

How to Write Fast Numerical Code

Spring 2019

Lecture: SIMD extensions, SSE, compiler vectorization

Instructor: Markus Püschel

TA: Tyler Smith, Gagandeep Singh, Alen Stojanov

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Flynn's Taxonomy

	Single instruction	Multiple instruction
Single data	<i>SISD</i> Uniprocessor	<i>MISD</i>
Multiple data	<i>SIMD</i> Vector computer Short vector extensions	<i>MIMD</i> Multiprocessors VLIW

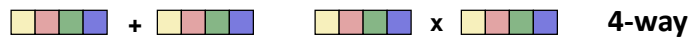
2

SIMD Extensions and SSE

- SSE intrinsics
- Compiler vectorization
- *This lecture and material was created together with Franz Franchetti (ECE, Carnegie Mellon)*

3

SIMD Vector Extensions

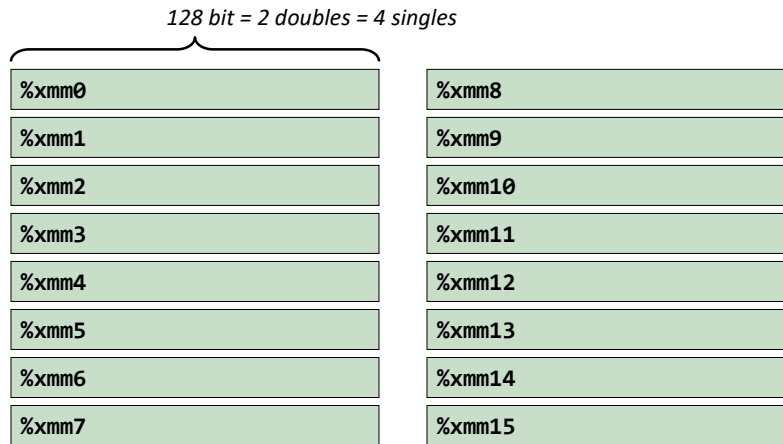


- **What is it?**
 - Extension of the ISA
 - Data types and instructions for the parallel computation on short (length 2, 4, 8, ...) vectors of integers or floats
 - Names: MMX, SSE, SSE2, ...
- **Why do they exist?**
 - **Useful:** Many applications have the necessary fine-grain parallelism
Then: speedup by a factor close to vector length
 - **Doable:** Relatively easy to design by replicating functional units

4

Core 2

- Has SSE3
- 16 SSE registers



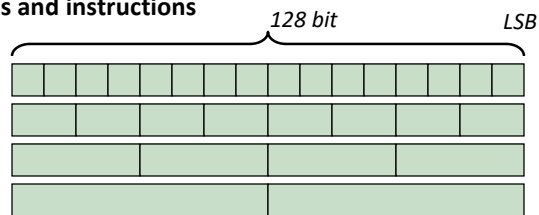
7

SSE3 Registers

- Used for different data types and instructions

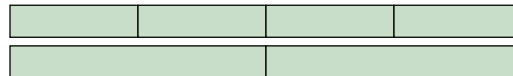
- Integer vectors:

- 16-way byte
- 8-way 2 bytes
- 4-way 4 bytes
- 2-way 8 bytes



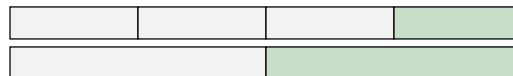
- Floating point vectors:

- 4-way single (since SSE)
- 2-way double (since SSE2)



- Floating point scalars:

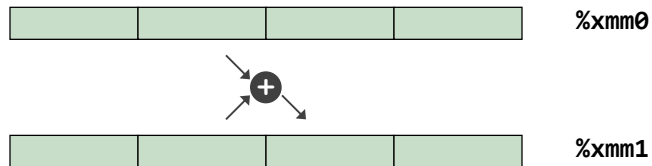
- single (since SSE)
- double (since SSE2)



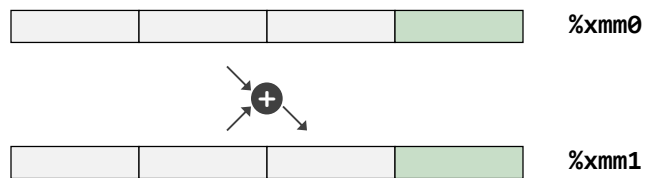
8

SSE3 Instructions: Examples

- Single precision *4-way vector add*: `addps %xmm0, %xmm1`



- Single precision *scalar add*: `addss %xmm0, %xmm1`



9

SSE3 Instruction Names

packed (vector)

`addps`

single precision

`addpd`

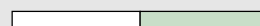
double precision

single slot (scalar)

`addss`



`addsd`



Compiler will use this for floating point

- on x86-64
- with proper flags if SSE/SSE2 is available

10

x86-64 FP Code Example

- Inner product of two vectors

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

```
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;
}
```

```
ipf:
    xorps    %xmm1, %xmm1           # result = 0.0
    xorl    %ecx, %ecx              # i = 0
    jmp     .L8                     # goto middle
.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
    jnl    .L10                    # if < goto loop
    movaps %xmm1, %xmm0           # return result
    ret
```

11

SSE: How to Take Advantage?



- Necessary: fine grain parallelism
- Options (ordered by effort):
 - Use vectorized libraries (easy, not always available)
 - Compiler vectorization (this lecture)
 - Use intrinsics (this lecture)
 - Write assembly
- We will focus on floating point and single precision (4-way)

12

SIMD Extensions and SSE

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

References:

[*Intel Intrinsics Guide*](#)

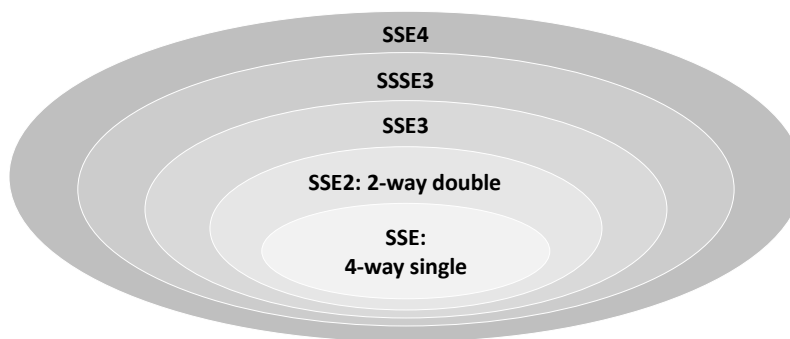
(easy access to all instructions, also contains latency and throughput information!)

Intel icc compiler manual

Visual Studio manual

13

SSE Family: Floating Point



- Not drawn to scale
- From SSE2: Only additional instructions
- *Every Core 2 has SSE3*

14

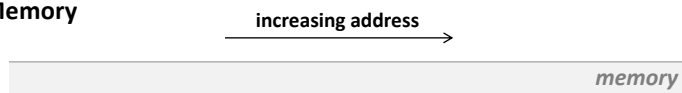
Intrinsics

- **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

15

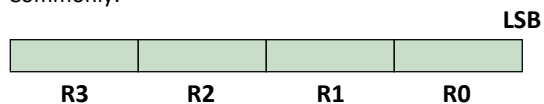
Visual Conventions We Will Use

- **Memory**

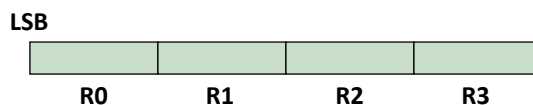


- **Registers**

- Commonly:



- We will use

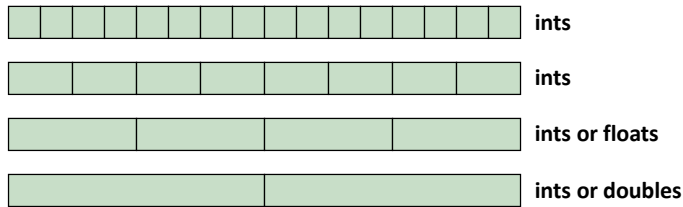


16

SSE Intrinsics (Focus Floating Point)

■ Data types

```
__m128 f; // = {float f0, f1, f2, f3}  
__m128d d; // = {double d0, d1}  
__m128i i; // 16 8-bit, 8 16-bit, 4 32-bit, or 2 64-bit ints
```



17

SSE Intrinsics (Focus Floating Point)

■ Instructions

- Naming convention: `_mm_<intrin_op>_<suffix>`
- Example:

```
// a is 16-byte aligned  
float a[4] = {1.0, 2.0, 3.0, 4.0};  
__m128 t = _mm_load_ps(a);
```

p: packed
s: single precision

LSB

1.0	2.0	3.0	4.0
-----	-----	-----	-----

- Same result as

```
__m128 t = _mm_set_ps(4.0, 3.0, 2.0, 1.0)
```

18

SSE Intrinsics

- **Native instructions (one-to-one with assembly)**

`_mm_load_ps()` ↔ `movaps`

`_mm_add_ps()` ↔ `addps`

`_mm_mul_pd()` ↔ `mulpd`

...

- **Multi instructions (map to several assembly instructions)**

`_mm_set_ps()`

`_mm_set1_ps()`

...

- **Macros and helpers**

`_MM_TRANSPOSE4_PS()`

`_MM_SHUFFLE()`

...

19

What Are the Main Issues?

- **Alignment is important (128 bit = 16 byte)**
- **You need to code explicit loads and stores**
- **Overhead through shuffles**

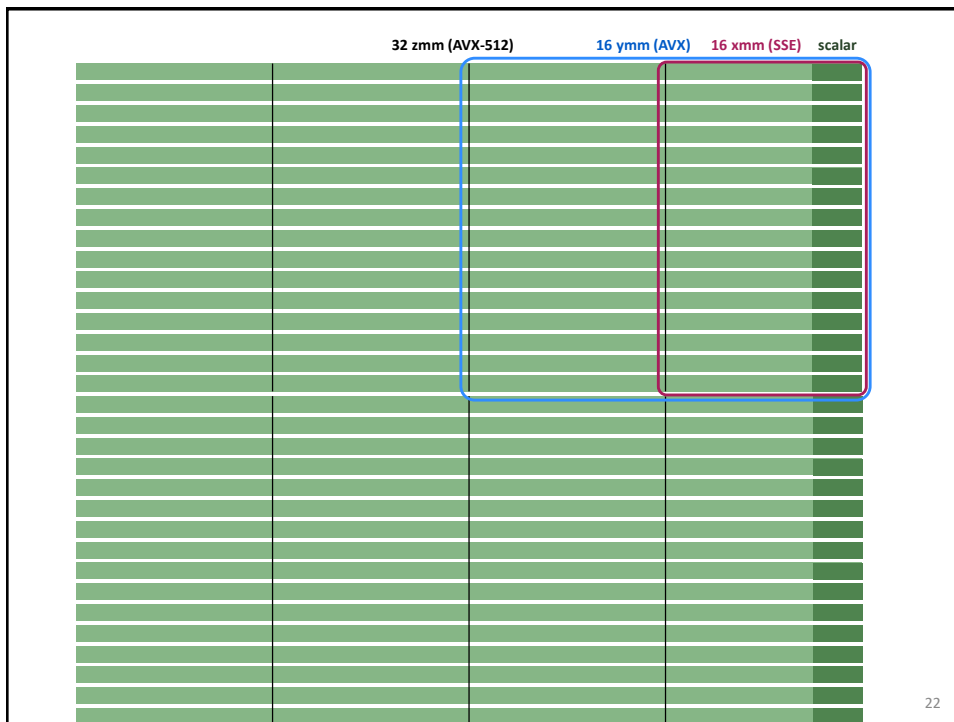
20

SSE vs. AVX

	SSE	AVX
float, double	4-way, 2-way	8-way, 4-way
register	16 x 128 bits: %xmm0 - %xmm15	16 x 256 bits: %ymm0 - %ymm15 The lower halves are the %xmms
assembly ops	addps, mulpd, ...	vaddps, vmulpd
intrinsics data type	__m128, __m128d	__m256, __m256d
intrinsics instructions	_mm_load_ps, _mm_add_pd, ...	_mm256_load_ps, _mm256_add_pd

Mixing SSE and AVX may incur penalties

21



22

SSE Intrinsics

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

23

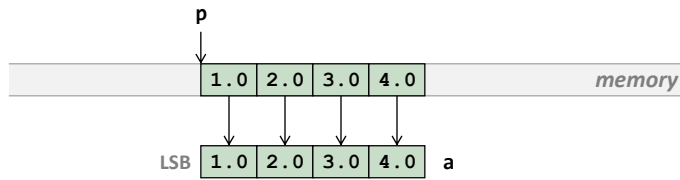
Loads and Stores

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

Intrinsic Name	Operation	Corresponding SSE Instruction
<code>_mm_set_ss</code>	Set the low value and clear the three high values	Composite
<code>_mm_set1_ps</code>	Set all four words with the same value	Composite
<code>_mm_set_ps</code>	Set four values, address aligned	Composite
<code>_mm_setr_ps</code>	Set four values, in reverse order	Composite
<code>_mm_setzero_ps</code>	Clear all four values	Composite

24

Loads and Stores



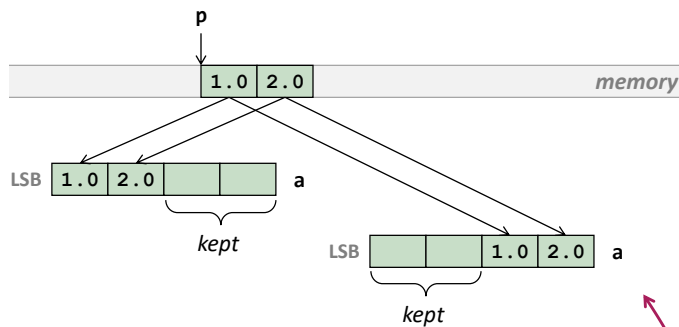
```
a = _mm_load_ps(p); // p 16-byte aligned
```

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

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

load_ps on unaligned pointer: seg fault

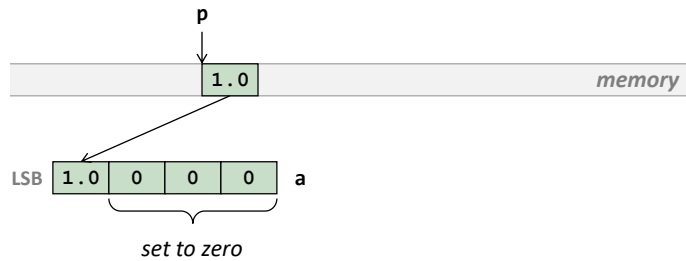
Loads and Stores



```
a = _mm_loadl_pi(a, p); // p 8-byte aligned
```

```
a = _mm_loadh_pi(a, p); // p 8-byte aligned
```

Loads and Stores



```
a = _mm_load_ss(p); // p any alignment
```

→ blackboard 27

Stores Analogous to Loads

Intrinsic Name	Operation	Corresponding SSE Instruction
<code>_mm_storeh_pi</code>	Store high	MOVHPS mem, reg
<code>_mm_storel_pi</code>	Store low	MOVLPS mem, reg
<code>_mm_store_ss</code>	Store the low value	MOVSS
<code>_mm_store1_ps</code>	Store the low value across all four words, address aligned	Shuffling + MOVSS
<code>_mm_store_ps</code>	Store four values, address aligned	MOVAPS
<code>_mm_storeu_ps</code>	Store four values, address unaligned	MOVUPS
<code>_mm_storer_ps</code>	Store four values, in reverse order	MOVAPS + Shuffling

28

Constants

LSB

1.0	2.0	3.0	4.0
-----	-----	-----	-----

 a

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

LSB

1.0	1.0	1.0	1.0
-----	-----	-----	-----

 b

```
b = _mm_set1_ps(1.0);
```

LSB

1.0	0	0	0
-----	---	---	---

 c

```
c = _mm_set_ss(1.0);
```

LSB

0	0	0	0
---	---	---	---

 d

```
d = _mm_setzero_ps();
```

→ blackboard ²⁹

Arithmetic

SSE

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_add_ss	Addition	ADDSS
_mm_add_ps	Addition	ADDPS
_mm_sub_ss	Subtraction	SUBSS
_mm_sub_ps	Subtraction	SUBPS
_mm_mul_ss	Multiplication	MULSS
_mm_mul_ps	Multiplication	MULPS
_mm_div_ss	Division	DIVSS
_mm_div_ps	Division	DIVPS
_mm_sqrt_ss	Squared Root	SQRTSS
_mm_sqrt_ps	Squared Root	SQRTPS
_mm_rcp_ss	Reciprocal	RCPPSS
_mm_rcp_ps	Reciprocal	RCPPS
_mm_rsqrt_ss	Reciprocal Squared Root	RSQRTSS
_mm_rsqrt_ps	Reciprocal Squared Root	RSQRTPS
_mm_min_ss	Computes Minimum	MINSS
_mm_min_ps	Computes Minimum	MINPS
_mm_max_ss	Computes Maximum	MAXSS
_mm_max_ps	Computes Maximum	MAXPS

SSE3

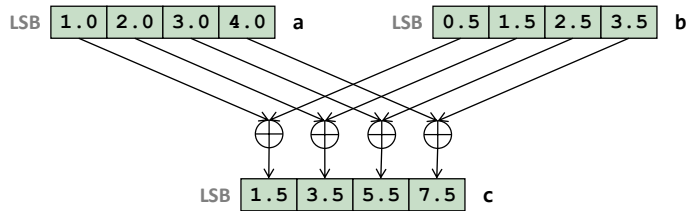
Intrinsic Name	Operation	Corresponding SSE3 Instruction
_mm_addsub_ps	Subtract and add	ADDSUBPS
_mm_hadd_ps	Add	HADDPS
_mm_hsub_ps	Subtracts	HSUBPS

SSE4

Intrinsic	Operation	Corresponding SSE4 Instruction
_mm_dp_ps	Single precision dot product	DPPS

30

Arithmetic



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

analogous:

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

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

→ blackboard 31

Example

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

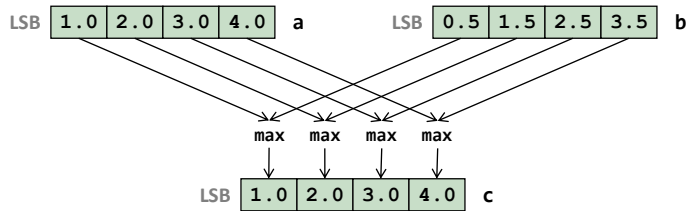
```
#include <ia32intrin.h>  
  
// n a multiple of 4, x is 16-byte aligned  
void addindex_vec(float *x, int n) {  
    __m128 index, x_vec;  
  
    for (int i = 0; i < n; i+=4) {  
        x_vec = _mm_load_ps(x+i); // load 4 floats  
        index = _mm_set_ps(i+3, i+2, i+1, i); // create vector with indexes  
        x_vec = _mm_add_ps(x_vec, index); // add the two  
        _mm_store_ps(x+i, x_vec); // store back  
    }  
}
```

Is this the best solution?

No! `_mm_set_ps` may be too expensive

32

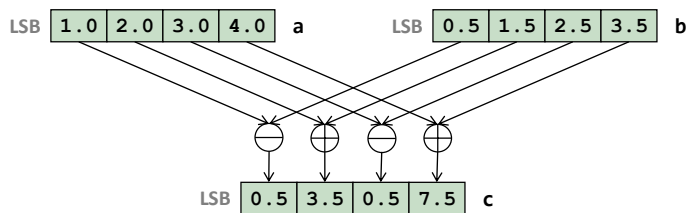
Arithmetic



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

35

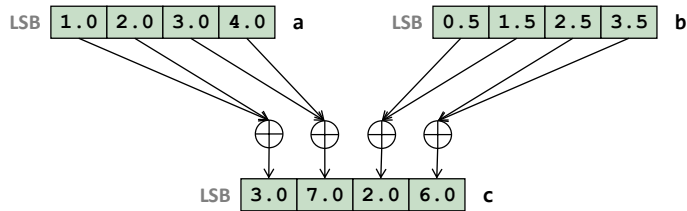
Arithmetic



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

36

Arithmetic



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

analogous:

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

→ blackboard 37

Example

```
// 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;
}
```

```
#include <ia32intrin.h>

// 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
    }
}
```

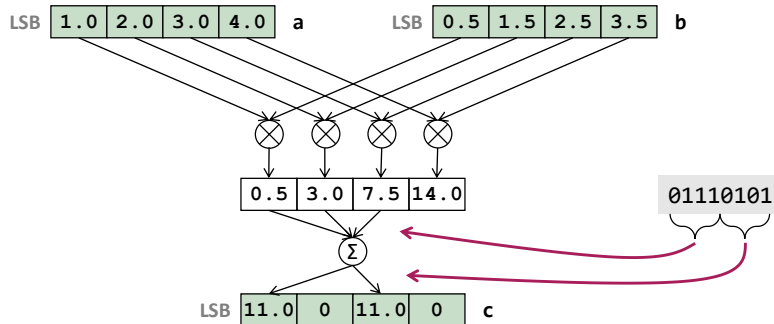
38

Arithmetic

```
__m128 __mm_dp_ps(__m128 a, __m128 b, const int mask)
```

(SSE4) Computes the pointwise product of a and b and writes a selected sum of the resulting numbers into selected elements of c; the others are set to zero. The selections are encoded in the mask.

Example: mask = 117 = 01110101



39

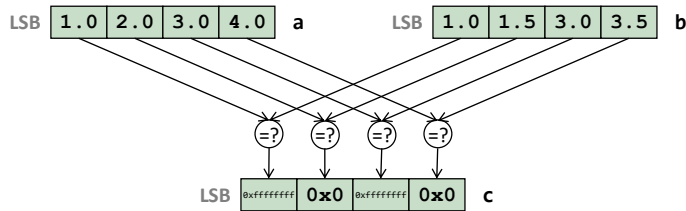
Comparisons

Intrinsic Name	Operation	Corresponding SSE Instruction
<code>__mm_cmpeq_ss</code>	Equal	CMPEQSS
<code>__mm_cmpeq_ps</code>	Equal	CMPEQPS
<code>__mm_cmlt_ss</code>	Less Than	CMLTSS
<code>__mm_cmlt_ps</code>	Less Than	CMLTPS
<code>__mm_cmple_ss</code>	Less Than or Equal	CMPLSS
<code>__mm_cmple_ps</code>	Less Than or Equal	CMPLPS
<code>__mm_cmpgt_ss</code>	Greater Than	CMPGTSS
<code>__mm_cmpgt_ps</code>	Greater Than	CMPGTPS
<code>__mm_cmpge_ss</code>	Greater Than or Equal	CMPSSE
<code>__mm_cmpge_ps</code>	Greater Than or Equal	CMPSPS
<code>__mm_cmpneq_ss</code>	Not Equal	CMPNEQSS
<code>__mm_cmpneq_ps</code>	Not Equal	CMPNEQPS
<code>__mm_cmnlt_ss</code>	Not Less Than	CMPNLTSS
<code>__mm_cmnlt_ps</code>	Not Less Than	CMPNLTPS
<code>__mm_cmnle_ss</code>	Not Less Than or Equal	CMPNLESS
<code>__mm_cmnle_ps</code>	Not Less Than or Equal	CMPNLEPS
<code>__mm_cmpngt_ss</code>	Not Greater Than	CMPNLTSS
<code>__mm_cmpngt_ps</code>	Not Greater Than	CMPNLTPS
<code>__mm_cmpnge_ss</code>	Not Greater Than or Equal	CMPNLESS
<code>__mm_cmpnge_ps</code>	Not Greater Than or Equal	CMPNLEPS

Intrinsic Name	Operation	Corresponding SSE Instruction
<code>__mm_cmpord_ss</code>	Ordered	CMPORDSS
<code>__mm_cmpord_ps</code>	Ordered	CMPORDPS
<code>__mm_cmpunord_ss</code>	Unordered	CMPUNORDSS
<code>__mm_cmpunord_ps</code>	Unordered	CMPUNORDPS
<code>__mm_comieq_ss</code>	Equal	COMISS
<code>__mm_comilt_ss</code>	Less Than	COMISS
<code>__mm_comile_ss</code>	Less Than or Equal	COMISS
<code>__mm_comigt_ss</code>	Greater Than	COMISS
<code>__mm_comige_ss</code>	Greater Than or Equal	COMISS
<code>__mm_comineq_ss</code>	Not Equal	COMISS
<code>__mm_ucomieq_ss</code>	Equal	UCOMISS
<code>__mm_ucomilt_ss</code>	Less Than	UCOMISS
<code>__mm_ucomile_ss</code>	Less Than or Equal	UCOMISS
<code>__mm_ucomigt_ss</code>	Greater Than	UCOMISS
<code>__mm_ucomige_ss</code>	Greater Than or Equal	UCOMISS
<code>__mm_ucomineq_ss</code>	Not Equal	UCOMISS

40

Comparisons



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

analogous:

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

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

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

etc.

Each field:
0xffffffff if true
0x0 if false

Return type: `__m128`

41

Example

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

    for(i = 0; i < n; i++) {
        if(x[i] > 0.5)
            x[i] += 1.;
        else x[i] -= 1.;
    }
}
```

```
#include <xmmintrin.h>

void fcond(float *x, 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(x+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(x+i, vr);
    }
}
```

42

Vectorization

=



Picture: www.druckundbestell.de

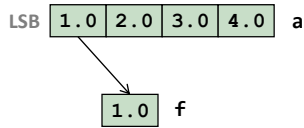
Conversion

Intrinsic Name	Operation	Corresponding SSE Instruction
<code>_mm_cvtsi32_si32</code>	Convert to 32-bit integer	CVTSS2SI
<code>_mm_cvtsi64_si64*</code>	Convert to 64-bit integer	CVTSS2SI
<code>_mm_cvtps_pi32</code>	Convert to two 32-bit integers	CVTPS2PI
<code>_mm_cvtss_si32</code>	Convert to 32-bit integer	CVTSS2SI
<code>_mm_cvtss_si64*</code>	Convert to 64-bit integer	CVTSS2SI
<code>_mm_cvtps_pi32</code>	Convert to two 32-bit integers	CVTTPS2PI
<code>_mm_cvtsi32_ss</code>	Convert from 32-bit integer	CVTSI2SS
<code>_mm_cvtsi64_ss*</code>	Convert from 64-bit integer	CVTSI2SS
<code>_mm_cvtpi32_ps</code>	Convert from two 32-bit integers	CVTTPS2PS
<code>_mm_cvtpi16_ps</code>	Convert from four 16-bit integers	composite
<code>_mm_cvtpu16_ps</code>	Convert from four 16-bit integers	composite
<code>_mm_cvtpi8_ps</code>	Convert from four 8-bit integers	composite
<code>_mm_cvtpu8_ps</code>	Convert from four 8-bit integers	composite
<code>_mm_cvtpi32x2_ps</code>	Convert from four 32-bit integers	composite
<code>_mm_cvtps_pi16</code>	Convert to four 16-bit integers	composite
<code>_mm_cvtps_pi8</code>	Convert to four 8-bit integers	composite
<code>_mm_cvtsi_f32</code>	Extract	composite

44

Conversion

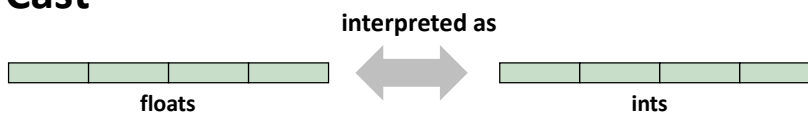
```
float _mm_cvtss_f32(__m128 a)
```



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

45

Cast



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

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

Reinterprets the four single precision floating point values in `a` as four 32-bit integers, and vice versa.

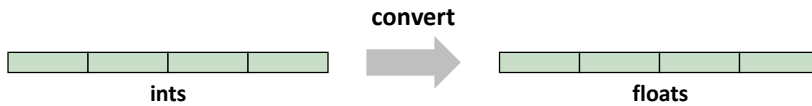
No conversion is performed. Does not map to any assembly instructions.

Makes integer shuffle instructions usable for floating point.

→ blackboard

Actual Conversion

```
__m128 __mm_cvt_pi2ps (__m128 a, __m64 b)
```



Shuffles

SSE

Intrinsic Name	Operation	Corresponding SSE Instruction
<code>__mm_shuffle_ps</code>	Shuffle	SHUFPS
<code>__mm_unpackhi_ps</code>	Unpack High	UNPCKHPS
<code>__mm_unpacklo_ps</code>	Unpack Low	UNPCKLPS
<code>__mm_move_ss</code>	Set low word, pass in three high values	MOVSS
<code>__mm_movehl_ps</code>	Move High to Low	MOVHLPS
<code>__mm_movelh_ps</code>	Move Low to High	MOVLHPS
<code>__mm_movemask_ps</code>	Create four-bit mask	MOVMSKPS

SSE3

Intrinsic Name	Operation	Corresponding SSE3 Instruction
<code>__mm_movehdup_ps</code>	Duplicates	MOVSHDUP
<code>__mm_movelldup_ps</code>	Duplicates	MOVSLDUP

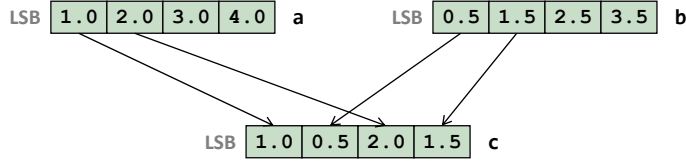
SSSE3

Intrinsic Name	Operation	Corresponding SSSE3 Instruction
<code>__mm_shuffle_epi8</code>	Shuffle	PSHUFB
<code>__mm_alignr_epi8</code>	Shift	PALIGNR

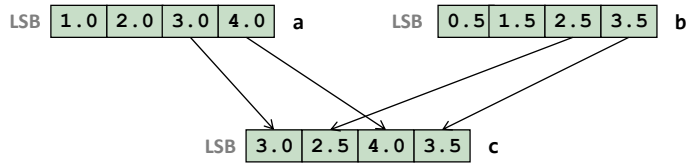
SSE4

Intrinsic Syntax	Operation	Corresponding SSE4 Instruction
<code>__m128 __mm_blend_ps(__m128 v1, __m128 v2, const int mask)</code>	Selects float single precision data from 2 sources using constant mask	BLENDDPS
<code>__m128 __mm_blendv_ps(__m128 v1, __m128 v2, __m128 v3)</code>	Selects float single precision data from 2 sources using variable mask	BLENDDVPS
<code>__m128 __mm_insert_ps(__m128 dst, __m128 src, const int ndx)</code>	Insert single precision float into packed single precision array element selected by index.	INSERTPS
<code>int __mm_extract_ps(__m128 src, const int ndx)</code>	Extract single precision float from packed single precision array selected by index.	EXTRACTPS

Shuffles



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

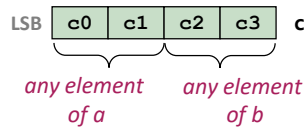


```
c = _mm_unpackhi_ps(a, b);
```

Shuffles

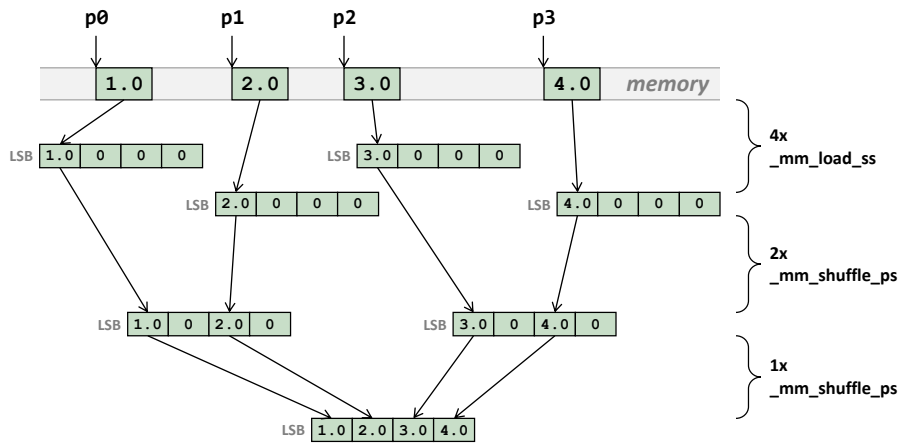
```
c = _mm_shuffle_ps(a, b, _MM_SHUFFLE(1, k, j, i));
```

helper macro to create mask



```
c0 = ai
c1 = aj
c2 = bk
c3 = bl
i, j, k, l in {0, 1, 2, 3}
```

Example: Loading 4 Real Numbers from Arbitrary Memory Locations



7 instructions, this is one good way of doing it

51

Code For Previous Slide

```
#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));
}
```

52

Example: Loading 4 Real Numbers from Arbitrary Memory Locations (cont'd)

- Whenever possible avoid the previous situation
- Restructure algorithm and use the aligned `_mm_load_ps()`
- Other possibility (but likely also yields 7 instructions)

```
__m128 vf;  
vf = _mm_set_ps(*p3, *p2, *p1, *p0);
```

- SSE4: `_mm_insert_epi32` together with `_mm_castsi128_ps`
 - Not clear whether better

53

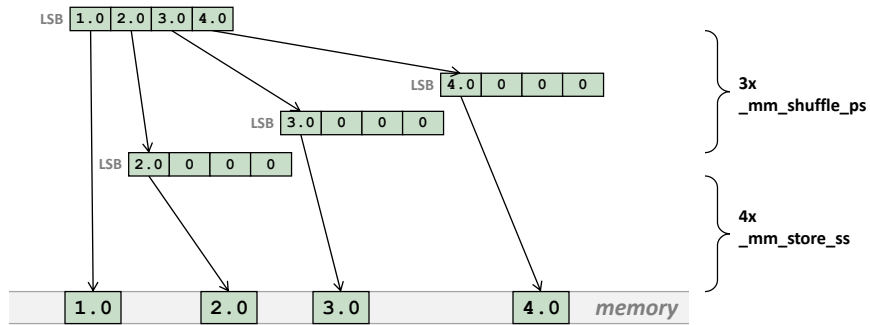
Example: Loading 4 Real Numbers from Arbitrary Memory Locations (cont'd)

- Do not do this (why?):

```
__declspec(align(16)) float g[4];  
__m128 vf;  
  
g[0] = *p0;  
g[1] = *p1;  
g[2] = *p2;  
g[3] = *p3;  
vf = _mm_load_ps(g);
```

54

Example: Storing 4 Real Numbers to Arbitrary Memory Locations



7 instructions, shorter critical path

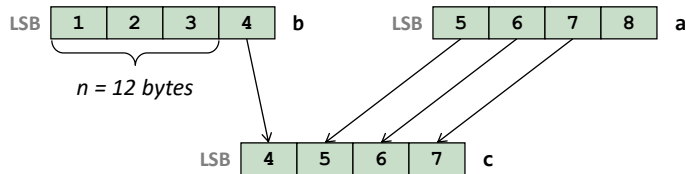
55

Shuffle

```
__m128i _mm_alignr_epi8(__m128i a, __m128i b, const int n)
```

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

Example: View `__m128i` as 4 32-bit ints; $n = 12$



How to use this with floating point vectors?

Use with `_mm_castsi128_ps` !

56

Example

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

```
#include <ia32intrin.h>

// n a multiple of 4, x, y are 16-byte aligned
void shift_vec(float *x, float *y, int n) {
    __m128 f;
    __m128i i1, i2, i3;

    i1 = _mm_castps_si128(_mm_load_ps(x));           // load first 4 floats and cast to int

    for (int i = 0; i < n-8; i = i + 4) {
        i2 = _mm_castps_si128(_mm_load_ps(x+4+i));   // load next 4 floats and cast to int
        f = _mm_castsi128_ps(_mm_alignr_epi8(i2,i1,4)); // shift and extract and cast back
        _mm_store_ps(y+i,f);                          // store it
        i1 = i2;                                       // make 2nd element 1st
    }

    // we are at the last 4
    i2 = _mm_castps_si128(_mm_setzero_ps());        // set the second vector to 0 and cast to int
    f = _mm_castsi128_ps(_mm_alignr_epi8(i2,i1,4)); // shift and extract and cast back
    _mm_store_ps(y+n-4,f);                          // store it
}
```

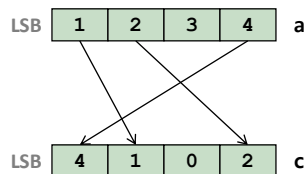
57

Shuffle

```
__m128i _mm_shuffle_epi8(__m128i a, __m128i mask)
```

Result is filled in each position by any element of a or with 0, as specified by mask

Example: View `__m128i` as 4 32-bit ints



Use with `_mm_castsi128_ps` to do the same for floating point

58

Shuffle

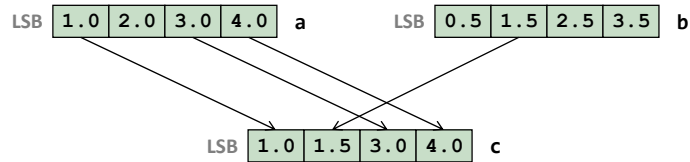
```
__m128 __mm_blendv_ps(__m128 a, __m128 b, __m128 mask)
```

(SSE4) Result is filled in each position by an element of a or b in the same position as specified by mask

Example: LSB

0x0	0xffffffff	0x0	0x0
-----	------------	-----	-----

 mask



see also `_mm_blend_ps`

59

Example (Continued From Before)

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

    for(i = 0; i < n; i++) {
        if(x[i] > 0.5)
            x[i] += 1.;
        else x[i] -= 1.;
    }
}
```

```
#include <xmmintrin.h>

void fcond(float *a, size_t n) {
    int i;
    __m128 vt, vmask, vp, vm, vr, ones, mones, thresholds;

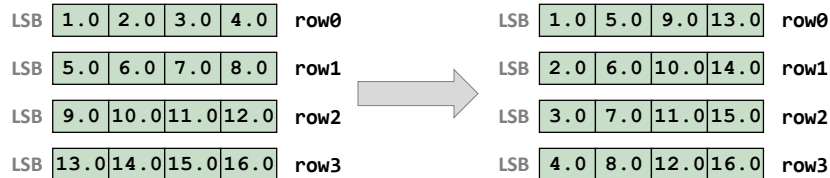
    ones      = _mm_set1_ps(1.);
    mones     = _mm_set1_ps(-1.);
    thresholds = _mm_set1_ps(0.5);
    for(i = 0; i < n; i+=4) {
        vt = _mm_load_ps(a+i);
        vmask = _mm_cmpgt_ps(vt, thresholds);
        vb = _mm_blendv_ps(mones, ones, vmask);
        vr = _mm_add_ps(vt, vb);
        _mm_store_ps(x+i, vr);
    }
}
```

60

Shuffle

```
__MM_TRANSPOSE4_PS(row0, row1, row2, row3)
```

Macro for 4 x 4 matrix transposition: The arguments row0,..., row3 are __m128 values each containing a row of a 4 x 4 matrix. After execution, row0, .., row3 contain the columns of that matrix.



In SSE: 8 shuffles (`4_mm_unpacklo_ps`, `4_mm_unpackhi_ps`)

61

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!](#))

62

SIMD Extensions and SSE

- SSE intrinsics
- *Compiler vectorization*

References:

[*Intel icc manual*](#) (look for auto vectorization)

63

Compiler Vectorization

- Compiler flags
- Aliasing
- Proper code style
- Alignment

64

How Do I Know the Compiler Vectorized?

- vec-report
- **Look at assembly:** mulps, addps, xxxps
- **Generate assembly with source code annotation:**
 - Visual Studio + icc: /Fas
 - icc on Linux/Mac: -S

65

Example

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

unvectorized: /Qvec-

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

vectorized:

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

why this?

why everything twice?
why movups and movaps?

unaligned

aligned

66

Aliasing

```
for (i = 0; i < n; i++)  
    a[i] = a[i] + b[i];
```

Cannot be vectorized in a straightforward way due to potential aliasing.

However, in this case compiler can insert runtime check:

```
if (a + n < b || b + n < a)  
    /* vectorized loop */  
    ...  
else  
    /* serial loop */  
    ...
```

67

Removing Aliasing

- Globally with compiler flag:
 - -fno-alias, /Oa
 - -fargument-noalias, /Qalias-args- (function arguments only)

- For one loop: pragma

```
void add(float *a, float *b, int n) {  
    #pragma ivdep  
    for (i = 0; i < n; i++)  
        a[i] = a[i] + b[i];  
}
```

- For specific arrays: restrict (needs compiler flag -restrict, /Qrestrict)

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

68

Proper Code Style

- Use countable loops = number of iterations known at runtime
 - *Number of iterations is a:*
 - constant
 - loop invariant term
 - linear function of outermost loop indices
- Countable or not?

```
for (i = 0; i < n; i++)  
    a[i] = a[i] + b[i];
```

```
void vsum(float *a, float *b, float *c) {  
    int i = 0;  
  
    while (a[i] > 0.0) {  
        a[i] = b[i] * c[i];  
        i++;  
    }  
}
```

69

Proper Code Style

- Use arrays, structs of arrays, not arrays of structs
- Ideally: unit stride access in innermost loop

```
void mmm1(float *a, float *b, float *c) {  
    int N = 100;  
    int i, j, k;  
  
    for (i = 0; i < N; i++)  
        for (j = 0; j < N; j++)  
            for (k = 0; k < N; k++)  
                c[i][j] = c[i][j] + a[i][k] * b[k][j];  
}
```

```
void mmm2(float *a, float *b, float *c) {  
    int N = 100;  
    int i, j, k;  
  
    for (i = 0; i < N; i++)  
        for (k = 0; k < N; k++)  
            for (j = 0; j < N; j++)  
                c[i][j] = c[i][j] + a[i][k] * b[k][j];  
}
```

70

Alignment

```
float *x = (float *) malloc(1024*sizeof(float));
int i;

for (i = 0; i < 1024; i++)
    x[i] = 1;
```

Cannot be vectorized in a straightforward way since x may not be aligned

However, the compiler can peel the loop to extract aligned part:

```
float *x = (float *) malloc(1024*sizeof(float));
int i;

peel = x & 0x0f; /* x mod 16 */
if (peel != 0) {
    peel = 16 - peel;
    /* initial segment */
    for (i = 0; i < peel; i++)
        x[i] = 1;
}
/* 16-byte aligned access */
for (i = peel; i < 1024; i++)
    x[i] = 1;
```

71

Ensuring Alignment

- Align arrays to 16-byte boundaries (see earlier discussion)
- If compiler cannot analyze:
 - Use pragma for loops

```
float *x = (float *) malloc(1024*sizeof(float));
int i;

#pragma vector aligned
for (i = 0; i < 1024; i++)
    x[i] = 1;
```

- For specific arrays:
 __assume_aligned(a, 16);

72

More Tips (icc 14.0) <https://software.intel.com/en-us/node/512631>

- **Use simple for loops.** Avoid complex loop termination conditions – **the upper iteration limit must be invariant within the loop.** For the innermost loop in a nest of loops, you could set the upper limit iteration to be a function of the outer loop indices.
- **Write straight-line code.** Avoid branches such as **switch, goto, or return statements**, most function calls, or if constructs that can not be treated as masked assignments.
- **Avoid dependencies between loop iterations** or at the least, avoid read-after-write dependencies.
- Try to use **array notations instead of the use of pointers.** C programs in particular impose very few restrictions on the use of pointers; aliased pointers may lead to unexpected dependencies. Without help, the compiler often cannot tell whether it is safe to vectorize code containing pointers.
- Wherever possible, **use the loop index directly in array subscripts** instead of incrementing a separate counter for use as an array address.
- Access memory efficiently:
 - **Favor inner loops with unit stride.**
 - **Minimize indirect addressing.**
 - **Align your data to 16 byte boundaries (for SSE instructions).**
- Choose a suitable data layout with care. **Most multimedia extension instruction sets are rather sensitive to alignment.**
- ...

73

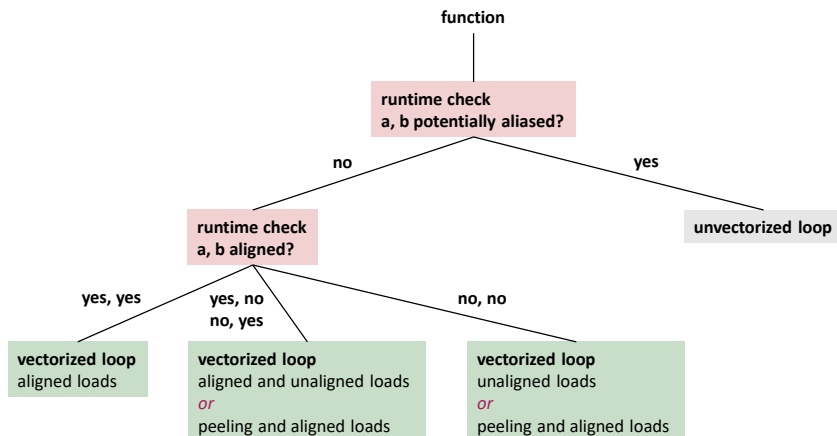
```
void myadd(float *a, float *b, const int n) {
    for (int i = 0; i < n; i++)
        a[i] = a[i] + b[i];
}
```

Assume:

- **No aliasing information**
- **No alignment information**

Can compiler vectorize?

Yes: Through versioning



Compiler Vectorization

- Read manual