ps (__m256 a, int imm8)</code></td>
<td>Permutes single 256-bit data</td>
</tr>
<tr>
<td><code>__m256d _mm256_permute2f128_pd (__m256d a, __m256d b, int imm8)</code></td>
<td>Permutes double 128-bit data with 256-bit shift</td>
</tr>
<tr>
<td><code>__m256 _mm256_permute2f128_ps (__m256 a, __m256 b, int imm8)</code></td>
<td>Permutes single 128-bit data with 256-bit shift</td>
</tr>
<tr>
<td><code>__m256i _mm256_permute2f128_si256 (__m256i a, __m256i b, int imm8)</code></td>
<td>Permutes single 128-bit integer with 256-bit shift</td>
</tr>
<tr>
<td><code>__m256i _mm256_permute2x128_si256 (__m256i a, __m256i b, const int imm8)</code></td>
<td>Permutes single 256-bit integer with 256-bit shift</td>
</tr>
<tr>
<td><code>__m256i _mm256_permute4x64_epi64 (__m256i a, const int imm8)</code></td>
<td>Permutes single 256-bit integer with 4x64-bit shift</td>
</tr>
</tbody>
</table>
Shuffles: SSE

c = _mm_unpacklo_ps(a, b);

c = _mm_unpackhi_ps(a, b);
Shuffles

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

helper macro to create mask

\[
\begin{array}{cccc}
\text{LSB} & 1.0 & 2.0 & 3.0 & 4.0 \\
\text{a} & & & & \\
\end{array}
\quad
\begin{array}{cccc}
\text{LSB} & 0.5 & 1.5 & 2.5 & 3.5 \\
\text{b} & & & & \\
\end{array}
\quad
\begin{array}{cccc}
\text{LSB} & c_0 & c_1 & c_2 & c_3 \\
c & & & & \\
\end{array}
\]

\[c_0 = a_i\]
\[c_1 = a_j\]
\[c_2 = b_k\]
\[c_3 = b_l\]
\[i, j, k, l \in \{0, 1, 2, 3\}\]
Example: Gather Operation

7 instructions, this is one good way of doing it
#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));
}

With AVX2 and AVX-512 gather/scatter operations are introduced
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!](#))
Example: Complex Multiplication SSE3

$$\begin{align*}
(a + ib)(c + id) &= (ac - bd) + i(ad + bc)
\end{align*}$$
## Looking at the Assembly

### SSE3:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movapd xmm0, XMMWORD PTR A</td>
<td></td>
</tr>
<tr>
<td>movddup xmm2, QWORD PTR B</td>
<td></td>
</tr>
<tr>
<td>mulpd xmm2, xmm0</td>
<td></td>
</tr>
<tr>
<td>movddup xmm1, QWORD PTR B+8</td>
<td></td>
</tr>
<tr>
<td>shufpd xmm0, xmm0, 1</td>
<td></td>
</tr>
<tr>
<td>mulpd xmm1, xmm0</td>
<td></td>
</tr>
<tr>
<td>addsubpd xmm2, xmm1</td>
<td></td>
</tr>
<tr>
<td>movapd XMMWORD PTR C, xmm2</td>
<td></td>
</tr>
</tbody>
</table>

### SSE2:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movsd xmm3, QWORD PTR A</td>
<td></td>
</tr>
<tr>
<td>movapd xmm4, xmm3</td>
<td></td>
</tr>
<tr>
<td>movsd xmm5, QWORD PTR A+8</td>
<td></td>
</tr>
<tr>
<td>mulpd xmm0, xmm5</td>
<td></td>
</tr>
<tr>
<td>movapd xmm1, QWORD PTR B</td>
<td></td>
</tr>
<tr>
<td>movsd xmm4, xmm1</td>
<td></td>
</tr>
<tr>
<td>mulsd xmm5, xmm1</td>
<td></td>
</tr>
<tr>
<td>movsd xmm2, QWORD PTR B+8</td>
<td></td>
</tr>
<tr>
<td>mulsd xmm0, xmm2</td>
<td></td>
</tr>
<tr>
<td>mulsd xmm3, xmm2</td>
<td></td>
</tr>
<tr>
<td>subsd xmm4, xmm0</td>
<td></td>
</tr>
<tr>
<td>movsd QWORD PTR C, xmm4</td>
<td></td>
</tr>
<tr>
<td>addsd xmm5, xmm3</td>
<td></td>
</tr>
<tr>
<td>movsd QWORD PTR C, xmm5</td>
<td></td>
</tr>
</tbody>
</table>

In SSE2 Intel C++ generates *scalar* code (better?)
int dwmonitor(float *X, double *D) {
    __m128d u1, u2, u3, u4, u5, u6, u7, u8, x1, x10, x13, x14, x17, x18, x19, x2, x3, x4, x6, x7, x8, x9;
    int w1;
    unsigned Xm = _mm_getcsr();
    _mm_setcsr(Xm & 0xffff0000 | 0x0000dfc0);
    u5 = _mm_set1_pd(0.0);
    u2 = _mm_cvtps_pd(_mm_addsub_ps(_mm_set1_ps(FLT_MIN), _mm_set1_ps(X[0])));
    u1 = _mm_set1_pd(1.0, (-1.0));
    for(int i5 = 0; i5 <= 2; i5++) {
        x6 = _mm_addsub_pd(_mm_set1_pd((DBL_MIN + DBL_MIN)), _mm_loadupsd(&D[i5]));
        x1 = _mm_addsub_pd(_mm_set1_pd(0.0), u1);
        x2 = _mm_mul_pd(x1, x6);
        x3 = _mm_mul_pd(_mm_shuffled_(x1, x1, _MM_SHUFFLE2(0, 1)), x6);
        x4 = _mm_shuffled_(x1, x1, _MM_SHUFFLE2(0, 1), _mm_mn_pd(x3, x2));
        u3 = _mm_add_pd(_mm_shuffled_(x4, x4, _MM_SHUFFLE2(0, 1)), _mm_mn_pd(x3, x2), _mm_set1_pd(DBL_MIN));
        u5 = _mm_add_pd(u5, u3);
        x7 = _mm_addsub_pd(_mm_set1_pd(0.0), u1);
        x8 = _mm_mul_pd(x7, u2);
        x9 = _mm_mul_pd(_mm_shuffled_(x7, x7, _MM_SHUFFLE2(0, 1)), u2);
        x10 = _mm_mn_pd(x9, x8);
        u1 = _mm_add_pd(_mm_shuffled_(x10, x10, _MM_SHUFFLE2(0, 1)), _mm_mn_pd(x9, x8), _mm_set1_pd(DBL_MIN));
    }
    u6 = _mm_set1_pd(0.0);
    for(int i3 = 0; i3 <= 1; i3++) {
        u8 = _mm_shuffled_(u8, _mm_shuffled_(u7, u7, _MM_SHUFFLE2(0, 1)), _mm_set1_ps(X[i3 + 1]));
        u7 = _mm_shuffled_(u8, _mm_set1_ps(FLT_MIN), _mm_set1_ps(X[i3 + 1]));
        x14 = _mm_add_pd(u7, _mm_shuffled_(u7, u7, _MM_SHUFFLE2(0, 1)));
        x13 = _mm_shuffled_(x14, x14, _MM_SHUFFLE2(0, 1));
        x14 = _mm_shuffled_(x14, x13, _mm_set1_pd(x14, x13), _MM_SHUFFLE2(0, 1));
        u4 = _mm_shuffled_(x14, x13, _MM_SHUFFLE2(0, 1), _mm_set1_pd(x13, x13));
        u6 = _mm_shuffled_(x14, x13, _MM_SHUFFLE2(0, 1), _mm_set1_pd(x13, x13));
    }
    x17 = _mm_addsub_pd(_mm_set1_pd(0.0), u6);
    x18 = _mm_addsub_pd(_mm_set1_pd(0.0), u5);
    x19 = _mm_shuffled_(x18, x18, _MM_SHUFFLE2(0, 1));
    w1 = (_mm_testc_si128(_mm_castpd_si128(x19), _mm_set_epi32(0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff)) -
    _mm_loadupsd(&D[0]));
    __asm__ __volatile__ {
        if (_mm_getcsr() & 0x0d) {
            w1 = _mm_setcsts(xm);
        }
    }
    return w1;
}
Beyond AVX2: AVX-512/KNC

Synopsis

```c
__m512 __mm512_4fmadd_ps (__m512 a, __m512 b0, __m512 b1, __m512 b2, __m512 b3, __m128 c)
```

#include <immintrin.h>

Instruction: v4fmaddps zmm {k}, zmm+3, m128
CPUID Flags: AVX512_4FMAPS

Description

Multiply packed single-precision (32-bit) floating-point elements specified in 4 consecutive operands $b0$ through $b3$ by the 4 corresponding packed elements in $c$, accumulating with the corresponding elements in $a$. Store the results in dst.

Operation

```c
dst := a
FOR m := 0 to 3
    FOR j := 0 to 15
        i = j*32
        n = m*32
        dst[i+31:i] := RoundFPControl_MXCSR(dst[i+31:i] + b[m][i+31:i] * c[n+31:n])
    ENDFOR
ENDFOR
```

dst[MAX:512] := 0
Beyond Intel

- **IBM PowerPC**
  - PowerPC: AltiVec, Cell Processor
  - POWER: VMX, VMX128, VSX
  - Supercomputer: Double FPU, Quad FPU

- **AMD**
  - Early competition with Intel: 3DNow! vs. MMX/SSE
  - Later: FMA4 vs. AVX/FMA3
  - Now: cross-licensed

- **ARM**
  - NEON instruction set
  - Complex numbers, half precision, advanced SIMD instructions

- **RISC V**
  - Hwatcha vector unit
How to Write Good Vector Code?

- Take the “right” algorithm and the “right” data structures
  - Fine grain parallelism
  - Correct alignment in memory
  - Contiguous arrays

- Use a good compiler (e.g., vendor compiler)

- First: Try compiler vectorization
  - Right options, pragmas and dynamic memory functions
    (Inform compiler about data alignment, loop independence,...)
  - Check generated assembly code and runtime

- If necessary: Write vector code yourself
  - Most expensive subroutine first
  - *Use intrinsics*, no (inline) assembly
  - *Important*: Understand the ISA
  - *Annoying*: understand the microarchitecture and instruction cost
18-645: How to Write Fast Code

1 Introduction

The growth in the performance of computing platforms in the past few decades has followed a reliable pattern usually referred to as Moore’s Law. Moore observed in 1965 [1] that the number of transistors per chip roughly doubles every 18 months and predicted—correctly—that this trend would continue. In parallel, due to the shrinking size of transistors, CPU frequencies could be increased at roughly the same exponential rate. This trend has been the big supporter for many performance demanding applications in scientific computing (such as climate modeling and other physics simulations), consumer computing (such as audio, image, and video processing), and embedded computing (such as control, communication, and signal processing). In fact, these domains have a practically unlimited need for performance (for example, the ever growing need for higher resolution videos), and it seems that the evolution of computers is well on track to support these needs.

However, everything comes at a price, and in this case it is the increasing difficulty of writing the fastest possible software. In this tutorial, we focus on numerical software. By that we mean code that mainly consists of floating point computations.

The problem. To understand the problem we investigate Fig. 1, which considers various Intel architectures from the first Pentium to the (at the time of this writing) latest Core2 Extreme. The x-axis shows the year of release. The y-axis, in log-scale, shows both the CPU frequency (in MHz) and the single double precision theoretical peak performance (in Mflops/s = Mega Floating point OPerations per Second) of the respective machines.

Tentative Course Calendar:

<table>
<thead>
<tr>
<th>Date</th>
<th>Day</th>
<th>Class Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>August</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Mon.</td>
<td>Semester Begin – Overview of course</td>
</tr>
<tr>
<td>29</td>
<td>Wed.</td>
<td>Architecture and its impact on performance</td>
</tr>
<tr>
<td>September</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Mon.</td>
<td>Labor Day; No Classes</td>
</tr>
<tr>
<td>5</td>
<td>Wed.</td>
<td>Designing fast kernels</td>
</tr>
<tr>
<td>10</td>
<td>Mon.</td>
<td>Benchmarking</td>
</tr>
<tr>
<td>12</td>
<td>Wed.</td>
<td>Intro to SIMD Programming</td>
</tr>
<tr>
<td>17</td>
<td>Mon.</td>
<td>Compiler Optimizations</td>
</tr>
<tr>
<td>19</td>
<td>Wed.</td>
<td>Class Cancelled – Replaced with Project Kickoff Meetings</td>
</tr>
<tr>
<td>24</td>
<td>Mon.</td>
<td>The memory hierarchy</td>
</tr>
<tr>
<td>26</td>
<td>Wed.</td>
<td>Optimization for the memory hierarchy – Part 1</td>
</tr>
<tr>
<td>October</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Mon.</td>
<td>Optimization for the memory hierarchy – Part 2</td>
</tr>
<tr>
<td>3</td>
<td>Wed.</td>
<td>Intro to Explicit Parallelism</td>
</tr>
<tr>
<td>8</td>
<td>Mon.</td>
<td>Collective Communications – Part 1</td>
</tr>
<tr>
<td>10</td>
<td>Wed.</td>
<td>Class Cancelled – Replaced with Project Midterm Review Meetings</td>
</tr>
<tr>
<td>15</td>
<td>Mon.</td>
<td>Collective Communications – Part 2</td>
</tr>
<tr>
<td>17</td>
<td>Wed.</td>
<td>Designing Efficient Communication Patterns – Part 1</td>
</tr>
<tr>
<td>19</td>
<td>Fri.</td>
<td>Mid-Semester Break: No Classes</td>
</tr>
<tr>
<td>22</td>
<td>Mon.</td>
<td>Designing Efficient Communication Patterns – Part 2</td>
</tr>
<tr>
<td>24</td>
<td>Wed.</td>
<td>Shared Memory Parallelism</td>
</tr>
<tr>
<td>29</td>
<td>Mon.</td>
<td>From Shared Memory to Heterogeneous Architectures</td>
</tr>
<tr>
<td>31</td>
<td>Wed.</td>
<td>Introduction to Algorithm Design</td>
</tr>
<tr>
<td>November</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Mon.</td>
<td>Putting everything together</td>
</tr>
<tr>
<td>7</td>
<td>Wed.</td>
<td>Class Cancelled – Replaced with Project Final Review Meetings</td>
</tr>
<tr>
<td>12</td>
<td>Mon.</td>
<td>Midterm Review</td>
</tr>
<tr>
<td>14</td>
<td>Wed.</td>
<td>Midterm Exam</td>
</tr>
<tr>
<td>19</td>
<td>Mon.</td>
<td>Project Meeting Time</td>
</tr>
<tr>
<td>21-23</td>
<td>W-F</td>
<td>Thanksgiving Break: No Classes</td>
</tr>
<tr>
<td>26</td>
<td>Mon.</td>
<td>Project Presentations</td>
</tr>
<tr>
<td>28</td>
<td>Wed.</td>
<td>Project Presentations</td>
</tr>
<tr>
<td>December</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Mon.</td>
<td>Project Presentations</td>
</tr>
<tr>
<td>5</td>
<td>Wed.</td>
<td>Last Day of Class – Wrap Up</td>
</tr>
</tbody>
</table>

https://courses.ece.cmu.edu/18645
http://www.ece.cmu.edu/~franzf/papers/gttseU.pdt
https://www.inf.ethz.ch/personal/markusp/teaching/18-645-CMU-spring08/course.html

## Notes
- The exact topics of the lectures are subject to change.
- We do not anticipate changing any of the other dates (exams, assignments, etc.)

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan 15</td>
<td>MLK Day. <em>No class</em></td>
<td></td>
</tr>
<tr>
<td>Jan 17</td>
<td>Why parallelism (pdf, video)</td>
<td>Assignment 1 out (pdf)</td>
</tr>
<tr>
<td>Jan 19</td>
<td>Modern multicore processors (pdf, video)</td>
<td></td>
</tr>
<tr>
<td>Jan 22</td>
<td>Parallel programming models (pdf, video)</td>
<td></td>
</tr>
<tr>
<td>Jan 24</td>
<td>Parallel programming basics (pdf, video)</td>
<td>Assignment 1 due for waitlisted students</td>
</tr>
<tr>
<td>Jan 26</td>
<td>Work distribution and scheduling (pdf, video)</td>
<td></td>
</tr>
<tr>
<td>Jan 29</td>
<td><strong>Recitation</strong>: ILP, SIMD instructions (pdf, pptx, video)</td>
<td>Assignment 1 due for registered students, assignment 2 out (pdf)</td>
</tr>
<tr>
<td>Jan 31</td>
<td>Graphic processing units and CUDA (pdf, video)</td>
<td></td>
</tr>
<tr>
<td>Feb 2</td>
<td><strong>Recitation</strong>: CUDA programming 1 (pdf, pptx)</td>
<td></td>
</tr>
<tr>
<td>Feb 5</td>
<td>Locality, communication, and contention (pdf, video)</td>
<td></td>
</tr>
<tr>
<td>Feb 7</td>
<td>Application case studies (pdf, video)</td>
<td></td>
</tr>
<tr>
<td>Feb 9</td>
<td><strong>Recitation</strong>: CUDA programming 2 (pdf, pptx)</td>
<td></td>
</tr>
<tr>
<td>Feb 12</td>
<td>Workload-driven performance evaluation (pdf, video)</td>
<td>Assignment 2 due, assignment 3 out (pdf)</td>
</tr>
<tr>
<td>Feb 14</td>
<td>Snooping-based cache coherence (pdf, video)</td>
<td></td>
</tr>
<tr>
<td>Feb 16</td>
<td><strong>Recitation</strong>: Understanding Assignment 3 (pdf, pptx, video)</td>
<td></td>
</tr>
<tr>
<td>Feb 19</td>
<td>Directory-based cache coherence (pdf, video)</td>
<td></td>
</tr>
<tr>
<td>Feb 21</td>
<td>Snooping implementation (pdf, video)</td>
<td></td>
</tr>
</tbody>
</table>