Programming Vector Extensions

1. SSE/AVX overview (CMU)
2. SSE/AVX Programming (Alex Fr)
3. SSE/AVX from C w/ GCC (B&O)

Part 1: SSE/AVX Basics

- 128-bit registers (8 of these on the i7)
- 256-bit registers on our machines
  - Flexible usage: **`2xDOUBLE or 4xFLOAT or any integer combination (e.g., 16xCHAR or 4xINT or 2xLONGINT)`**
  - Flexible usage: **`4xDOUBLE or 8xFLOAT or any integer combination (e.g., 32xCHAR or 8xINT or 4xLONGINT)`**
- 512-bit registers in latest high-end processors!

- ***Rich*** instruction set: *>> note correspondence with canonical SIMD*
  - Load/store vectors to/from memory
  - Element-wise vector-vector operations
  - Element-wise scalar-vector operations
  - Load constants into vector → broadcast to elements of vector
  - Reorder elements in vector: pack, unpack, shuffle, align, blend, etc.
  - Collective operations on elements of vector: reduction, dot product, broadcast
  - Mask enabled instructions (activity control/conditional move)

Organization

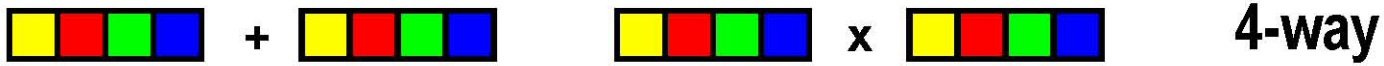
How to Write Fast Code
SIMD Vectorization
18-645, spring 2008
13th and 14th Lecture

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

- **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

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)

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

Intel Streaming SIMD Extensions (SSE, now AVX)

What are we looking at here? Hardware instructions. How do access comes later.

- Instruction Classes *>> again, canonical set of SIMD ops*

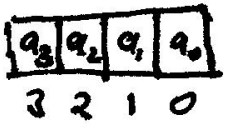
  → *Elementwise unless noted*

  - Memory access (explicit and implicit)
  - Basic arithmetic (+,-,*)
  - Expensive arithmetic (1/x,sqrt(x),/)
  - Reductions (min, max, sum of register elements)
  - Logic (and,or,xor,nand)
  - Comparison (==,<,>, …)
  - Reordering (shift, merge, permute within a register)

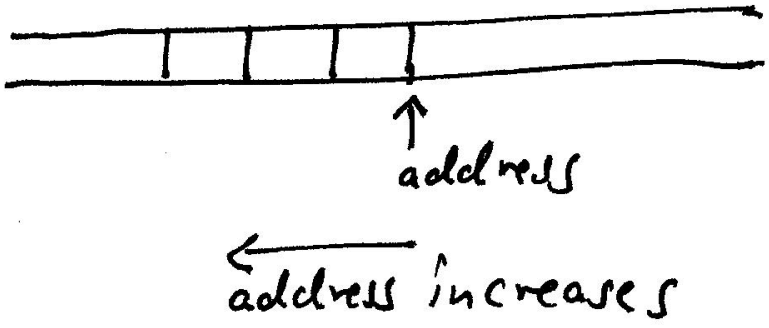
- Data Types *>> depending on CPU registers are 128-bit, 256-bit, or 512-bit*
  - Float: __m128, __m256, __m512
  - Double: __m128d, __m256d, __m512d
  - Integer: __m128i, __m256i, __m512i (8-bit – 128-bit)

AVX Basics – assume 128-bit

Information

vector indexing: a: 

in most intrinsics, the order of operands matter

memory: 

SIMD extensions timeline: SSE, SSE2, SSE3, SSSE3, SSE4

We focus on single precision float, 4-way
1 vector = 128 bit = 16 B, data type --m128

Unless stated otherwise, instructions are SSE or later

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()`

p = “packed” i.e., use AVX
s = “scalar” i.e., use only low-order value as a normal register

s = float (single precision)
d = double
i = integer

Intel SSE: Load Instructions

Intel(R) C++ Compiler Documentation

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

Contents | Index | Search | Favorites

- 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
    - Overview
    - Floating-point Intrinsics
    - Integer Intrinsics
    - Miscellaneous Functions and Intrinsics
  - Streaming SIMD Extensions 3
    - Overview
    - Integer Vector Intrinsic for Streaming SIMD Extensions 3
    - Single-precision Floating-point Vector Intrinsics for Streaming SIMD Extensions 3
    - Double-precision Floating-point Vector Intrinsics for Streaming SIMD Extensions 3

Load Operations for Streaming SIMD Extensions

The prototypes for Streaming SIMD Extensions (SSE) intrinsics are in the `xmmintrin.h` header file.

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

The results of each intrinsic operation are placed in a register. This register is illustrated for each intrinsic with R0-R3. R0, R1, R2 and R3 each represent one of the 4 32-bit pieces of the result register.

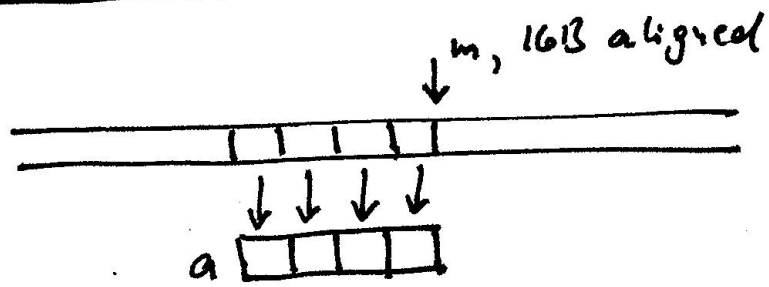
Intrinsic Name	Operation	Corresponding SSE Instruction
`_mm_loadh_pi`	Load high	`MOVHPS reg, mem`
`_mm_loadl_pi`	Load low	`MOVLPS reg, mem`
`_mm_load_ss`	Load the low value and clear the three high values	`MOVSS`
`_mm_load1_ps`	Load one value into all four words	`MOVSS + Shuffling`
`_mm_load_ps`	Load four values, address aligned	`MOVAPS`
`_mm_loadu_ps`	Load four values, address unaligned	`MOVUPS`
`_mm_loadr_ps`	Load four values in reverse	`MOVAPS + Shuffling`

```
__m128 _mm_loadh_pi(__m128 a, __m64 const *p)
```

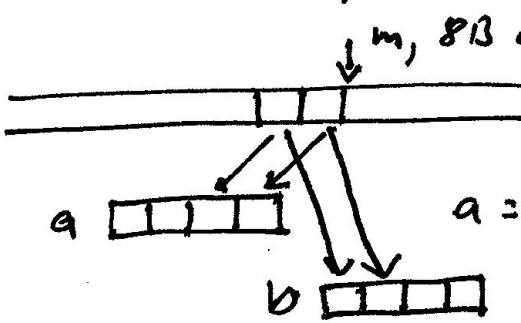
Sets the upper two SP FP values with 64 bits of data loaded from the address `p`.

R0	R1	R2	R3
a0	a1	*p0	*p1

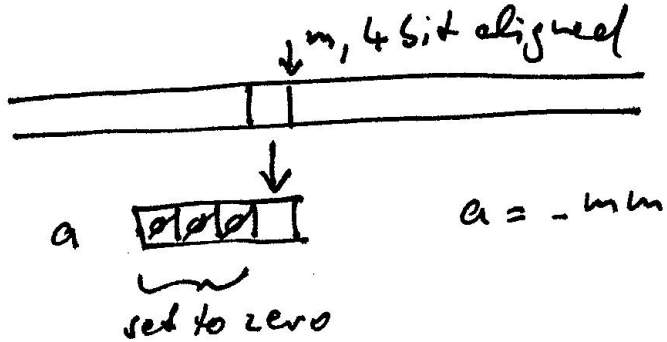
Load and Store




```
a = _mm_load_ps(m);     aligned
a = _mm_loadu_ps(m);    unaligned (avoid)
a = p[i];   if p is: __m128 *p
```

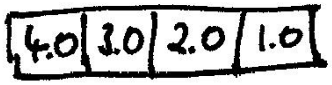



```
a = _mm_loadl_pi(a,m);   (keeps upper half)
b = _mm_loadh_pi(b,m);   (keeps lower half)
```

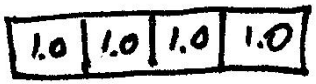



```
a = _mm_load_ss(m)
```

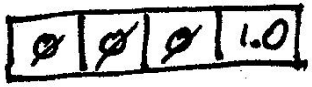
Constants

c: 


```
c = _mm_set_ps(4.0, 3.0, 2.0, 1.0);
```

d: 


```
d = _mm_set1_ps(1.0);
```

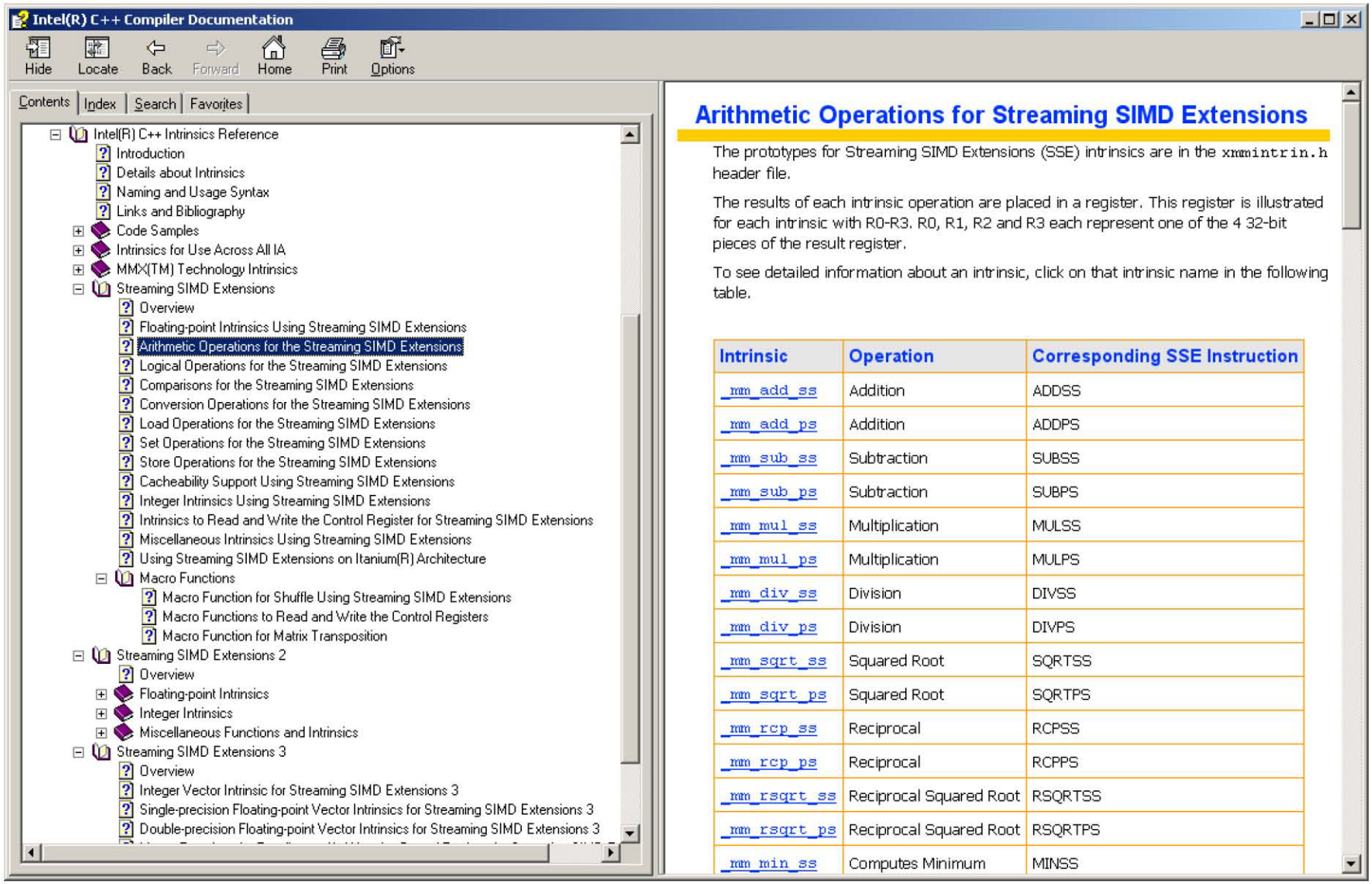
e: 


```
e = _mm_set_ss(1.0);
```

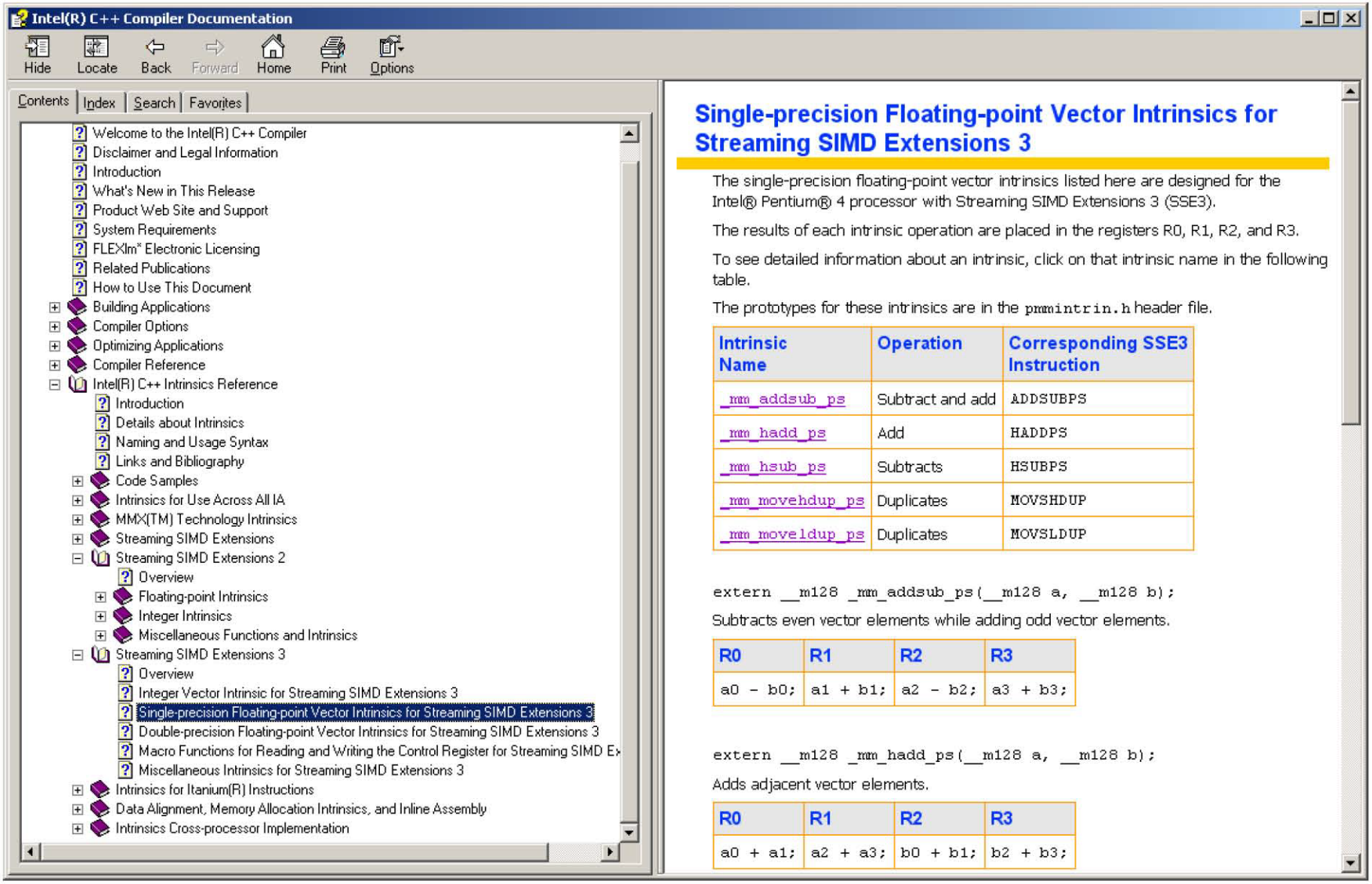
f: 

```
f = _mm_setzero_ps();
```

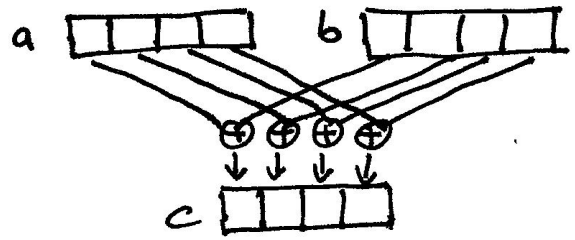
Intel SSE: Vector Arithmetic



Intel SSE: SSE3 Horizontal Add and SUB



Vector arithmetic

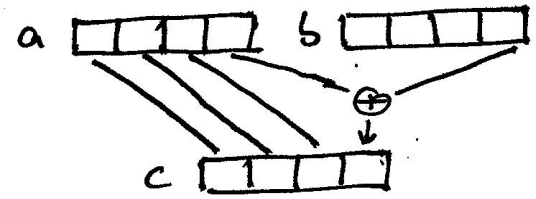



```
c = _mm_add_ps(a, b);      "a+b"
```

analogous:

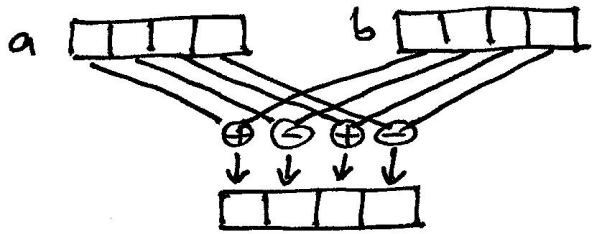
```
c = _mm_sub_ps(a, b);      "a-b"
c = _mm_mul_ps(a, b);      "a·b"
```

Scalar arithmetic

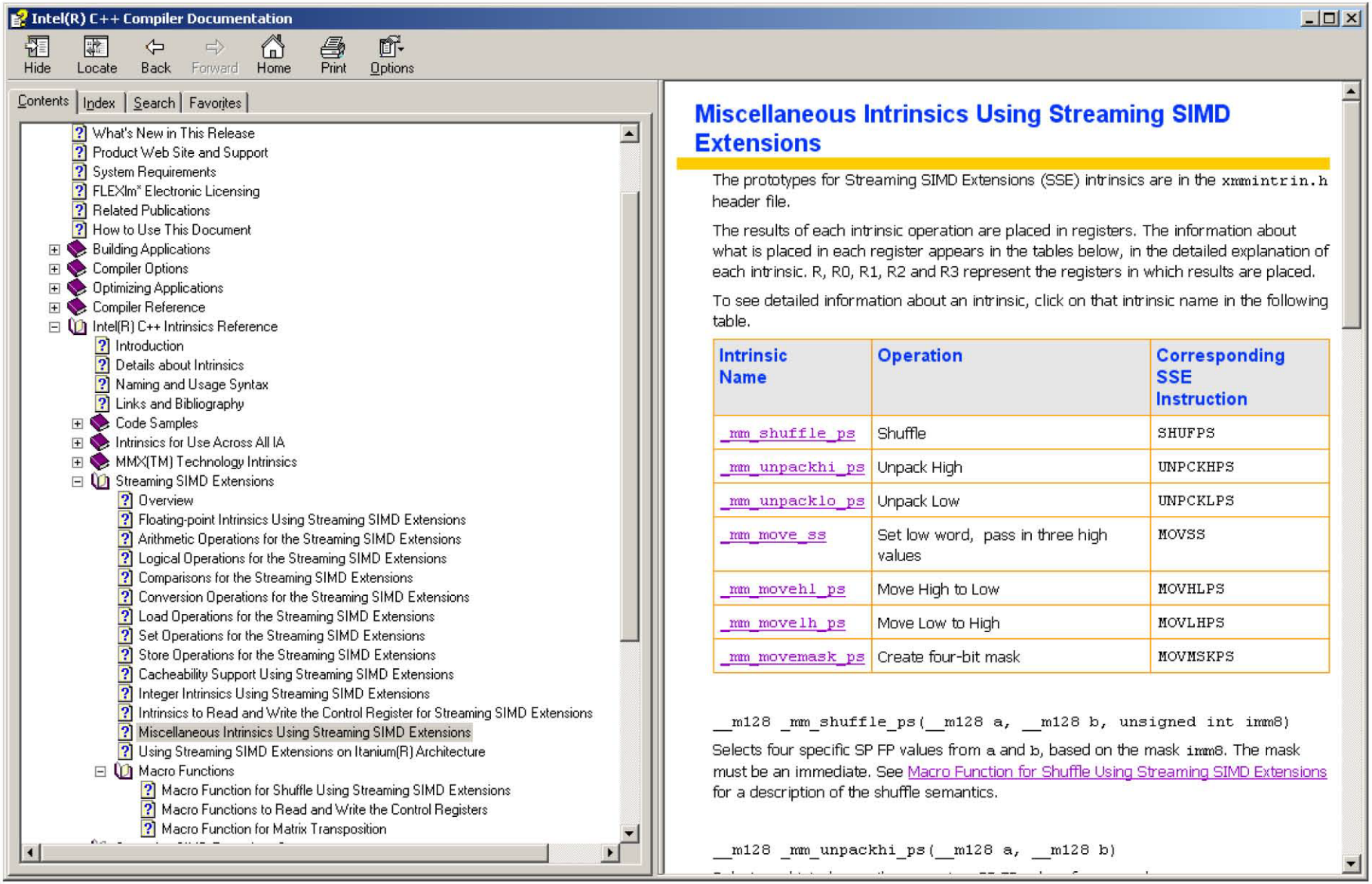



```
c = _mm_add_ss(a, b);
```

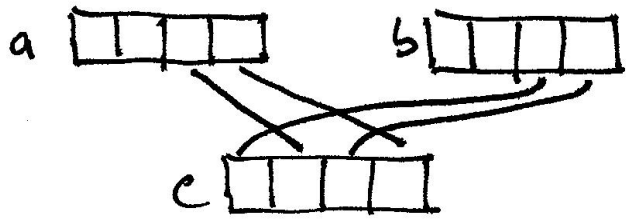
AddSub (SSE3 and later)



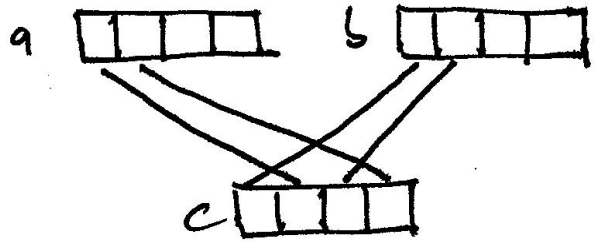
Intel SSE: Reorder Instructions



Reorder Instructions

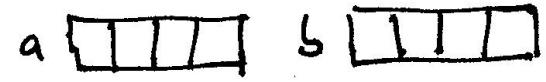


c = _mm_unpacklo_ps(a, b)



c = _mm_unpackhi_ps(a, b)

shuffle:



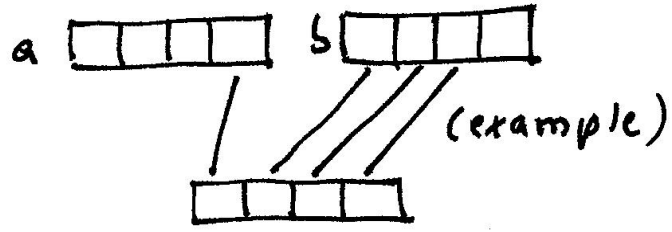
any element of b any element of a

c = _mm_shuffle_ps(a, b, _MM_SHUFFLE(l, k, j, i))

$c_0 = a_i$
$c_1 = a_j$
$c_2 = b_k$
$c_3 = b_l$

$i, j, k, l \in \{0 \ldots 3\}$

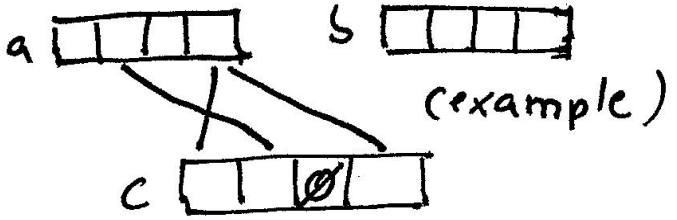
align: (SSE3 and later)



"any 4 consecutive elements of the concatenation of a and b go into c"

_mm_alignr_epi8 use with _mm_castsi128_ps

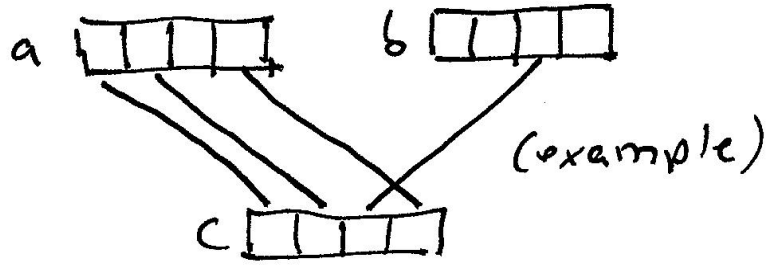
shuffle: (SSE3 and later)



"c is filled in each position with any element from a or 0, as specified by b"

_mm_shuffle_epi8

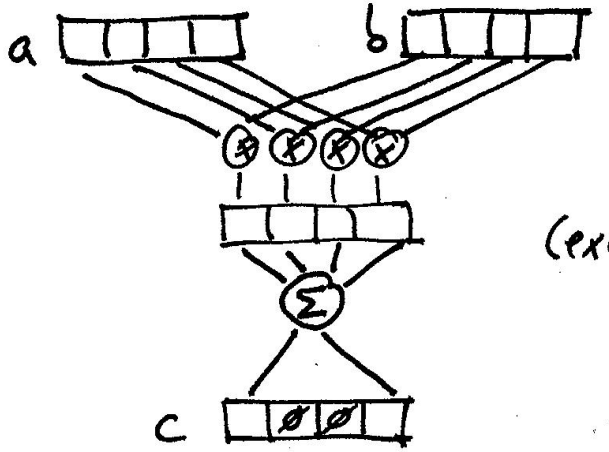
blend: (SSE4.1 and later)



"c is filled in each position with an element from a or b from the same position"

_mm_blend_ps

Dot product (SSE4 and later)

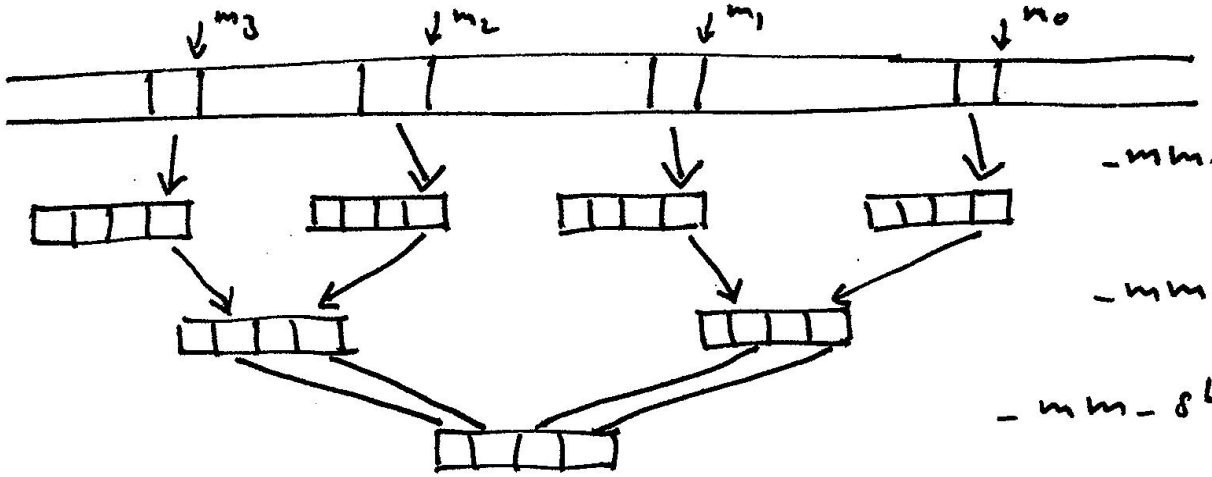


(example)

"computes the pointwise product of a and b and writes an arbitrary sum of the resulting numbers into selected elements of c — the others are set to zero"

_mm_dp_ps(a, b, mask)

Load 4 real numbers from arbitrary memory locations



7 instructions, this is the right way

Note:

- whenever possible avoid this by restructuring the algorithm or data to have aligned vector loads _mm_load_ps
- the above should be equivalent to the following but a.) the above is safer; b.) be aware that the below are 7 instructions

```
float f[20] = {... };
__m128 vf = _mm_set_ps(f[3], f[5], f[19], f[1]);
```

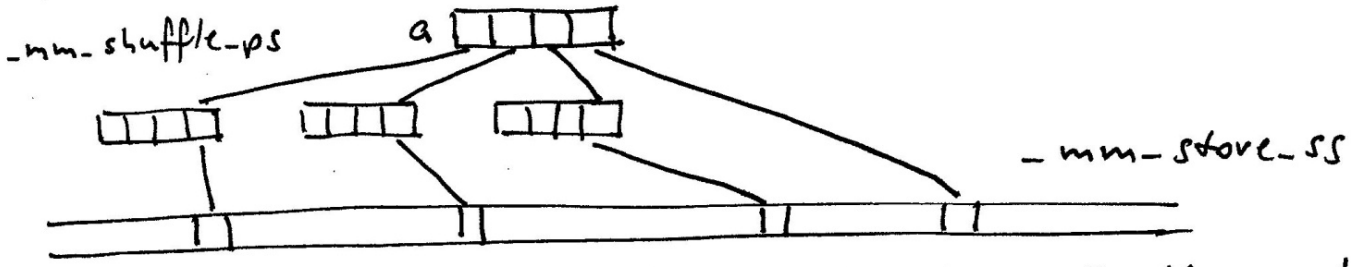
Don't do this:

```
float f[20] = {...};
__declspec(align(16)) g[4];
__m128 vf;
g[0] = f[1];
g[1] = f[17];
g[2] = f[5];
g[3] = f[3];
vf = _mm_load_ps(g);
```

mem → register → mem round trip ⇒ expensive

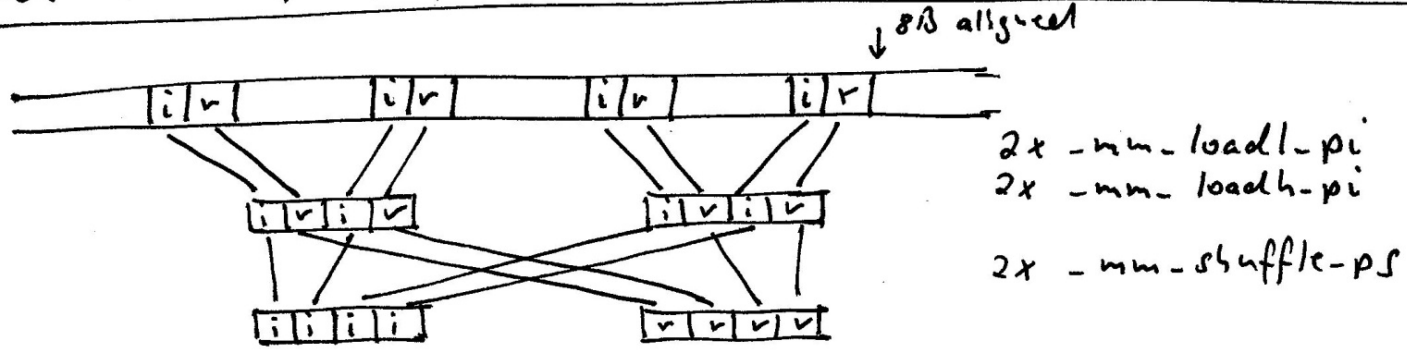
(same problem with unions and pointers)

Store 4 real numbers to arbitrary memory locations (



7 instructions, shorter critical path than load

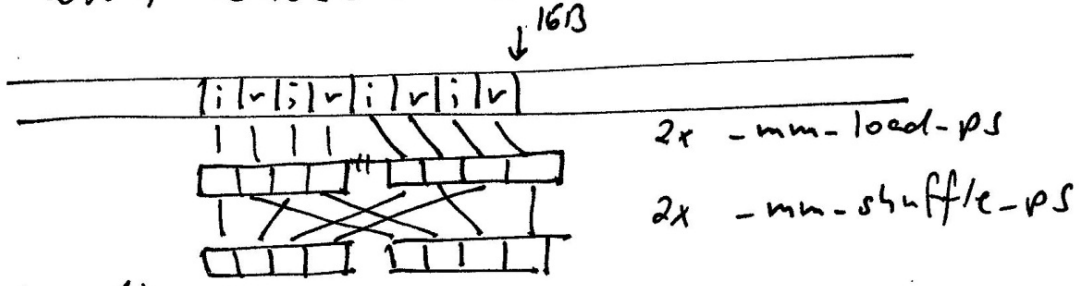
Load 4 complex numbers (= 4 pairs of real numbers)



6 instructions

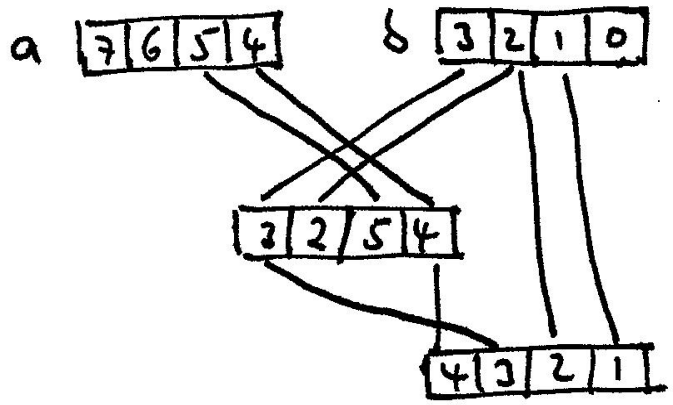
store analogous

same with consecutive data:



4 instructions

Shift by 1



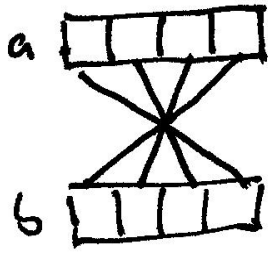
_mm_shuffle_ps

_mm_shuffle_ps

2 instructions

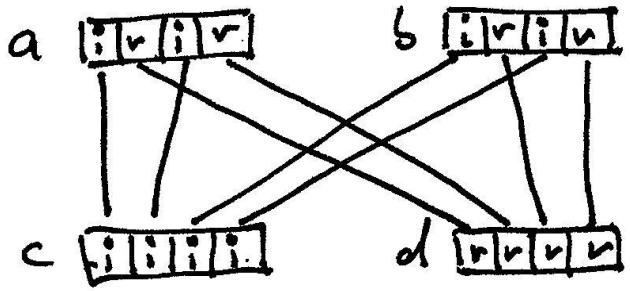
SSE3 and later: _mm_alignr_epi8 + casts 1 instruction

Reverse vector



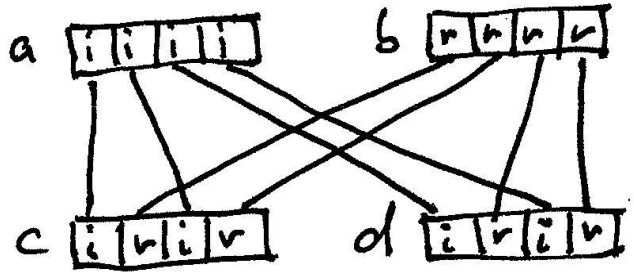

```
b = _mm_shuffle_ps(a, a, _MM_SHUFFLE(0,1,2,3));
```

Interleaved complex → split complex




```
c = _mm_shuffle_ps(b, a, _MM_SHUFFLE(3,1,3,1));
d = ..... _MM_SHUFFLE(2,0,2,0));
```

Split complex → interleaved complex




```
c = _mm_unpackhi_ps(b, a);
d = _mm_unpacklo_ps(b, a);
```

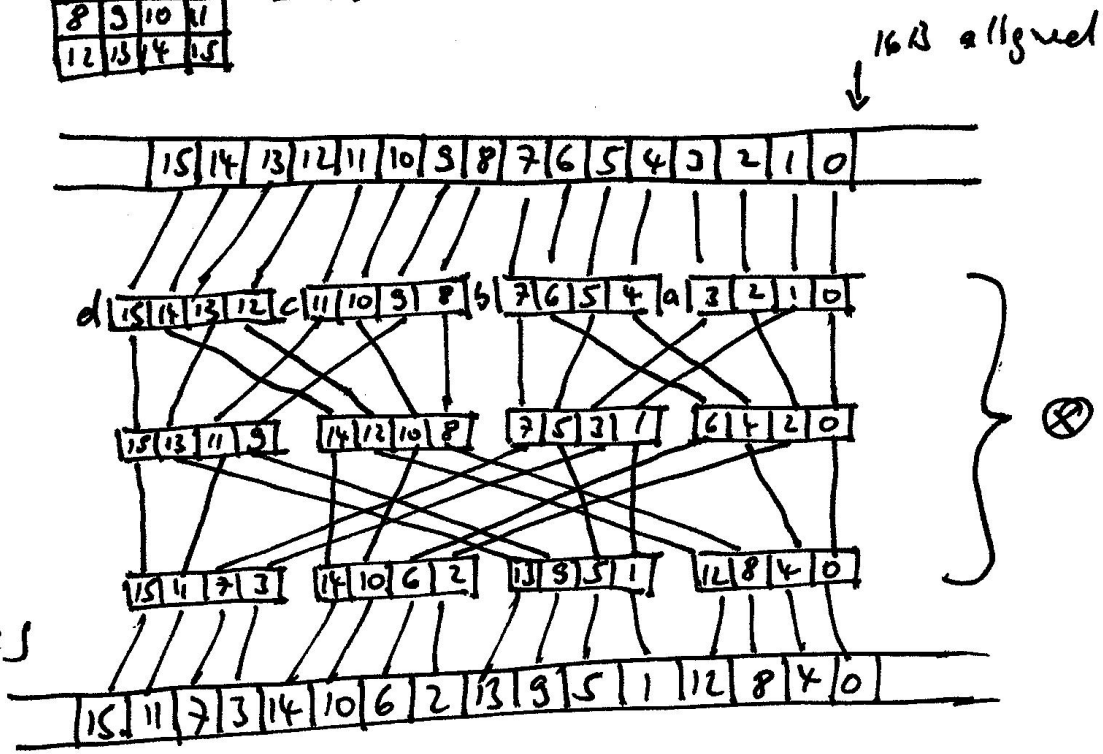
Transposition: 4×4 matrix

4×4 matrix:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

$= A$

in memory:



as matrix:

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

$= A^T$

⊗ done by the macro `_MM_TRANSPOSE4_PS(a,b,c,d);`

8 instructions

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

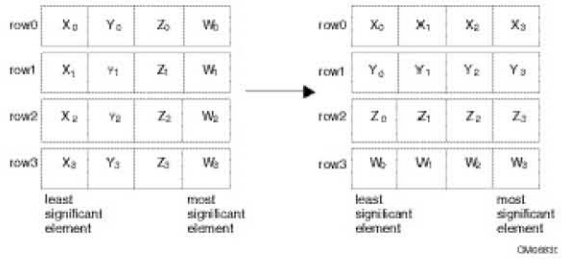
The Streaming SIMD Extensions (SSE) provide the following macro function to transpose a 4 by 4 matrix of single precision floating point values.

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

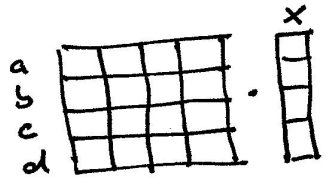
The arguments `row0`, `row1`, `row2`, and `row3` are `__m128` values whose elements form the corresponding rows of a 4 by 4 matrix. The matrix transposition is returned in arguments `row0`, `row1`, `row2`, and `row3` where `row0` now holds column 0 of the original matrix, `row1` now holds column 1 of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

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

Matrix Transposition Using _MM_TRANSPOSE4_PS Macro

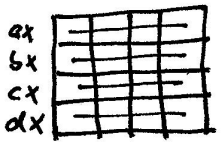


Matrix - vector product



1. step: 4 vector products ax, bx, cx, dx (4 instructions)

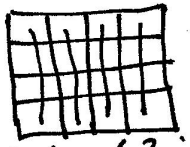
result:



SSE:

2. step: transpose (8 instructions)

result:

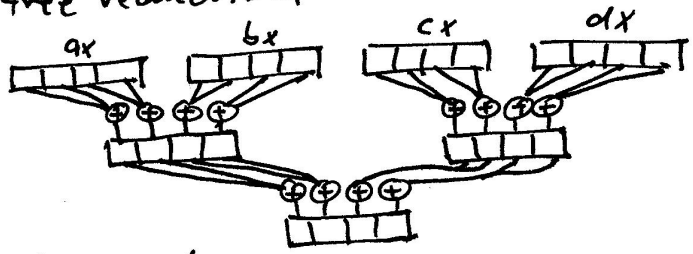


3. step: sum rows (3 instructions)

total: 15 instructions

SSE3:

2. step: tree reduction



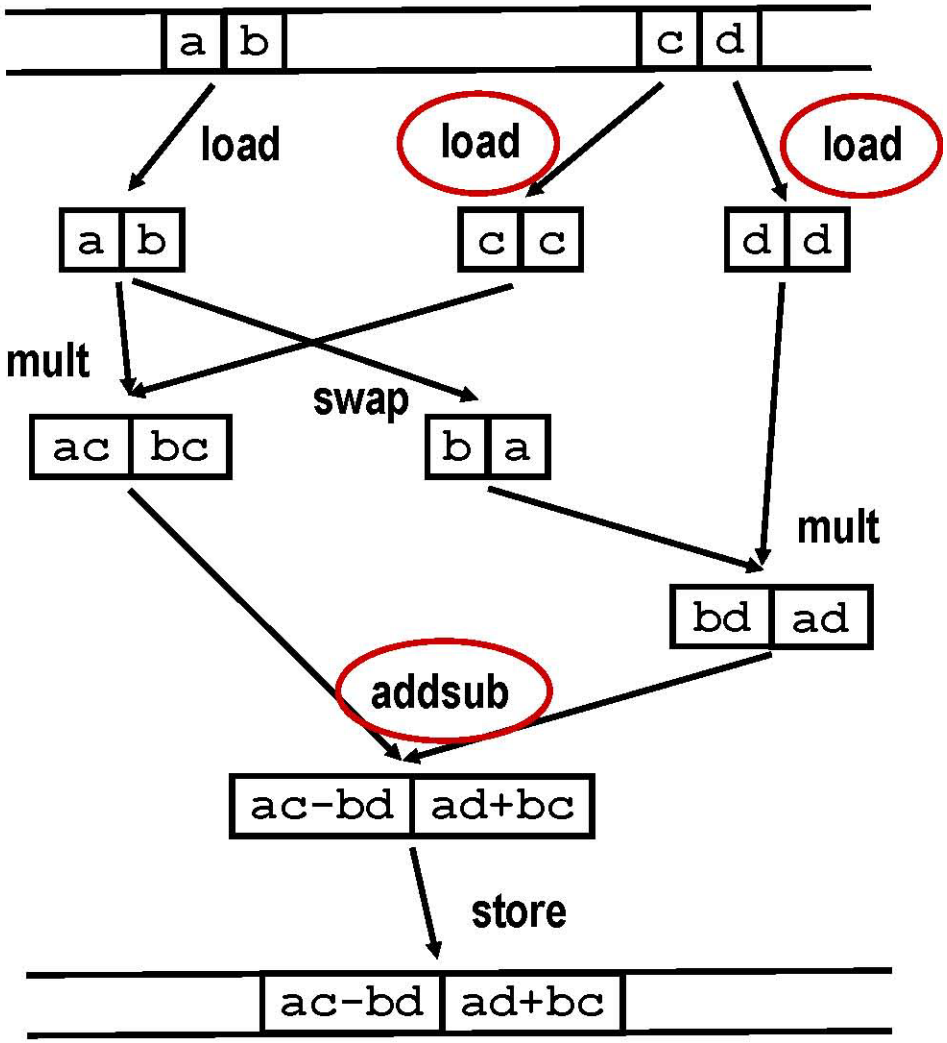
3 instructions
(_mm_haadd_ps)

total: 7 instructions

SSE4: has dot product instruction but still 7 instructions are needed (exercise)

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?)

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

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));
}
```

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

Part 2: AVX Programming

```
// example pseudocode, pre AVX
for each float element f in array
    f = sqrt(f)

// convert to AVX - pseudocode shown
for 4 (8) float elements at a time in array {
  1. load 4 (8) elements into AVX register
  2. calculate the square root of the 4 (8) elements
  in the register in parallel
  3. write the 4 (8) results from AVX register to
  memory
}
```

To use AVX in everyday C/C++ programs:

Use new datatype: `__m128` (or `__m256`)

and intrinsics interface:

```
#include <xmmintrin.h>  // AVX
#include <immintrin.h>  // AVX
#include <x86intrin.h>  //everything
```

Note: Intrinsics ≠ library functions

Commonly create arrays of __m128 (or __m256) from arrays of other types: `char, int, float, double`

Must align those arrays (see three method in next two slides)

Array Alignment `#include <stdlib.h>`

```
int posix_memalign(void **memptr, size_t alignment, size_t size);
```

The posix_memalign() function shall allocate *size* bytes aligned on a boundary specified by *alignment*, and shall return a pointer to the allocated memory in *memptr*. The value of *alignment* shall be a multiple of *sizeof(void *)*, that is also a power of two. Upon successful completion, the value pointed to by *memptr* shall be a multiple of alignment.

EXAMPLE:

```
err = posix_memalign((void**) &buf, ALIGN, BUFFSIZE);
if (err) { HANDLE ERROR }
```

The *free()* function shall deallocate memory that has previously been allocated by *posix_memalign()*.

```
void* _mm_malloc        a function call to allocate aligned memory
```

Used with _mm_free(x1)

```
//  void* _mm_malloc (int size, int align);
x1 = (float*)_mm_malloc(100 * sizeof(float),BOUNDARY_ALIGNMENT);
-- USE x1 --
_mm_free(x1);
```

Use *_mm_malloc* and *_mm_free* to allocate and free aligned blocks of memory. Based on *malloc* and *free*.

The keyword *__attribute__* allows you to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses. Four attributes are currently defined for variables: aligned, mode, packed, and section. Other attributes are defined for functions, and thus not documented here; see Function Attributes.

aligned (alignment)

This attribute specifies a minimum alignment for the variable or structure field, measured in bytes. For example, the declaration:

```
int x __attribute__ ((aligned (16))) = 0;
```

causes the compiler to allocate the global variable x on a 16-byte boundary.

You can also specify the alignment of structure fields. For example, to create a double-word aligned int pair, you could write:

```
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

This is an alternative to creating a union with a double member that forces the union to be double-word aligned.

Test Code -- Original Version

```
// Given two input arrays, compute the element-wise "hypotenuse"
// and save the results in the output array.

void ArrayTest1(float* pArray1,        // [in] 1st source array
                float* pArray2,        // [in] 2nd source array
                float* pResult,        // [out] result array
                int nSize)             // [in] size of all arrays
{
  int i;
  float* pSource1 = pArray1;
  float* pSource2 = pArray2;
  float* pDest = pResult;
  float  sqrtf(float x);

  for (i = 0; i < nSize; i++){
    *pDest = sqrtf((*pSource1) * (*pSource1) +
                   (*pSource2) * (*pSource2)) + 0.5f;
    pSource1++;
    pSource2++;
    pDest++;
  }
}
```

Test Code – AVX version (128 bit)

```
void ArrayTest2(float* pArray1,        // [in] 1st source array
                float* pArray2,        // [in] 2nd source array
                float* pResult,        // [out] result array
                int nSize)             // [in] size of all arrays
{
  int  i, nLoop = nSize/4;

  __m128   m1, m2, m3, m4;             // AVX registers
  __m128   m0_5 = _mm_set_ps1(0.5f);

  __m128*  pSrc1 = (__m128*) pArray1;  // recast as AVX pointers
  __m128*  pSrc2 = (__m128*) pArray2;
  __m128*  pDest = (__m128*) pResult;

  for (i = 0; i < nLoop; i++){
    m1 = _mm_mul_ps(*pSrc1, *pSrc1);   // AVX intrinsic mul
    m2 = _mm_mul_ps(*pSrc2, *pSrc2);
    m3 = _mm_add_ps(m1,m2);
    m4 = _mm_sqrt_ps(m3);
    *pDest = _mm_add_ps(m4,m0_5);      // round for conversion to int

    pSrc1++;
    pSrc2++;
    pDest++;
  }
}
```

Test Code – AVX2 version (256 bit)

```
void ArrayTest2(float* pArray1,          // [in] 1st source array
                float* pArray2,          // [in] 2nd source array
                float* pResult,          // [out] result array
                int nSize)               // [in] size of all arrays
{
  int  i, nLoop = nSize/8;

  __m256   m1, m2, m3, m4;                 // AVX registers
  __m256   m0_5 = _mm256_set1_ps(0.5f);   // broadcast to vector

  __m256*  pSrc1 = (__m256*) pArray1;     // recast as AVX pointers
  __m256*  pSrc2 = (__m256*) pArray2;
  __m256*  pDest = (__m256*) pResult;

  for (i = 0; i < nLoop; i++){
    m1 = _mm256_mul_ps(*pSrc1, *pSrc1);    // AVX intrinsic mul
    m2 = _mm256_mul_ps(*pSrc2, *pSrc2);
    m3 = _mm256_add_ps(m1,m2);
    m4 = _mm256_sqrt_ps(m3);
    *pDest = _mm_add256_ps(m4,m0_5);       // round for conversion to int

    pSrc1++;
    pSrc2++;
    pDest++;
  }
}
```

Part 3: AVX in C alone *(w/o intrinsics)*

GCC supports extensions to the C language that let programmers express a program in terms of vector operations that can be compiled into AVX.

Preferable to intrinsics (which is preferable to inline assembly language) because of enhanced flexibility & portability:

- Target machine independent
  - Generates scalar code if no AVX available
  - Generates non-AVX SIMD code as supported on target CPU

- Can parameterize by type (w/o recoding double/single/integer)

https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html

6.50 Using Vector Instructions through Built-in Functions

On some targets, the instruction set contains SIMD vector instructions which operate on multiple values contained in one large register at the same time. For example, on the x86 the MMX, 3DNow! and AVX extensions can be used this way. The first step in using these extensions is to provide the necessary data types. This should be done using an appropriate typedef:

```
typedef int v4si __attribute__ ((vector_size (16)));
```

The **int** type specifies the base type, while the attribute specifies the vector size for the variable, measured in bytes. For example, the declaration above *causes the compiler to set the mode for the v4si type to be 16 bytes wide and divided into **int** sized units*. For a 32-bit **int** this means a vector of 4 units of 4 bytes.

The vector_size attribute is only applicable to integral and float scalars, although arrays, pointers, and function return values are allowed in conjunction with this construct. Only sizes that are a power of two are currently allowed. All the basic integer types can be used as base types, both as signed and as unsigned: char, short, int, long, long long. In addition, float and double can be used to build floating-point vector types. Specifying a combination that is not valid for the current architecture causes GCC to synthesize the instructions using a narrower mode. For example, if you specify a variable of type V4SI and your architecture does not allow for this specific SIMD type, GCC produces code that uses 4 SIs.

The types defined in this manner can be used with a subset of normal C operations. Currently, GCC allows using the following operators on these types: +, -, *, /, unary minus, ^, |, &, ~, %. The operations behave like C++ valarrays. Addition is defined as the addition of the corresponding elements of the operands. For example, in the code below, each of the 4 elements in a is added to the corresponding 4 elements in b and the resulting vector is stored in c.

```
typedef int v4si __attribute__ ((vector_size (16)));
     v4si a, b, c;
     c = a + b;
```

EX: Combine – Step 1: Vector Declarations

Make code independent of data type and vector size so use typedef declarations and constant definitions

```
// Assume initialization of data_t, e.g.:  typedef float data_t;

/* Number of bytes in a vector */
#define VBYTES 32

/* Number of elements in a vector */
#define VSIZE VBYTES/sizeof(data_t)

/* Create vector data type vec_t with a GCC specific construct */
typedef data_t vec_t __attribute__((vector_size(VBYTES)));

/* Mechanism to access elements of a vector, i.e., overlay scalar & vector */
typedef union {
    vec_t v;
    data_t d[VSIZE];
} pack_t;
```

Step 2: Use Vectors (w/ dot product)

```
typedef float data_t;          // for example, make this a vector of floats

#define VBYTES 32
#define VSIZE VBYTES/sizeof(data_t)

typedef data_t vec_t __attribute__(vector_size(VBYTES)));

typedef union {
    vec_t v;
    data_t d[VSIZE];
} pack_t;

/* Compute inner product of a single AVX vector (not array of vectors!) */
data_t innerv(vec_t av, vec_t bv) {   // pass two vectors by value, return scalar

  pack_t xfer;                // declare vector/scalar union
  int i;

  vec_t pv = av * bv;         // declare vector pv and initialize w/ vector multiply
  data_t result = 0;          // declare and initialize scalar variable for return
  xfer.v = pv;                // copy vector into vector/scalar struct
  for (i = 0; i < VSIZE; i++)
    result += xfer.d[i];      // sum the element-wise products

  return result;
}
```

Step 3: Deal w/ Corner Cases
Ex: *Combine*

```
typedef float data_t;  // Ex:  floats

#define VBYTES 32
#define VSIZE VBYTES/sizeof(data_t)

typedef data_t vec_t
__attribute__(vector_size(VBYTES)));

typedef union {
    vec_t v;
    data_t d[VSIZE];
} pack_t;

void simd_v1_combine(vec_ptr v, data_t *dest)
{
 long int i;
 pack_t xfer;
 vec_t accum;
 data_t *data = get_vec_start(v);
 int cnt = vec_length(v);
 data_t result = IDENT;

/* Initialize accum entries to IDENT */
 for (i = 0; i < VSIZE; i++)
   xfer.d[i] = IDENT;
 accum = xfer.v;
```

```
/* Single step until have memory alignment */
 while (((long) data) % VBYTES && cnt) { // ptr
   result = result OP *data++;
   cnt--;
 }

/* Step through data with VSIZE-way parallelism */
 while (cnt >= VSIZE) {
   vec_t chunk = *((vec_t *) data);
   accum = accum OP chunk;
   data += VSIZE;
   cnt -= VSIZE;
 }

/* Single-step through remaining elements */
 while (cnt) {
   result = result OP *data++;
   cnt--;
 }

/* Combine elements of accumulator vector */
 xfer.v = accum;
 for (i = 0; i < VSIZE; i++)
   result = result OP xfer.d[i];

/* Store result */
 *dest = result;
}
```

SIMD/AVX Combine – Main Sections -

1. *Element-wise initialization of acc vector w/ scalar*
2. *Align front w/ scalar summation*
3. *Main loop → vector operations into acc vector*
4. *Remainder < vec length*
5. *Sum elements of accumulator vector*