How to Write Fast Code

SIMD Vectorization

18-645, spring 2008
13th and 14th Lecture

Instructor: Markus Püschel
Guest Instructor: Franz Franchetti
TAs: Srinivas Chellappa (Vas) and Frédéric de Mesmay (Fred)

Organization

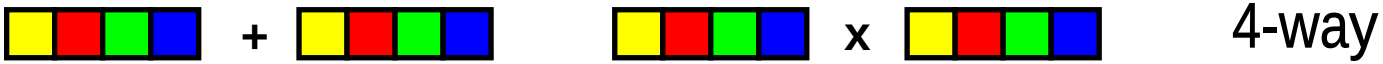
- Overview
  - Idea, benefits, reasons, restrictions
  - History and state-of-the-art floating-point SIMD extensions
  - How to use it: compiler vectorization, class library, intrinsics, inline assembly
- Writing code for Intel's SSE
  - Compiler vectorization
  - Intrinsics: instructions
  - Intrinsics: common building blocks
- Selected topics
  - SSE integer instructions
  - Other SIMD extensions: AltiVec/VMX, Cell SPU
- Conclusion: How to write good vector code

Organization

- Overview
  - Idea, benefits, reasons, restrictions
  - History and state-of-the-art floating-point SIMD extensions
  - How to use it: compiler vectorization, class library, intrinsics, inline assembly
- Writing code for Intel's SSE
  - Compiler vectorization
  - Intrinsics: instructions
  - Intrinsics: common building blocks
- Selected topics
  - SSE integer instructions
  - Other SIMD extensions: AltiVec/VMX, Cell SPU
- Conclusion: How to write good vector code

SIMD (Signal Instruction Multiple Data) 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



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

Evolution of Intel Vector Instructions

- MMX (1996, Pentium)
  - *CPU-based MPEG decoding*
  - Integers only, 64-bit divided into 2 x 32 to 8 x 8
  - Phased out with SSE4
- SSE (1999, Pentium III)
  - *CPU-based 3D graphics*
  - 4-way float operations, single precision
  - 8 new 128 bit Register, 100+ instructions
- SSE2 (2001, Pentium 4)
  - *High-performance computing*
  - Adds 2-way float ops, double-precision; same registers as 4-way single-precision
  - Integer SSE instructions make MMX obsolete
- SSE3 (2004, Pentium 4E Prescott)
  - *Scientific computing*
  - New 2-way and 4-way vector instructions for complex arithmetic
- SSSE3 (2006, Core Duo)
  - Minor advancement over SSE3
- SSE4 (2007, Core2 Duo Penryn)
  - *Modern codecs, cryptography*
  - New integer instructions
  - Better support for unaligned data, super shuffle engine

More details at http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions

Overview Floating-Point Vector ISAs

Vendor	Name		ν-way	Precision	Introduced with
Intel	SSE		4-way	single	Pentium III
	SSE2	+	2-way	double	Pentium 4
	SSE3				Pentium 4 (Prescott)
	SSSE3				Core Duo
	SSE4				Core2 Extreme (Penryn)
Intel	IPF		2-way	single	Itanium
AMD	3DNow!		2-way	single	K6
	Enhanced 3DNow!				K7
	3DNow! Professional	+	4-way	single	Athlon XP
	AMD64	+	2-way	double	Opteron
Motorola	AltiVec		4-way	single	MPC 7400 G4
IBM	VMX		4-way	single	PowerPC 970 G5
	SPU	+	2-way	double	Cell BE
IBM	Double FPU		2-way	double	PowerPC 440 FP2

Within a extension family, newer generations add features to older ones

Convergence: 3DNow! Professional = 3DNow! + SSE; VMX = AltiVec; SPU$\supset$VMX

Related Technologies

- Original SIMD machines (CM-2,…)
  - Don’t really have anything in common with SIMD vector extension
- Vector Computers (NEC SX6, Earth simulator)
  - Vector lengths of up to 128
  - High bandwidth memory, no memory hierarchy
  - Pipelined vector operations
  - Support strided memory access
- Very long instruction word (VLIW) architectures (Itanium,…)
  - Explicit parallelism
  - More flexible
  - No data reorganization necessary
- Superscalar processors (x86, …)
  - No explicit parallelism
  - Memory hierarchy

SIMD vector extensions borrow multiple concepts

How to use SIMD Vector Extensions?

- Prerequisite: fine grain parallelism
- Helpful: regular algorithm structure
- Easiest way: use existing libraries
  Intel MKL and IPP, Apple vDSP, AMD ACML,
  Atlas, FFTW, Spiral
- Do it yourself:
  - Use compiler vectorization: write vectorizable code
  - Use language extensions to explicitly issue the instructions
    Vector data types and intrinsic/builtin functions
    Intel C++ compiler, GNU C compiler, IBM VisualAge for BG/L,…
  - Implement kernels using assembly (inline or coding of full modules)

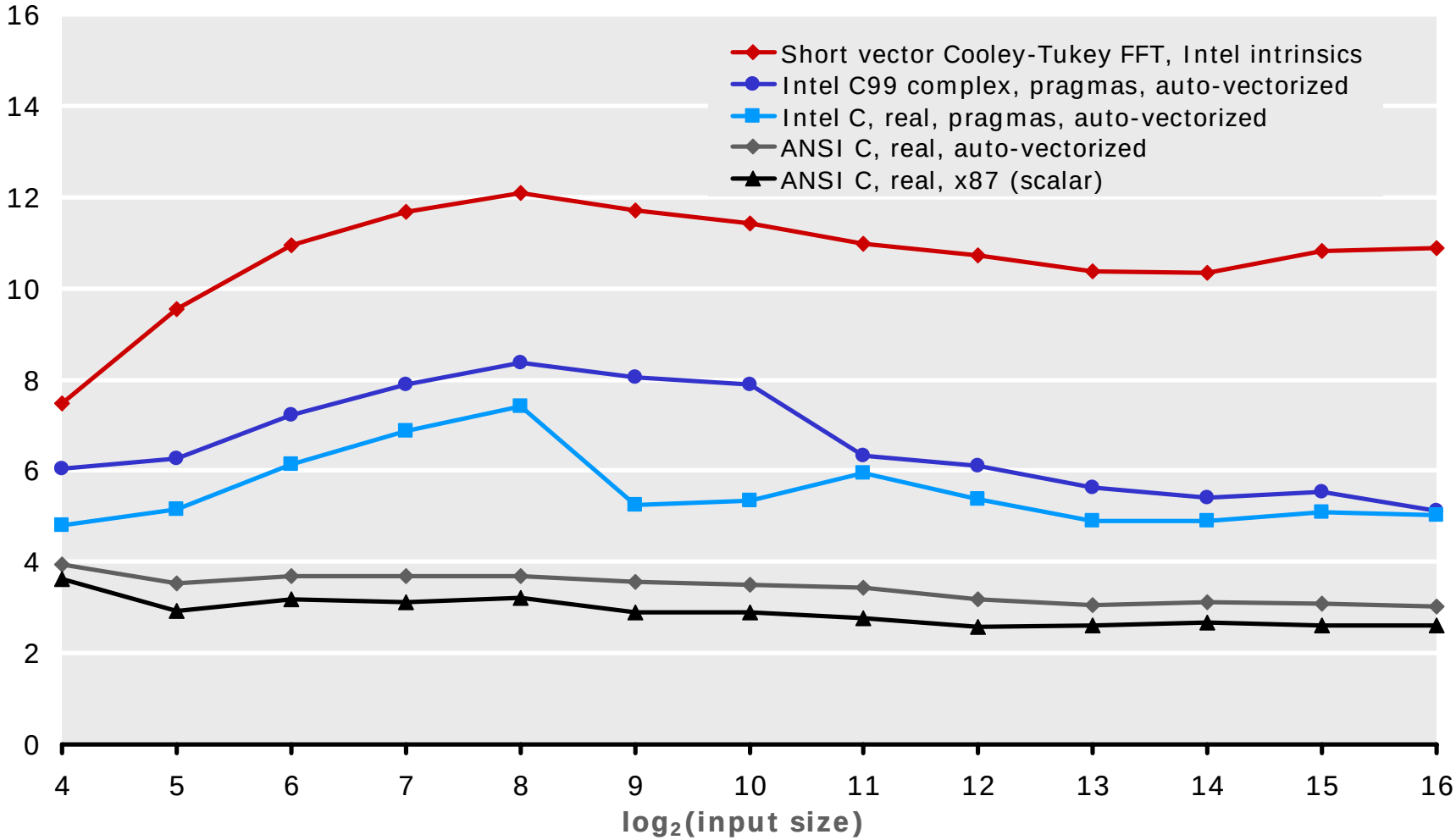
Characterization of Available Methods

- Interface used
  - Portable high-level language (possibly with pragmas)
  - Proprietary language extension (builtin functions and data types)
  - C++ Class interface
  - Assembly language
- Who vectorizes
  - Programmer or code generator expresses parallelism
  - Vectorizing compiler extracts parallelism
- Structures vectorized
  - Vectorization of independent loops
  - Instruction-level parallelism extraction
- Generality of approach
  - General purpose (e.g., for complex code or for loops)
  - Problem specific (for FFTs or for matrix products)

Electrical & Computer ENGINEERING

Spiral-generated FFT on 2.66 GHz Core2 (4-way SSE)

performance [Gflop/s], single-precision, Intel C++ 9.1, SSSE, Windows XP 32-bit



- limitations of compiler vectorization
- C99 **_Complex** and **#pragma** help, but still slower than hand-vectorized code

Problems

- Correct data alignment paramount
- Reordering data kills runtime
- Algorithms must be adapted to suit machine needs
- Adaptation and optimization is machine/extension dependent
- Thorough understanding of ISA + Micro architecture required

One can easily slow down a program by vectorizing it

Organization

- Overview
  - Idea, benefits, reasons, restrictions
  - History and state-of-the-art floating-point SIMD extensions
  - How to use it: compiler vectorization, class library, intrinsics, inline assembly
- Writing code for Intel's SSE
  - Compiler vectorization
  - Intrinsics: instructions
  - Intrinsics: common building blocks
- Selected topics
  - SSE integer instructions
  - Other SIMD extensions: AltiVec/VMX, Cell SPU
- Conclusion: How to write good vector code

Intel Streaming SIMD Extension (SSE)

- Instruction classes
  - Memory access (explicit and implicit)
  - Basic arithmetic (+, -, *)
  - Expensive arithmetic (1/x, sqrt(x), min, max, /, 1/sqrt)
  - Logic (and, or, xor, nand)
  - Comparison (+, <, >, …)
  - Data reorder (shuffling)
- Data types
  - float: __m128 (SSE)
  - double: __m128d (SSE2)
  - Integer: __m128i (8 bit – 128 bit)
- Intel C++ Compiler Manual
  http://www.intel.com/cd/software/products/asmo-na/eng/347618.htm
  http://www.intel.com/cd/software/products/asmo-na/eng/346158.htm
  http://msdn2.microsoft.com/en-us/library/26td21ds.aspx

Intel C++ Compiler: Automatic Vectorization

- Example program: pointwise vector multiplication

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

- Compiler invocation

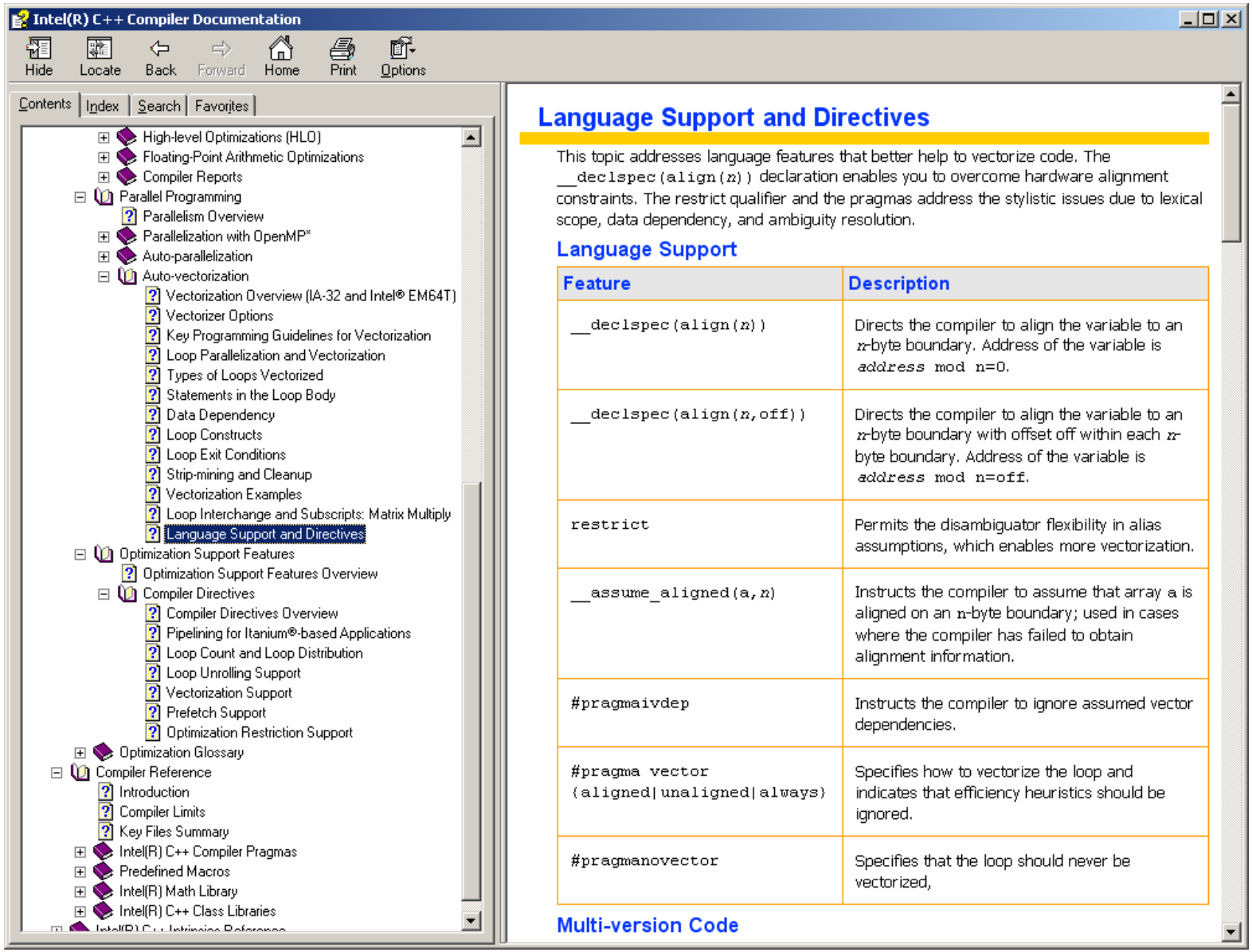
```
C:\>iclvars > NUL

C:\>C>icl /Qc99 /Qrestrict /O3 /QxW /Qvec-report3 /FAs /c
  test.c

Intel(R) C++ Compiler for 32-bit applications, Version 9.1
Copyright (C) 1985-2006 Intel Corporation.  All rights
  reserved.

test.c
test.c(2) : (col. 5) remark: LOOP WAS VECTORIZED.
```

Intel C++ Compiler: Auto Vectorizer Manual



Intel C++ Compiler: Options and Output

- On Windows Intel C++ compiler requires VisualStudio
- On command line **`iclvars.cmd`** initializes the environment
- Compiler Options
  - C99 syntax: /Qc99 /Qrestrict
  - Full optimization: /O3
  - Vectorization target: SSE2 /QxW
    other targets: /QxK (SSE) , /QxP (SSE3), /QxT (SSSE), /QxS (SSE4)
  - Vectorization report: /Qvec-report3
  - Assembly output (source + assembly): /FAs
- Check vectorization quality: Checking output assembly

```
$B1$17:                         ; Preds $B1$17 $B1$16
        movups    xmm0, XMMWORD PTR [ecx+edi*4]                 ;3.16
        mulps     xmm0, XMMWORD PTR [edx+edi*4]                 ;3.23
        movaps    XMMWORD PTR [esi+edi*4], xmm0                 ;3.9
        movups    xmm1, XMMWORD PTR [ecx+edi*4+16]              ;3.16
        mulps     xmm1, XMMWORD PTR [edx+edi*4+16]              ;3.23
        movaps    XMMWORD PTR [esi+edi*4+16], xmm1              ;3.9
        add       edi, 8                                        ;2.5
```

Intel C++ Compiler: Language Extension

- Language extension
  - C99 "**`restrict`**" keyword
  - Aligned C library functions: **`_mm_malloc()`**, **`_mm_free()`**
  - **`_assume_aligned()`**
  - **`__declspec(__align())`**
  - Pragmas

```
#pragma vector aligned | unaligned | always
#pragma ivdep
#pragma novector
```

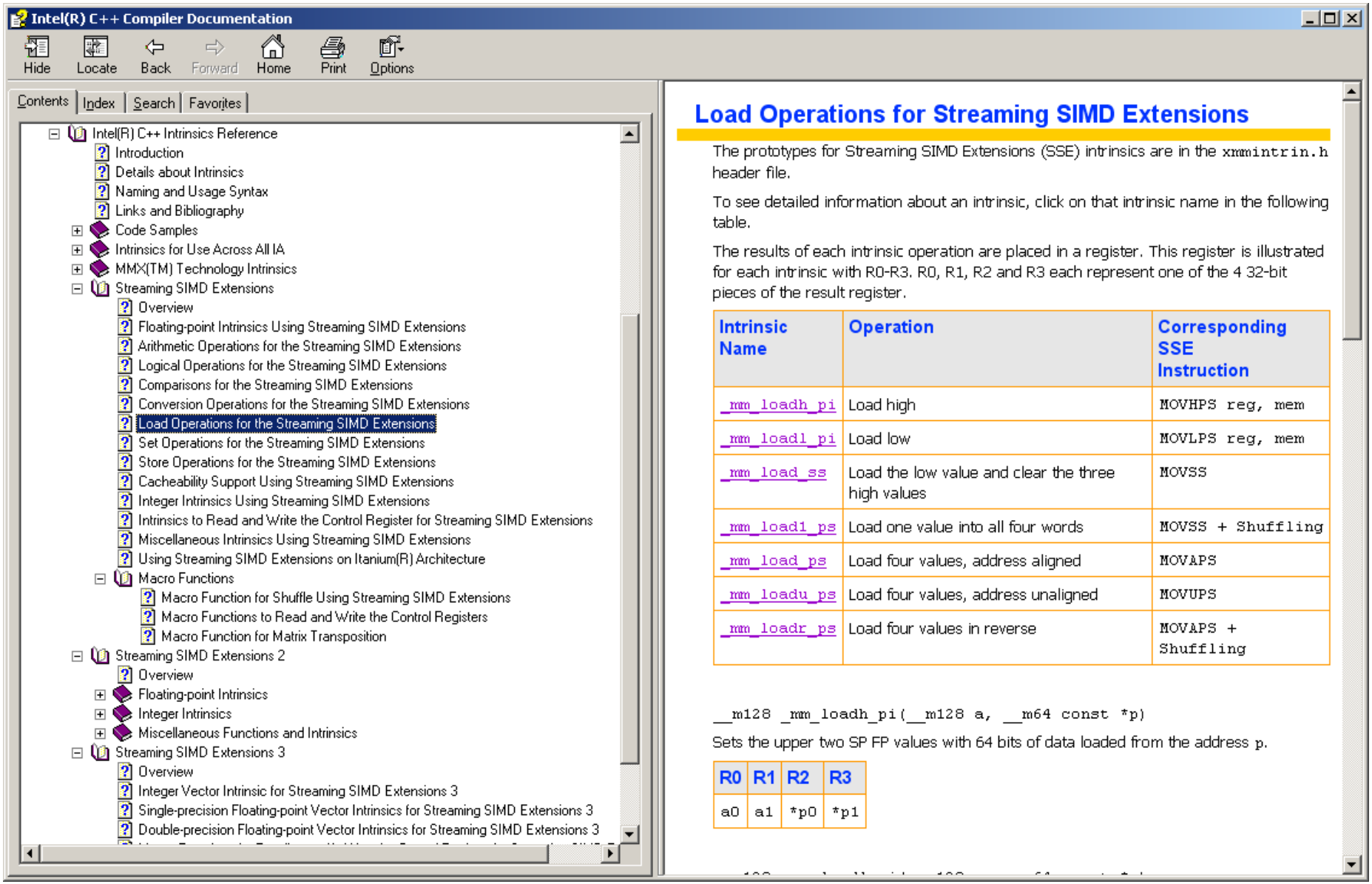
- Example using language extension

```
void func(float *restrict c, float *restrict a,
          float *restrict b, int n) {
#pragma vector always
    for (int i=0; i<n; i++)
        c[i] = a[i] * b[i];
}
```

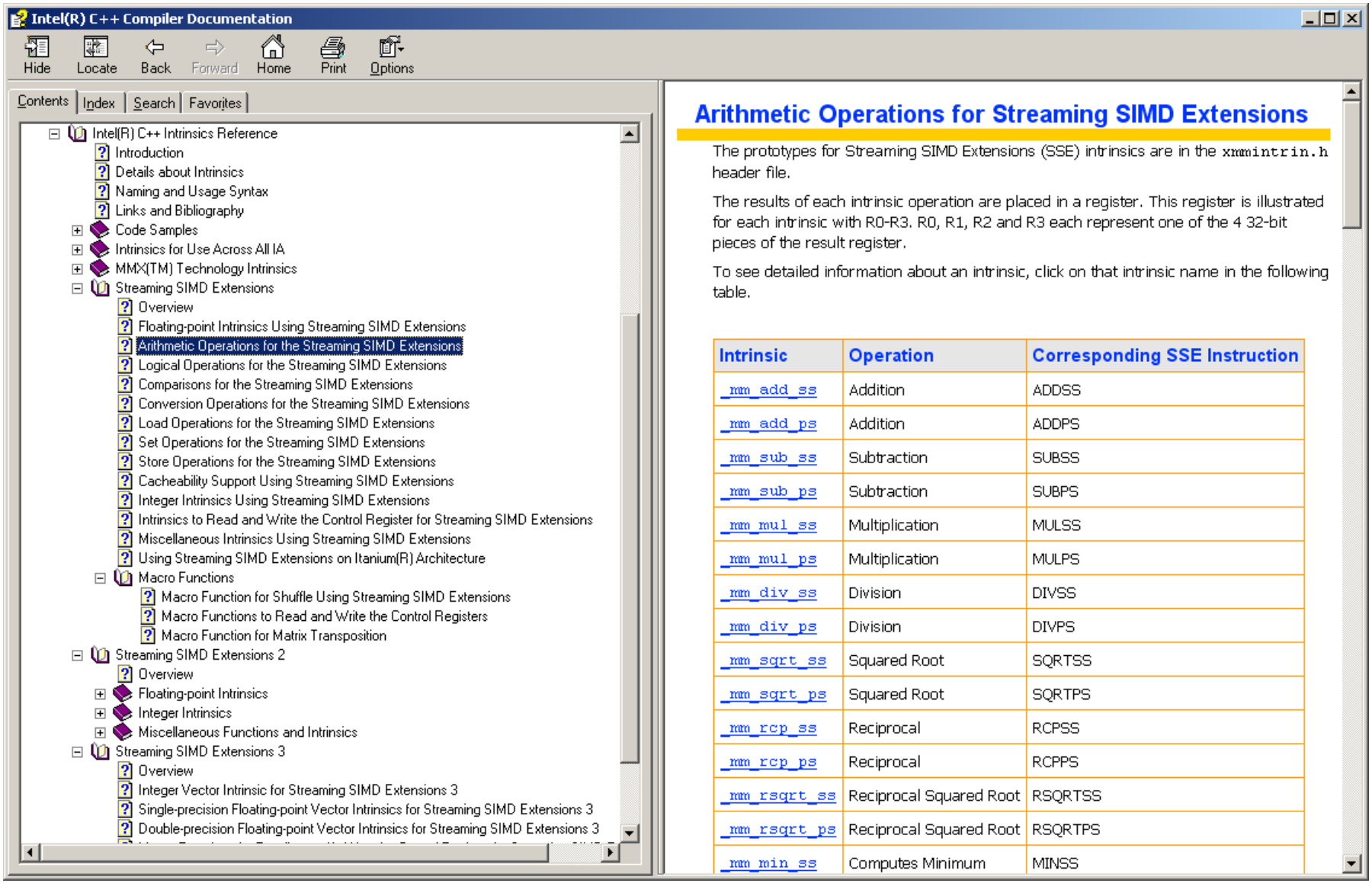
Intel SSE Intrinsics Interface

- Data types
  - `__m128 f; // ={float f3, f2, f1, f0}`
  - `__m128d d; // ={double d1, d0}`
- Intrinsics
  - Native instructions: **`_mm_add_ps()`**, **`_mm_mul_ps()`**,…
  - Multi-instruction: **`_mm_setr_ps()`**, **`_mm_set1_ps`**, …
- Macros
  - Transpose: **`_MM_TRANSPOSE4_PS()`**, …
  - Helper: **`_MM_SHUFFLE()`**

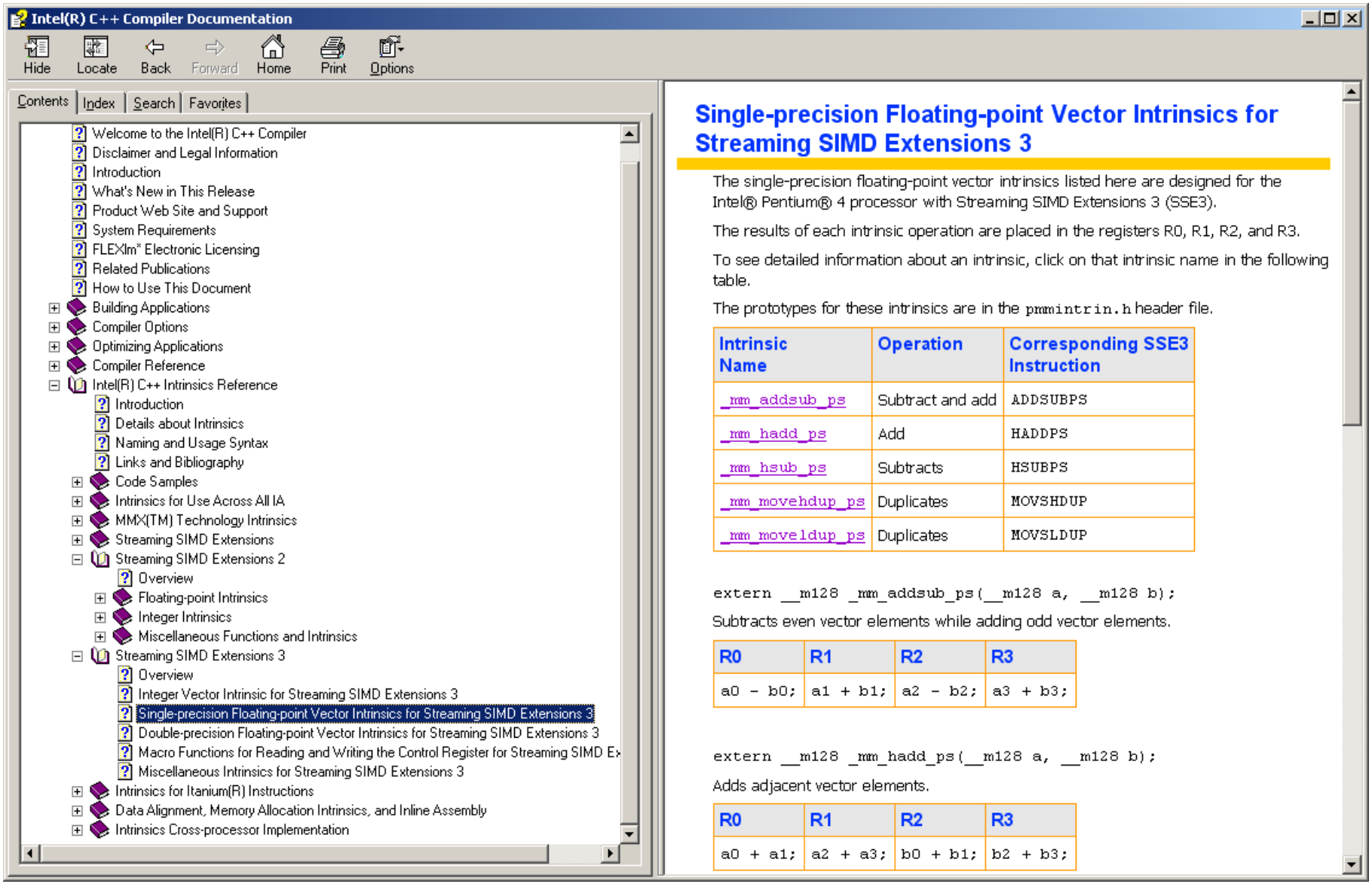
Intel SSE: Load Instructions



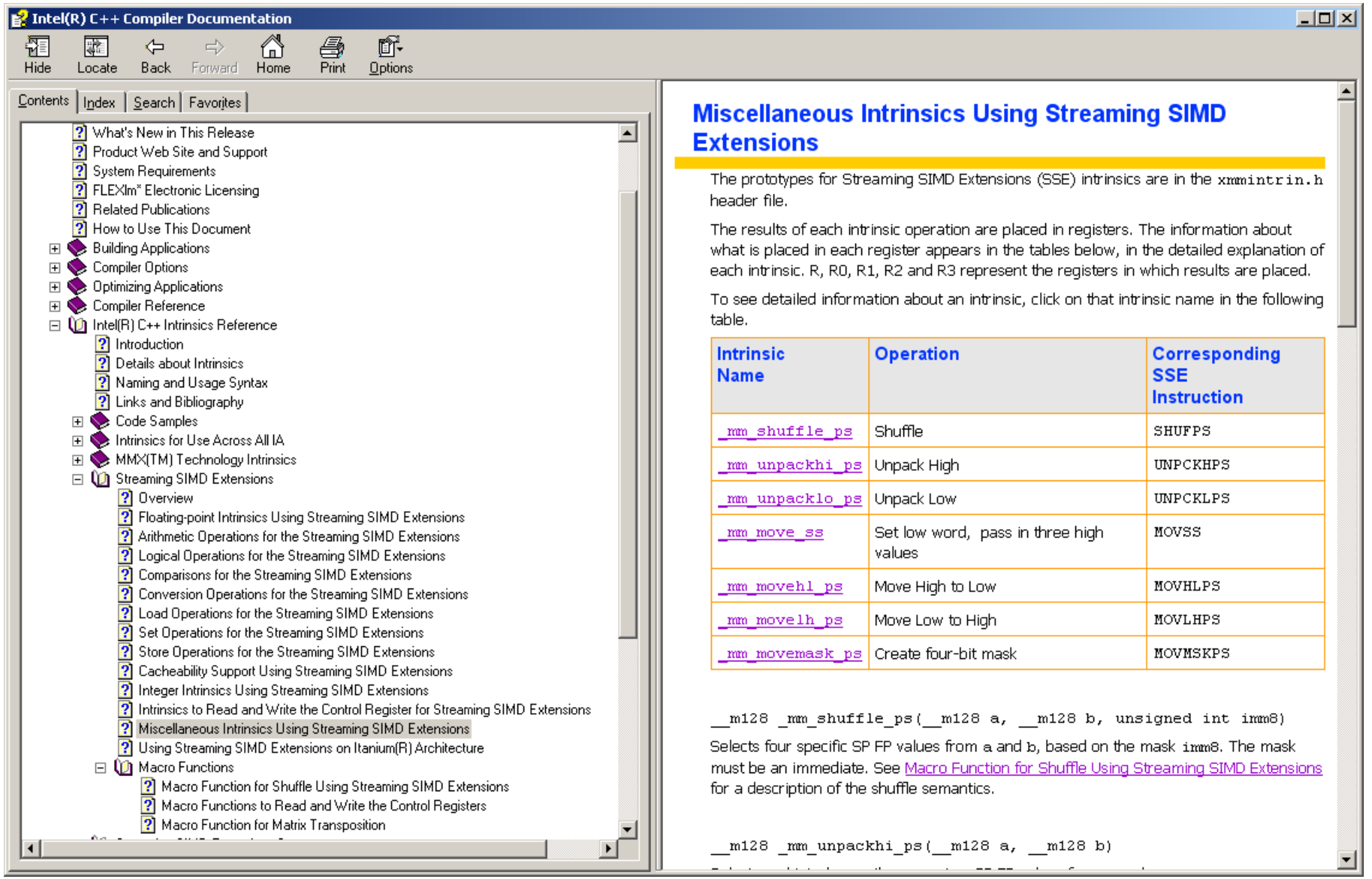
Intel SSE: Vector Arithmetic



Intel SSE: SSE3 Horizontal Add and SUB



Intel SSE: Reorder Instructions



Organization

- Overview
  - Idea, benefits, reasons, restrictions
  - History and state-of-the-art floating-point SIMD extensions
  - How to use it: compiler vectorization, class library, intrinsics, inline assembly
- Writing code for Intel's SSE
  - Compiler vectorization
  - Intrinsics: instructions
  - Intrinsics: common building blocks
- Selected topics
  - SSE integer instructions
  - Other SIMD extensions: AltiVec/VMX, Cell SPU
- Conclusion: How to write good vector code

Intel SSE: Transpose Macro

Intel(R) C++ Compiler Documentation

Hide | Locate | Back | Forward | Home | Print | Options

Contents | Index | Search | Favorites

- Related Publications
- How to Use This Document
- Building Applications
- Compiler Options
- Optimizing Applications
- Compiler Reference
- Intel(R) C++ Intrinsics Reference
  - Introduction
  - Details about Intrinsics
  - Naming and Usage Syntax
  - Links and Bibliography
  - Code Samples
  - Intrinsics for Use Across All IA
  - MMX(TM) Technology Intrinsics
  - Streaming SIMD Extensions
    - Overview
    - Floating-point Intrinsics Using Streaming SIMD Extensions
    - Arithmetic Operations for the Streaming SIMD Extensions
    - Logical Operations for the Streaming SIMD Extensions
    - Comparisons for the Streaming SIMD Extensions
    - Conversion Operations for the Streaming SIMD Extensions
    - Load Operations for the Streaming SIMD Extensions
    - Set Operations for the Streaming SIMD Extensions
    - Store Operations for the Streaming SIMD Extensions
    - Cacheability Support Using Streaming SIMD Extensions
    - Integer Intrinsics Using Streaming SIMD Extensions
    - Intrinsics to Read and Write the Control Register for Streaming SIMD Extensions
    - Miscellaneous Intrinsics Using Streaming SIMD Extensions
    - Using Streaming SIMD Extensions on Itanium(R) Architecture
    - Macro Functions
      - Macro Function for Shuffle Using Streaming SIMD Extensions
      - Macro Functions to Read and Write the Control Registers
      - **Macro Function for Matrix Transposition**
  - Streaming SIMD Extensions 2
  - Streaming SIMD Extensions 3
  - Intrinsics for Itanium(R) Instructions
  - Data Alignment, Memory Allocation Intrinsics, and Inline Assembly
  - Intrinsics Cross-processor Implementation

Macro Function for Matrix Transposition

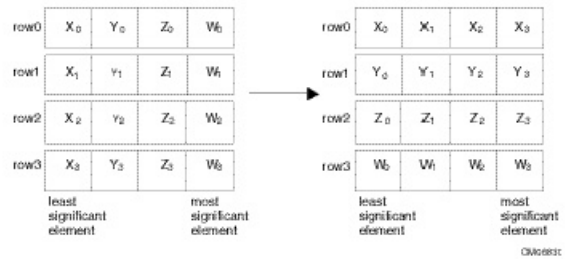
The Streaming SIMD Extensions (SSE) provide the following macro function to transpose a 4 by 4 matrix of single precision floating point values.

```
_MM_TRANSPOSE4_PS(row0, row1, row2, row3)
```

The arguments `row0`, `row1`, `row2`, and `row3` are `__m128` values whose elements form the corresponding rows of a 4 by 4 matrix. The matrix transposition is returned in arguments `row0`, `row1`, `row2`, and `row3` where `row0` now holds column 0 of the original matrix, `row1` now holds column 1 of the original matrix, and so on.

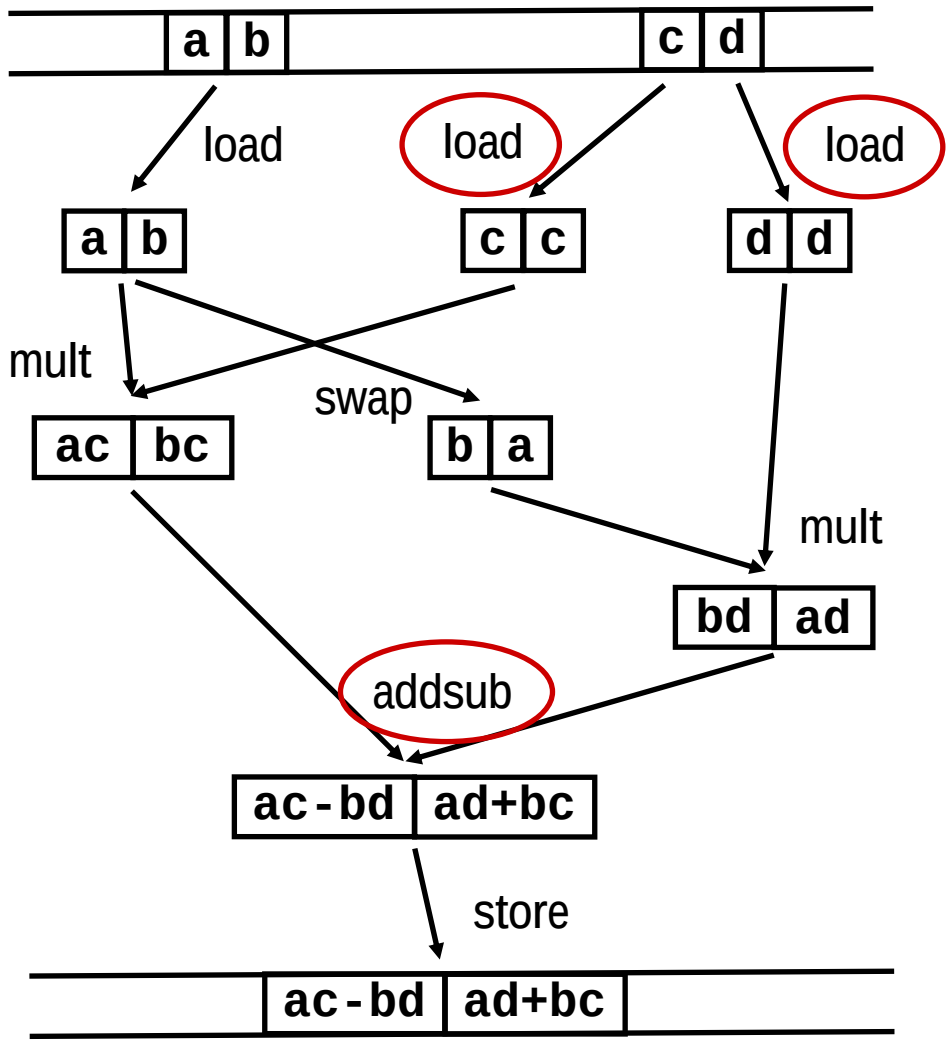
The transposition function of this macro is illustrated in the "Matrix Transposition Using `_MM_TRANSPOSE4_PS`" figure.

Matrix Transposition Using _MM_TRANSPOSE4_PS Macro



Example: Complex Multiplication SSE3

$$(a + ib)(c + id) = (ac - bd) + i(ad + bc)$$



Memory

Result:
4 load/stores
3 arithm. ops.
1 reorder op

Not available in SSE2

Memory

Looking a t the Assembly

SSE3:

```
movapd   xmm0, XMMWORD PTR A
movddup  xmm2, QWORD PTR   B
mulpd    xmm2, xmm0
movddup  xmm1, QWORD PTR   B+8
shufpd   xmm0, xmm0, 1
mulpd    xmm1, xmm0
addsubpd xmm2, xmm1
movapd   XMMWORD PTR C, xmm2
```

SSE2:

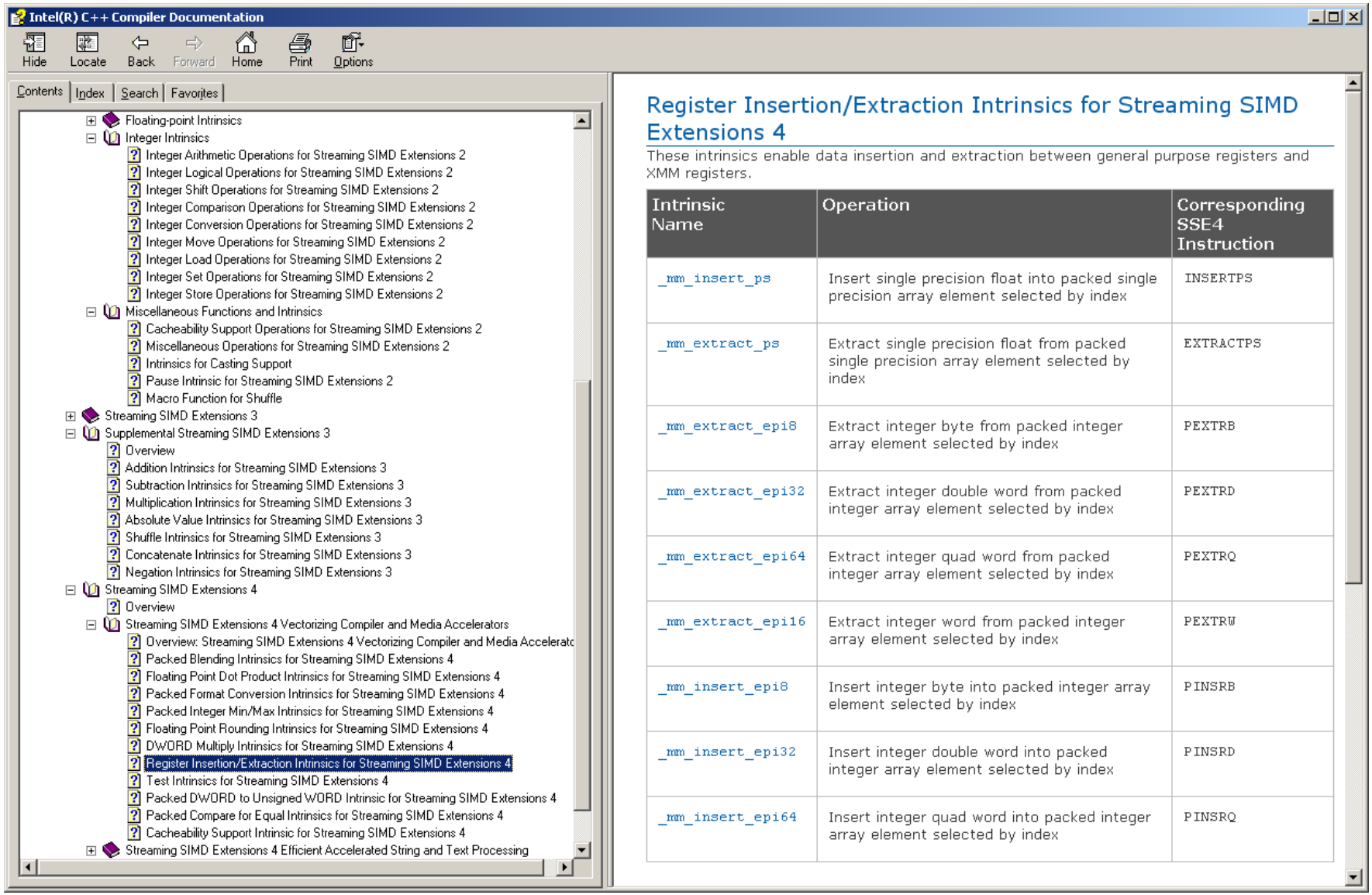
```
movsd    xmm3, QWORD PTR A
movapd   xmm4, xmm3
movsd    xmm5, QWORD PTR A+8
movapd   xmm0, xmm5
movsd    xmm1, QWORD PTR B
mulsd    xmm4, xmm1
mulsd    xmm5, xmm1
movsd    xmm2, QWORD PTR B+8
mulsd    xmm0, xmm2
mulsd    xmm3, xmm2
subsd    xmm4, xmm0
movsd    QWORD PTR C, xmm4
addsd    xmm5, xmm3
movsd    QWORD PTR C, xmm5
```

In SSE2 Intel C++ generates *scalar* code (better?)

Organization

- Overview
  - Idea, benefits, reasons, restrictions
  - History and state-of-the-art floating-point SIMD extensions
  - How to use it: compiler vectorization, class library, intrinsics, inline assembly
- Writing code for Intel's SSE
  - Compiler vectorization
  - Intrinsics: instructions
  - Intrinsics: common building blocks
- Selected topics
  - SSE integer instructions
  - Other SIMD extensions: AltiVec/VMX, Cell SPU
- Conclusion: How to write good vector code

Intel SSE: Integer Modes



Register Insertion/Extraction Intrinsics for Streaming SIMD Extensions 4

These intrinsics enable data insertion and extraction between general purpose registers and XMM registers.

Intrinsic Name	Operation	Corresponding SSE4 Instruction
_mm_insert_ps	Insert single precision float into packed single precision array element selected by index	INSERTPS
_mm_extract_ps	Extract single precision float from packed single precision array element selected by index	EXTRACTPS
_mm_extract_epi8	Extract integer byte from packed integer array element selected by index	PEXTRB
_mm_extract_epi32	Extract integer double word from packed integer array element selected by index	PEXTRD
_mm_extract_epi64	Extract integer quad word from packed integer array element selected by index	PEXTRQ
_mm_extract_epi16	Extract integer word from packed integer array element selected by index	PEXTRW
_mm_insert_epi8	Insert integer byte into packed integer array element selected by index	PINSRB
_mm_insert_epi32	Insert integer double word into packed integer array element selected by index	PINSRD
_mm_insert_epi64	Insert integer quad word into packed integer array element selected by index	PINSRQ

SSE Integer Modes (1)

- SSE generations
  - Introduced with SSE2
  - Functionality extended drastically with SSSE3 and SSE4
- Modes
  - 1x128 bit, 2x64 bit, 4x32 bit 8x 16 bit, 16x8 bit
  - Signed and unsigned
  - Saturating and non-saturating
- Operations
  - Arithmetic, logic, and shift, mullo/hi
  - Compare, test; min, max, and average
  - Conversion from/to floating-point, across precisions
  - Load/store/set
  - Shuffle, insert, extract, blend

SSE Integer Modes (2)

- Interoperability
  - Integer operations can be used with floating-point data
  - Typecast support
- Problems
  - Only subset of operations available in each mode
  - Sometimes need to “build” operation yourself
  - Gathers and scatters even more expensive (8- and 16-way)

```
// right-shift for signed __int8 16-way
__forceinline __m128i _mm_srli_epi8(__m128i x, int sh) {
    __m128i signs = _mm_and_si128(x, _mm_set1_epi32(0x80808080));
    __m128i z = _mm_srli_epi16(x, 1);
    z = _mm_and_si128(z, _mm_set1_epi32(0x7f7f7f7f));
    return _mm_or_si128(z, signs);
}
```

Extending Floating-Point Functionality

- Sign change
  - No sign-change instruction for vector elements exist
  - Integer exclusive-or helps

```
// sign-change of second vector element
__forceinline __m128 _mm_chsgn2_ps(__m128 f) {
    return _castsi128_ps(_mm_xor_si128(
        _mm_castps_si128(f),
        _mm_castps_si128(_mm_set_ps(0.0,0.0,-0.0,0.0))));
}
```

- Align instruction
  - **`alignr`** only exists for signed 8-bit integer

```
// alignr 4-way float variant
__forceinline __m128 _mm_alignr_ps(__m128 f1, __m128 f2, int sh) {
    return _castsi128_ps(_mm_alignr_epi8(
        _mm_castps_si128(f1), _mm_castps_si128(f2), sh));
}
```

Organization

- Overview
  - Idea, benefits, reasons, restrictions
  - History and state-of-the-art floating-point SIMD extensions
  - How to use it: compiler vectorization, class library, intrinsics, inline assembly
- Writing code for Intel's SSE
  - Compiler vectorization
  - Intrinsics: instructions
  - Intrinsics: common building blocks
- Selected topics
  - SSE integer instructions
  - Other SIMD extensions: AltiVec/VMX, Cell SPU
- Conclusion: How to write good vector code

AltiVec, VMX, Cell BE PPU and SPU,...

- AltiVec: 4-way float, 4-, 8-, and 16-way integer
  - Introduced with Motorola MPC 7400 G4 (direct competitor to Intel SSE and Pentium III)
  - Gave big boost to Apple multi media applications
  - Still available in Freescale PowerPC processors
  - Supported by GNU C builtin functions (2.95, 3.X)
- AltiVec became IBM VMX
  - PowerPC 970 G5 (G4 successor) and POWER6
  - Cell BE PPU (PowerPC)
  - VMX128 version for Xbox 360 (Xenon processor)
- Cell SPU: closely aligned with VMX
  - Double-precision instructions (very slow at this point)

AltiVec vs. SSE

- AltiVec: PowerPC is 3-operand RISC
  - Fused multiply-add
  - Powerful general shuffle instruction
  - More registers (32 – 128)
- Problem: non-vector memory access
  - Unaligned load/store
  - Subvector load/store
- AltiVec/VMX is not changing as quickly as SSE
  - Variants: AltiVec/VMX, VMX128, SPU
  - AltiVec important in embedded computing
  - SSE is closer to the consumer market, permanently updated

Organization

- Overview
  - Idea, benefits, reasons, restrictions
  - History and state-of-the-art floating-point SIMD extensions
  - How to use it: compiler vectorization, class library, intrinsics, inline assembly
- Writing code for Intel's SSE
  - Compiler vectorization
  - Intrinsics: instructions
  - Intrinsics: common building blocks
- Selected topics
  - SSE integer instructions
  - Other SIMD extensions: AltiVec/VMX, Cell SPU
- Conclusion: How to write good vector code

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

Remaining time: Discussion