

Performance Engineering of Software Systems

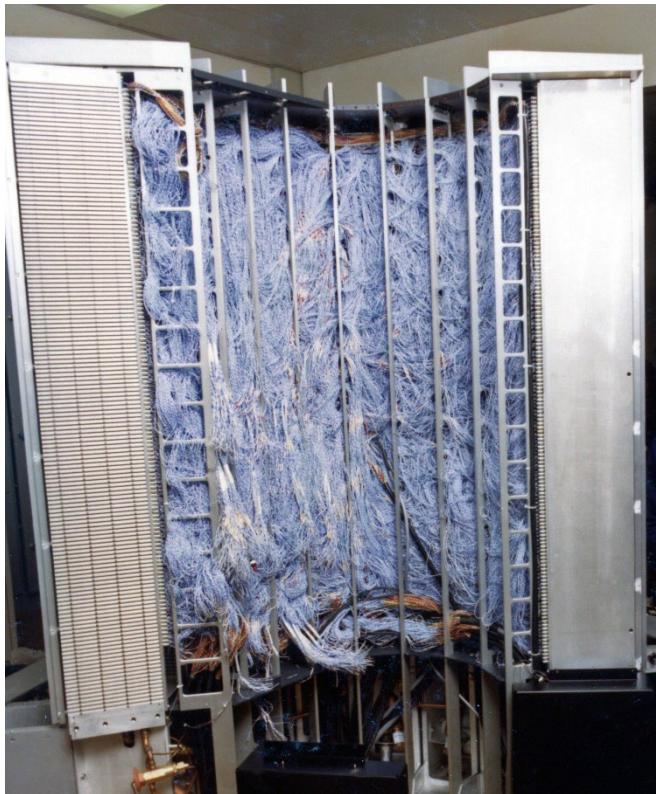
LECTURE 6
Vectorization
Saman Amarasinghe*

September 27, 2022



Vectorization

The First Supercomputers were developed in the 70's

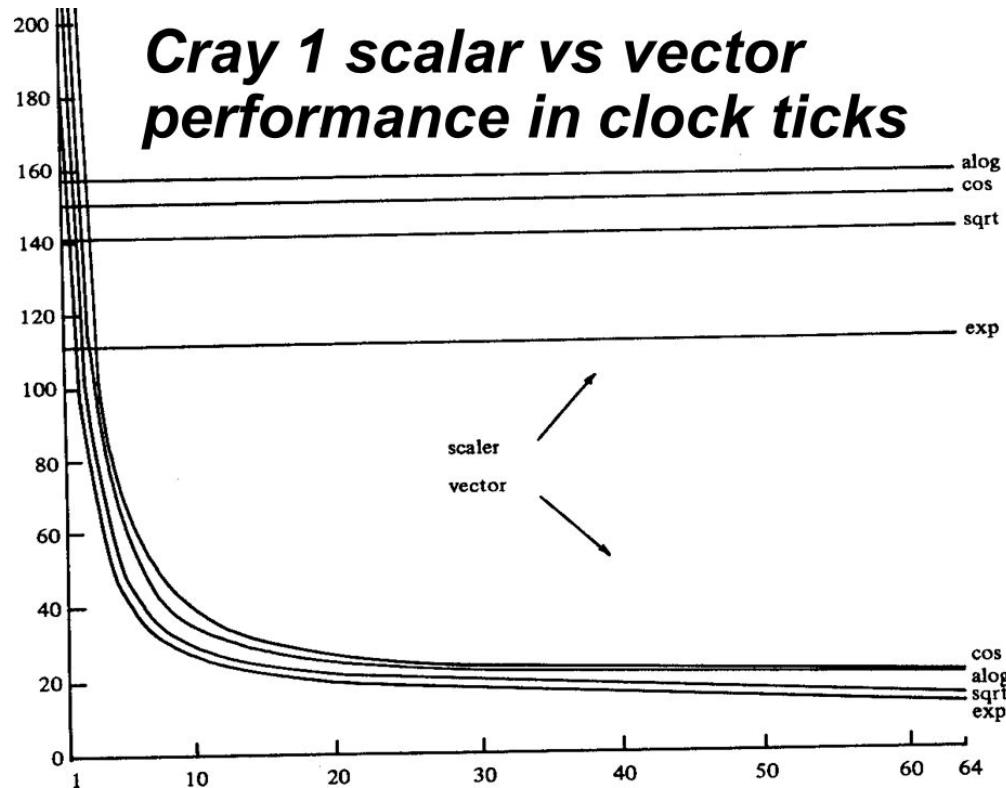


200,000 gates, 80 MHz, \$10M

Seymour Cray

Vectorization

The First Supercomputers were developed in the 70's



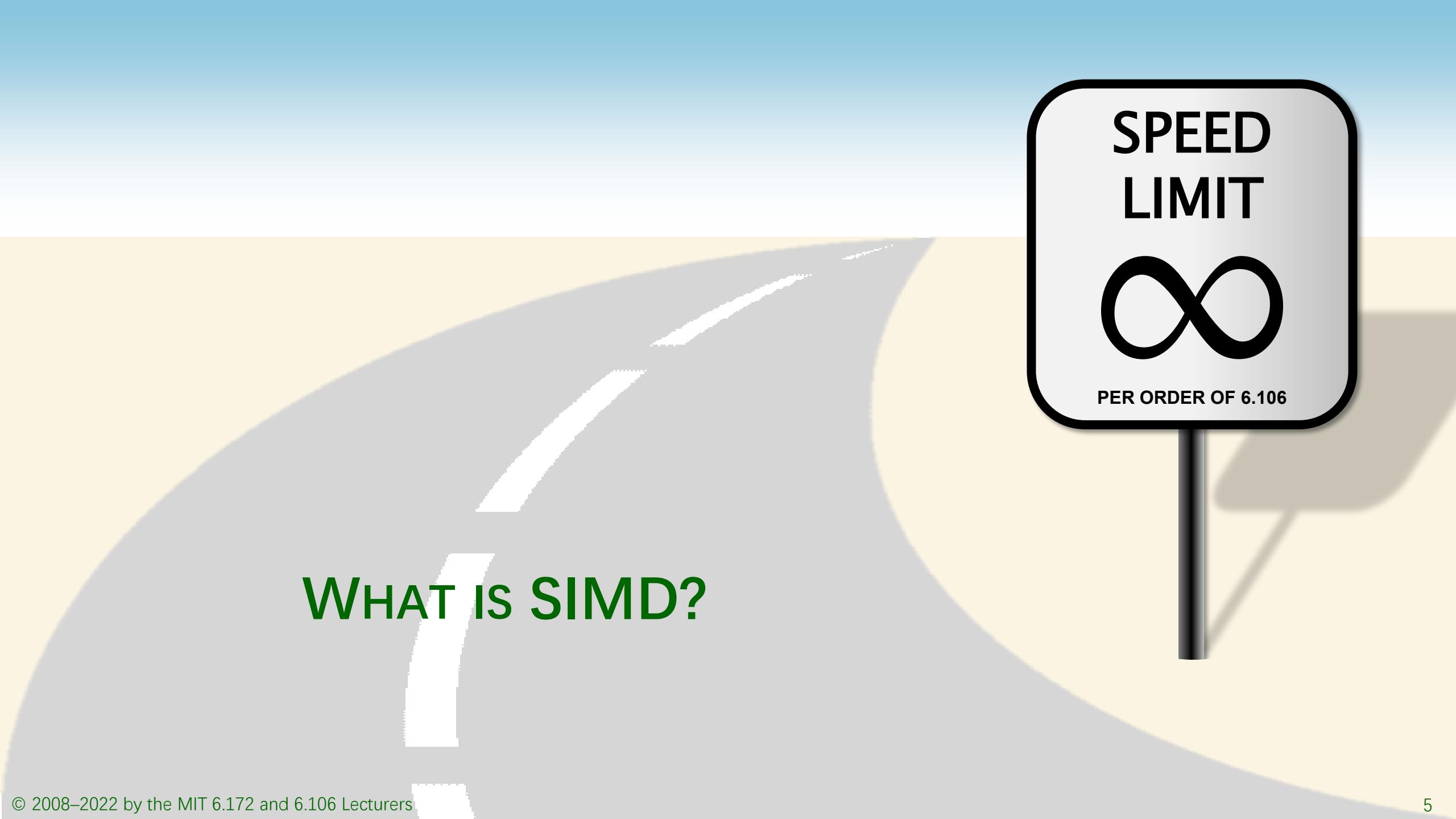
Required long vectors for performance

Seymour Cray

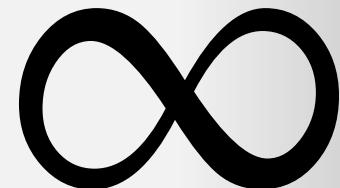
Multimedia Instructions

- Multimedia ISA extensions came in the 90's
- Multimedia instructions are
 - SIMD: Single Instruction Multiple Data
 - Integrated into the processor
 - Short and fast
 - No startup overhead
- Different register set
 - The multi-media register set
 - Extension of the original floating-point registers
 - Wide loads direct to the multimedia registers

| Vendor | ISA | Year | Bitwidth |
|--------|-------------|------|----------|
| Intel | MMX | 1996 | 64 bits |
| AMD | 3DNow! | 1998 | 64 bits |
| Intel | SSE | 1999 | 128 bits |
| Intel | SSE2 | 2001 | 128 bits |
| Intel | SSE3 | 2006 | 128 bits |
| Intel | AVX | 2008 | 256 bits |
| ARM | Neon | 2011 | 128 bits |
| Intel | AVX2 | 2013 | 256 bits |
| Intel | AVX512 | 2015 | 512 bits |
| Intel | AVX512-VNNI | 2021 | 512 bits |

A speed limit sign is positioned on the right side of a grey road with white dashed lines. The sign has a black border and a black infinity symbol in the center. The text "SPEED LIMIT" is at the top and "PER ORDER OF 6.106" is at the bottom.

SPEED
LIMIT

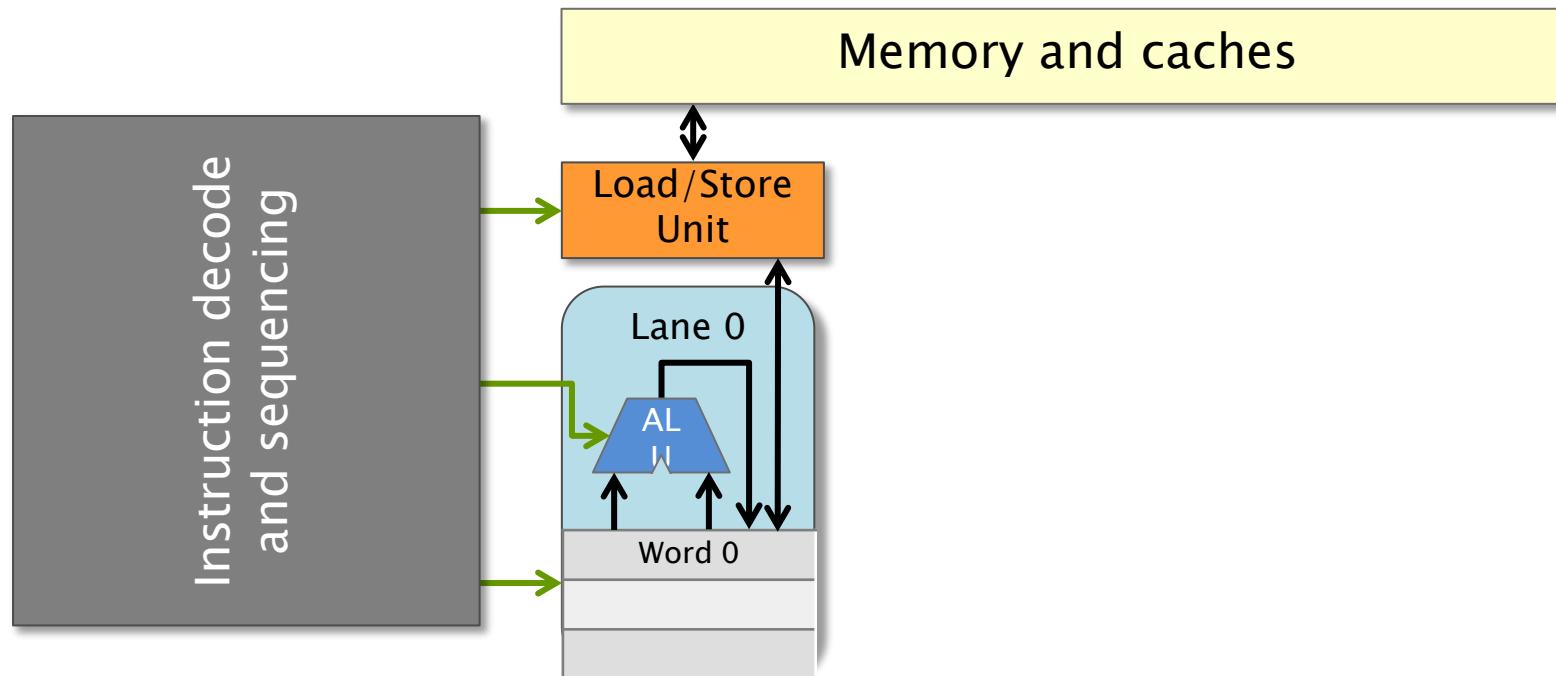


PER ORDER OF 6.106

WHAT IS SIMD?

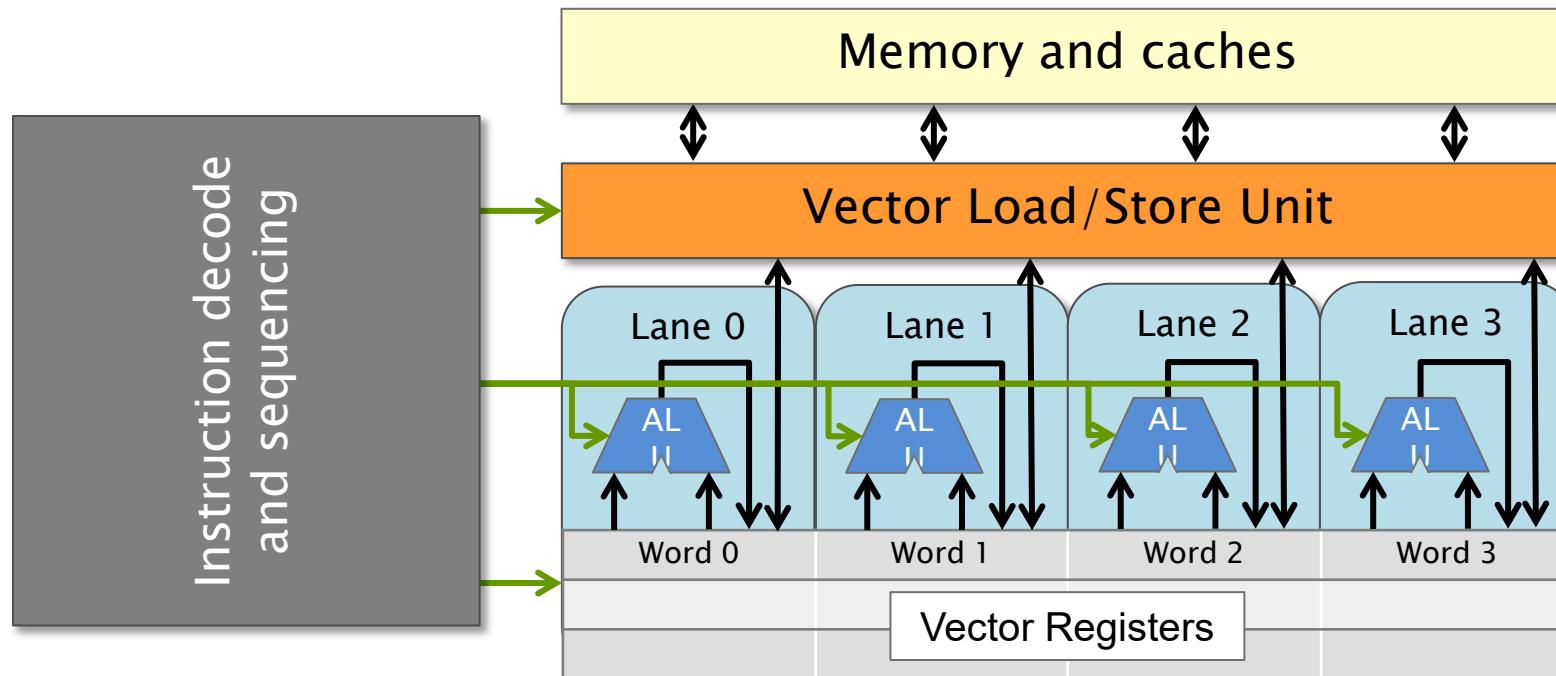
Scalar Hardware

Modern microprocessors spend a lot of real estate in instruction processing. The data paths and ALUs are relatively small.



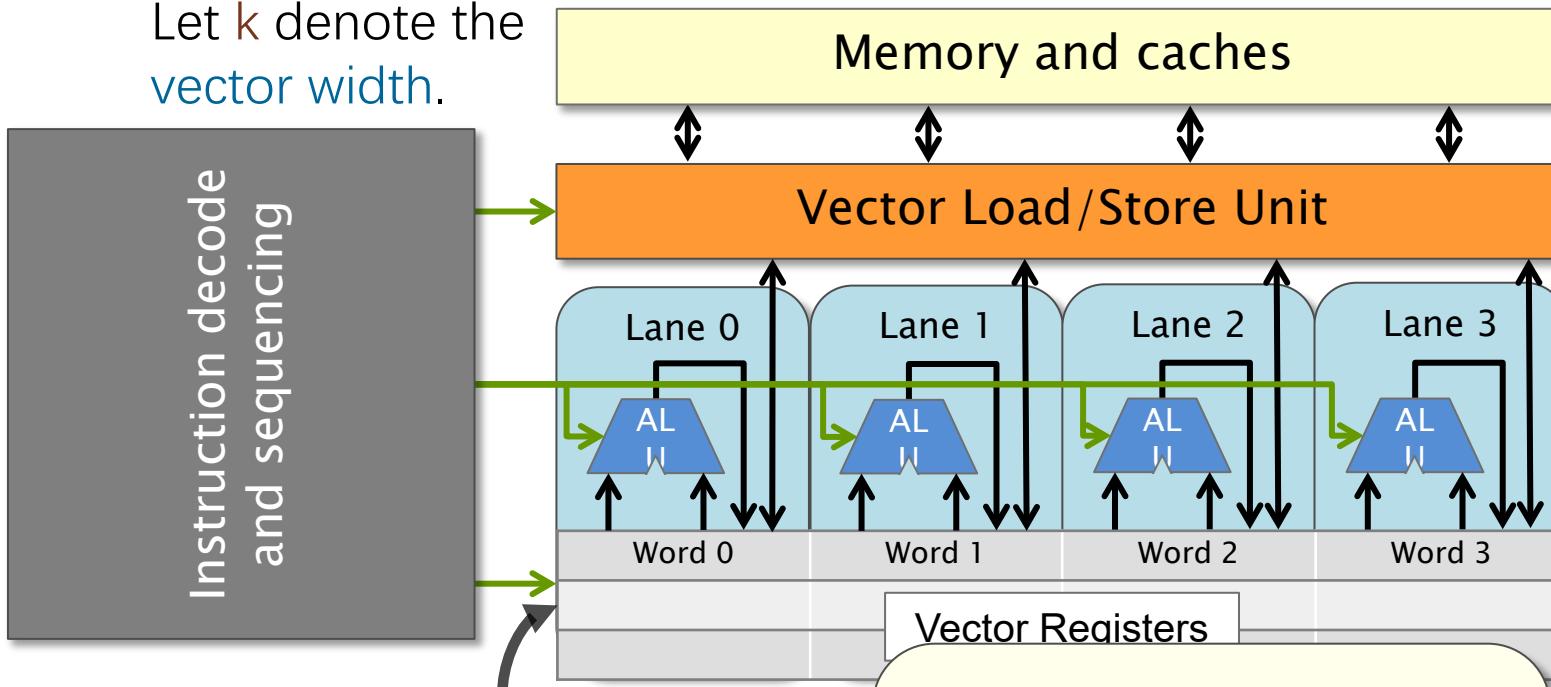
Vector Hardware

Modern microprocessors often incorporate vector hardware to process data in a single-instruction stream, multiple-data stream (SIMD) fashion.



A k -Wide Vector Unit

Let k denote the vector width.

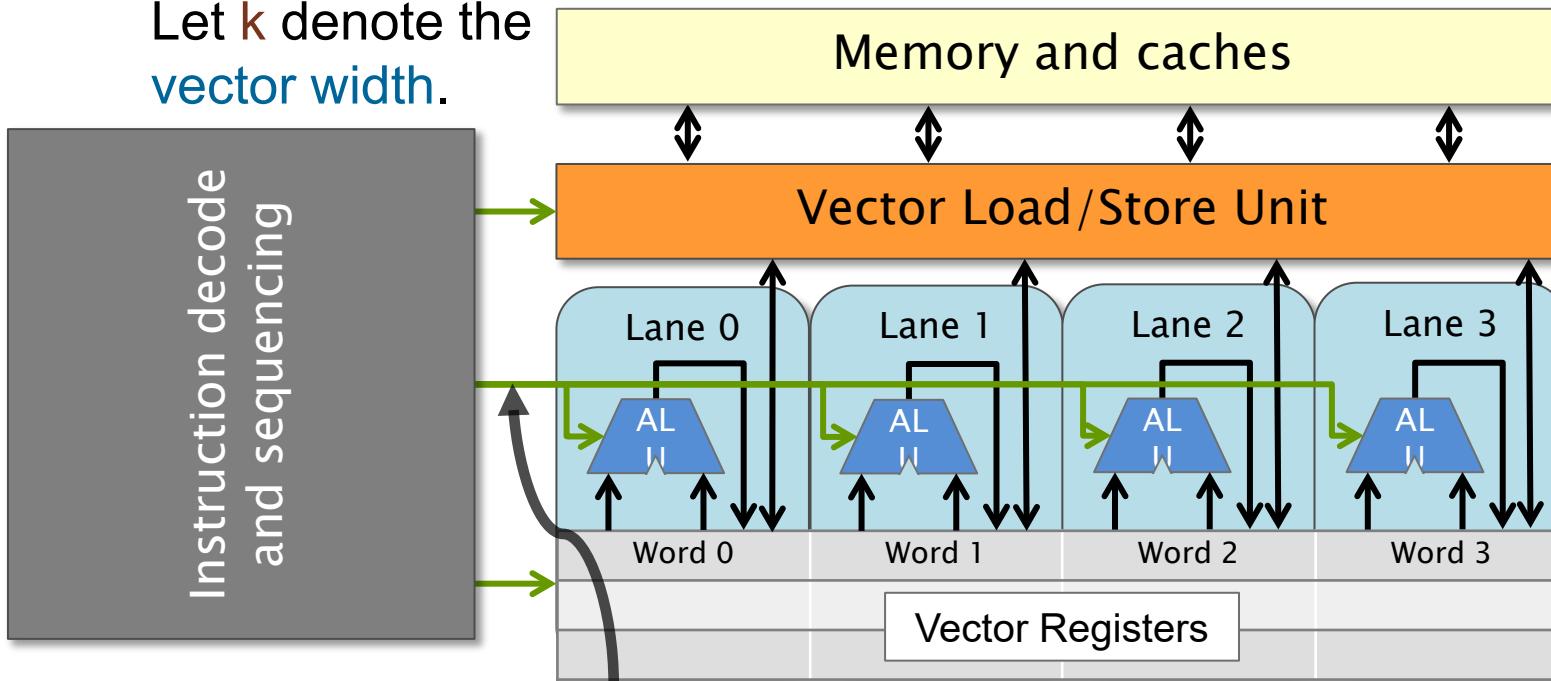


Each vector register holds k scalar integer or floating-point values.

The unit contains k vector lanes, each containing scalar integer or floating-point hardware.

A k -Wide Vector Unit

Let k denote the vector width.



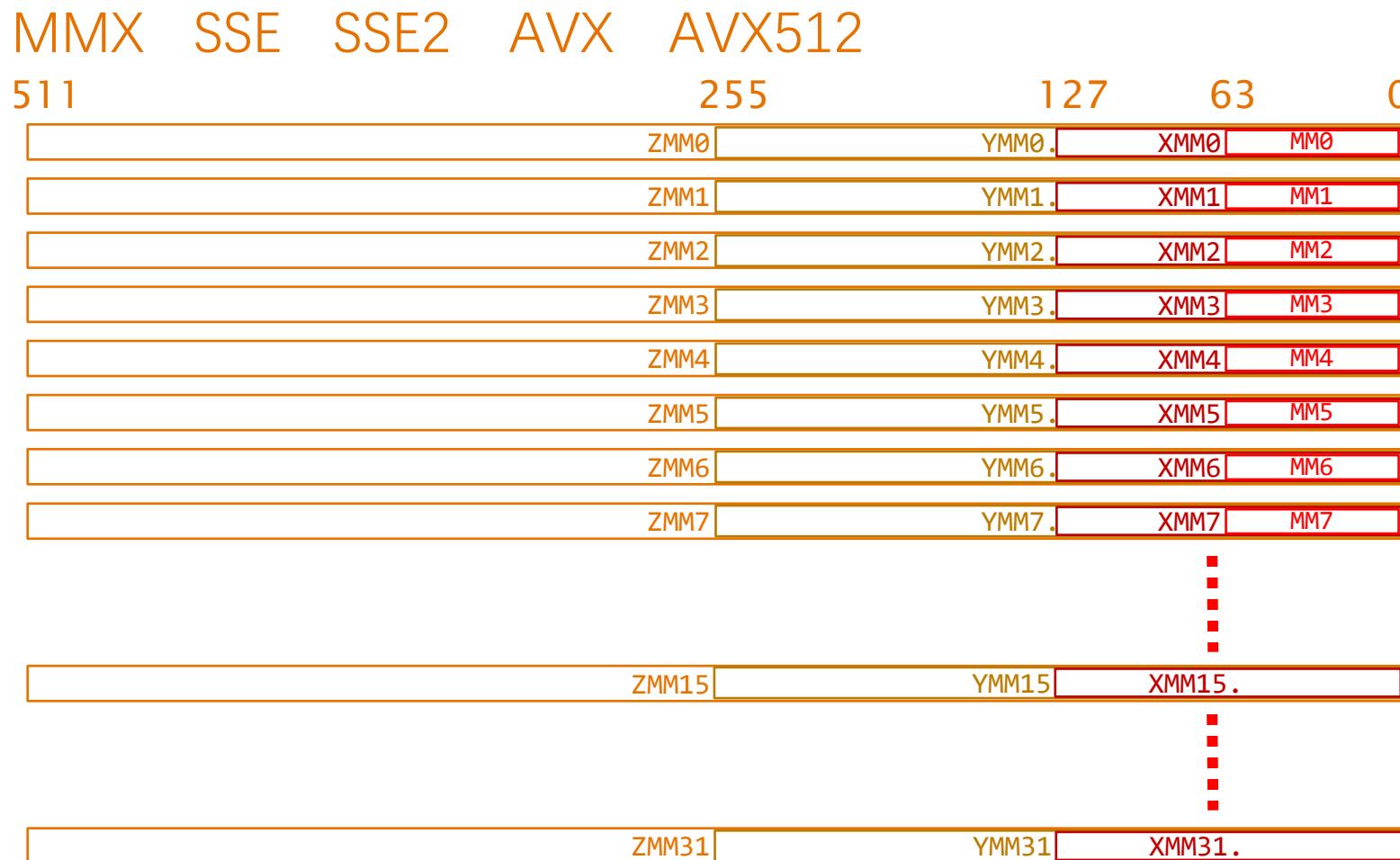
All vector lanes operate in lock-step and use the same instruction and control signals.

SIMD Instructions

- SIMD instructions typically operate in an elementwise fashion
 - The i -th element of one vector register only takes part in operations with the i -th element of other vector registers.
 - All lanes perform exactly the same operation on their respective elements of the vector.
 - Depending on the architecture, vector memory operands might need to be aligned, meaning their address must be a multiple of the vector width.
 - Some architectures support cross-lane operations, such as inserting or extracting subsets of vector elements, permuting (a.k.a., shuffling) the vector, scatter, or gather

Vector Register Aliasing

Different names, but same registers



Vector Register Aliasing

Each register can have different data types

SSE Data Types (16 XMM Registers)

| | | | | | |
|---------------------|-----------|-------|----------------|-------|------------------|
| <code>_m128</code> | Float | Float | Float | Float | 4x 32-bit float |
| <code>_m128d</code> | Double | | Double | | 2x 64-bit double |
| <code>_m128i</code> | B | B | B | B | 16x 8-bit byte |
| <code>_m128i</code> | short | short | short | short | 8x 16-bit short |
| <code>_m128i</code> | int | int | int | int | 4x 32bit integer |
| <code>_m128i</code> | long long | | long long | | 2x 64bit long |
| <code>_m128i</code> | | | doublequadword | | 1x 128-bit quad |

<https://www.codingame.com/playgrounds/283/sse-avx-vectorization/what-is-sse-and-avx>

Vector Register Aliasing

Each register can have different data types

SSE Data Types (16 XMM Registers)

| | | | | | |
|--------|----------------|-------|-----------|-------|------------------|
| _m128 | Float | Float | Float | Float | 4x 32-bit float |
| _m128d | Double | | Double | | 2x 64-bit double |
| _m128i | B | B | B | B | 16x 8-bit byte |
| _m128i | short | short | short | short | 8x 16-bit short |
| _m128i | int | int | int | int | 4x 32bit integer |
| _m128i | long long | | long long | | 2x 64bit long |
| _m128i | doublequadword | | | | 1x 128-bit quad |

AVX Data Types (16 YMM Registers)

| | | | | | | | | | |
|---------|--------|-------|--------|-------|--------|-------|--------|-------|------------------|
| _mm256 | Float | Float | Float | Float | Float | Float | Float | Float | 8x 32-bit float |
| _mm256d | Double | | Double | | Double | | Double | | 4x 64-bit double |

<https://www.codingame.com/playgrounds/283/sse-avx-vectorization/what-is-sse-and-avx>

AVX-512 has ZMM registers of 512 bits...

FLOATING POINT



Numbers, Numbers,...Numbers

| | bits | Smallest value | Biggest value |
|------------------------------|-----------|----------------------------|---------------------------------|
| Short | 16 | ± 1 | $\pm 32,768$ |
| Int | 32 | ± 1 | $\pm 2.147\dots \times 10^9$ |
| Long long | 64 | ± 1 | $\pm 9.223\dots \times 10^{18}$ |
| IEEE Single Precision | 64 | $\pm 1.2 \times 10^{-38}$ | $\pm 3.4 \times 10^{38}$ |
| IEEE Double Precision | 128 | $\pm 2.2 \times 10^{-308}$ | $\pm 1.8 \times 10^{308}$ |
| Minfloats | 8 | ± 0.125 | $\pm 15,360$ |

- IEEE 754 single precision



$$(-1)^s \times M \times 2^{E-127}$$

- IEEE 754 double precision



$$(-1)^s \times M \times 2^{E-1023}$$

Extracted from the lecture notes by Jeremy R. Johnson,
Anatole D. Ruslanov and William M. Mongan

Issues with Floating Point

$$a + (b + c) \stackrel{?}{=} (a + b) + c$$

Let floats $a = -2.7 \times 10^{23}$, $b = 2.7 \times 10^{23}$, and $c = 1.0$

$$a + (b + c) = -2.7 \times 10^{23} + (2.7 \times 10^{23} + 1.0) = -2.7 \times 10^{23} + 2.7 \times 10^{23} = 0.0$$

$$(a + b) + c = (-2.7 \times 10^{23} + 2.7 \times 10^{23}) + 1.0 = 0.0 + 1.0 = 1.0$$

- Compilers are conservative unless told otherwise
- Use the `fast-math` flag
- In `gcc -ffast-math` means disregarding a lot of rules
 - `-fno-math-errno`,
 - `-ffinite-math-only`,
 - `-fno-signaling-nans`,
 - `-fexcess-precision=fast`,
 - `-fno-trapping-math`,
 - `-freciprocal-math`
 - `-funsafe-math-optimizations`,
 - `-fno-rounding-math`,
 - `-fcx-limited-range`,
 - `-fno-signed-zeros`,
 - `-fassociative-math`,

X86 MULTI-MEDIA INSTRUCTIONS



Instruction Naming Convention

`_mm<vec-width>_<op>_<scalar-ty>`

- $\langle \text{vec-width} \rangle$
 - 64/128/256/512
- $\langle \text{scalar-ty} \rangle$
 - ss – one single precision floating point (scalar)
 - sd – one double precision floating point (scalar)
 - ps - packed single precision floating point (vector)
 - pd - packed double precision floating point (vector)
 - epi8/epi16/epi32/epi64 - packed signed integers (vector)
- Examples
 - 8-wide float addition: `_mm256_add_ps`
 - 16-wide absolute value of 16-bit ints: `_mm256_abs_epi16`
 - 4-wide float subtraction: `_mm_sub_ps` (128 is omitted)

Categories of Vector Instructions

1. Arithmetic and Logic
2. Compare
3. Load/Store
4. Shuffle (Swizzle)
5. Masks
6. Non-SIMD

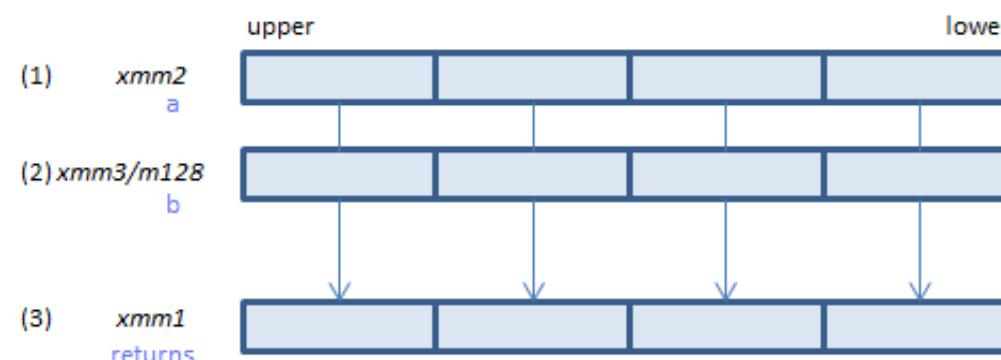
1. Arithmetic and Logic

- Types

- add, sub, mul, div, and, or, andnot, xor

★ Example

```
__m128i _mm_add_epi32 (__m128i a, __m128i b) {  
    for j := 0 to 3 {  
        i := j*32  
        dst[i+31:i] := a[i+31:i] + b[i+31:i]  
    }  
    return dst  
}
```



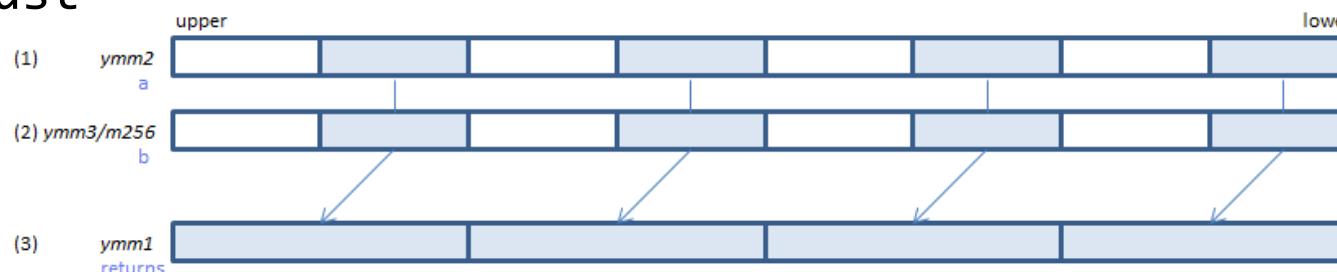
1. Arithmetic and Logic

- Types

- mul

★ Example

```
_m256i _mm_mul_epi32 (_m256i a, _m256i b) {  
    for j := 0 to 3 {  
        i := j*64  
        dst[i+63:i] := a[i+31:i] * b[i+31:i]  
    }  
    dst[MAX:256] := 0  
    return dst  
}
```



2. Compare

- Types

- "Normal" comparisons: sets lanes to ones if true
- AVX-512 comparisons: sets *mask* register (%k0 - %k7)

★ Example

```
__m128 _mm_cmp_ps (__m128 a, __m128 b, const int imm8) {
    CASE (imm8[4:0]) {
        0: OP := _CMP_EQ_OQ
        1: OP := _CMP_LT_OS
        ...
        31: OP := _CMP_TRUE_US
    }
    for j := 0 to 3 {
        i := j*32
        dst[i+31:i] := ( a[i+31:i] OP b[i+31:i] ) ? 0xFFFFFFFF : 0
    }
    dst[MAX:128] := 0
}
```

3. Load/Store

- Types

- Aligned load/store (e.g., vmovaps)
- Unaligned load/store (e.g., vmovups)
- Streaming load/store (e.g., vmovntps)
 - Streaming writes are 2x faster than normal writes

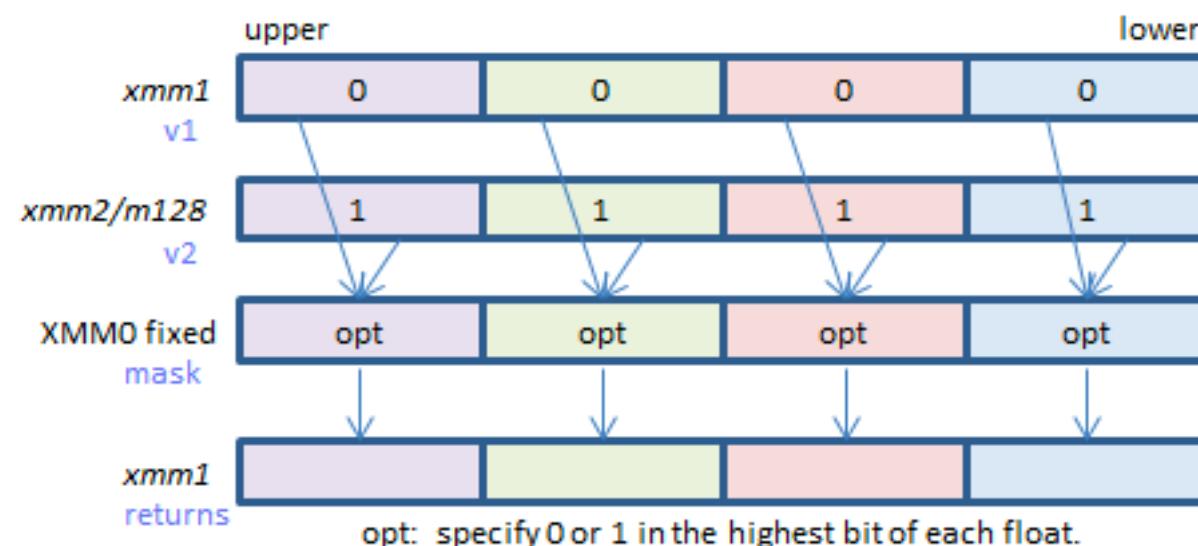
★ Example

```
_m256 _mm256_loadu_ps (float const * mem_addr) {  
    dst[255:0] := MEM[mem_addr+255:mem_addr]  
    dst[MAX:256] := 0  
}
```

4. Shuffle: Blending

★ Example

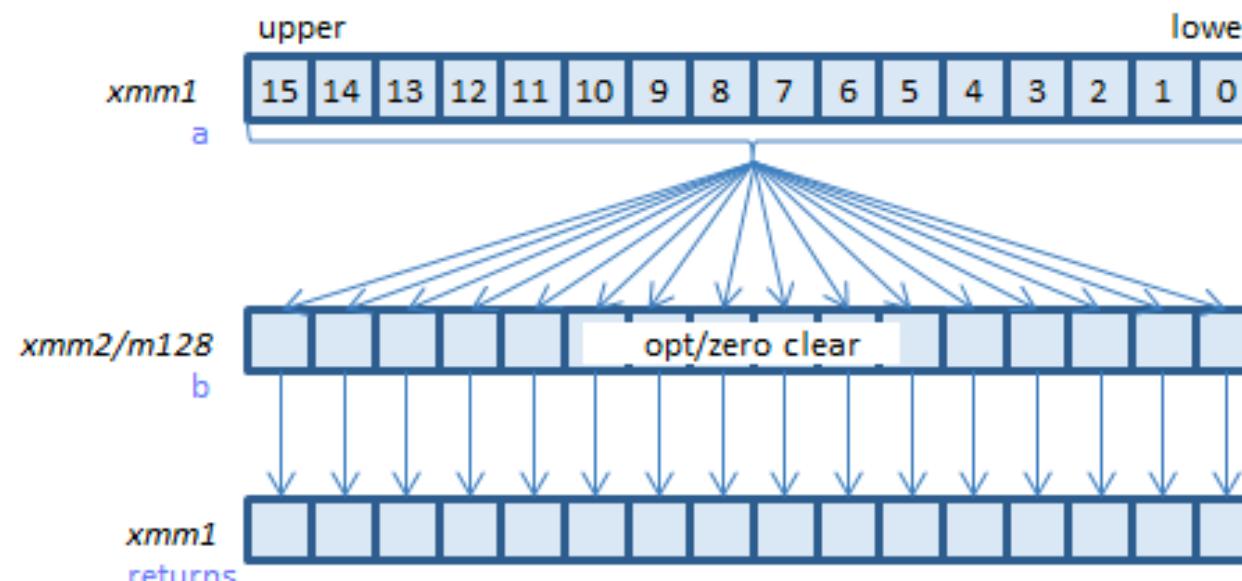
```
__m128 __mm_blendv_ps(__m128 v1, __m128 v2, __m128 mask)
```



4. Shuffle: all-to-all shuffle

★ Example

`__m128i _mm_shuffle_epi8(__m128i a, __m128i b)`



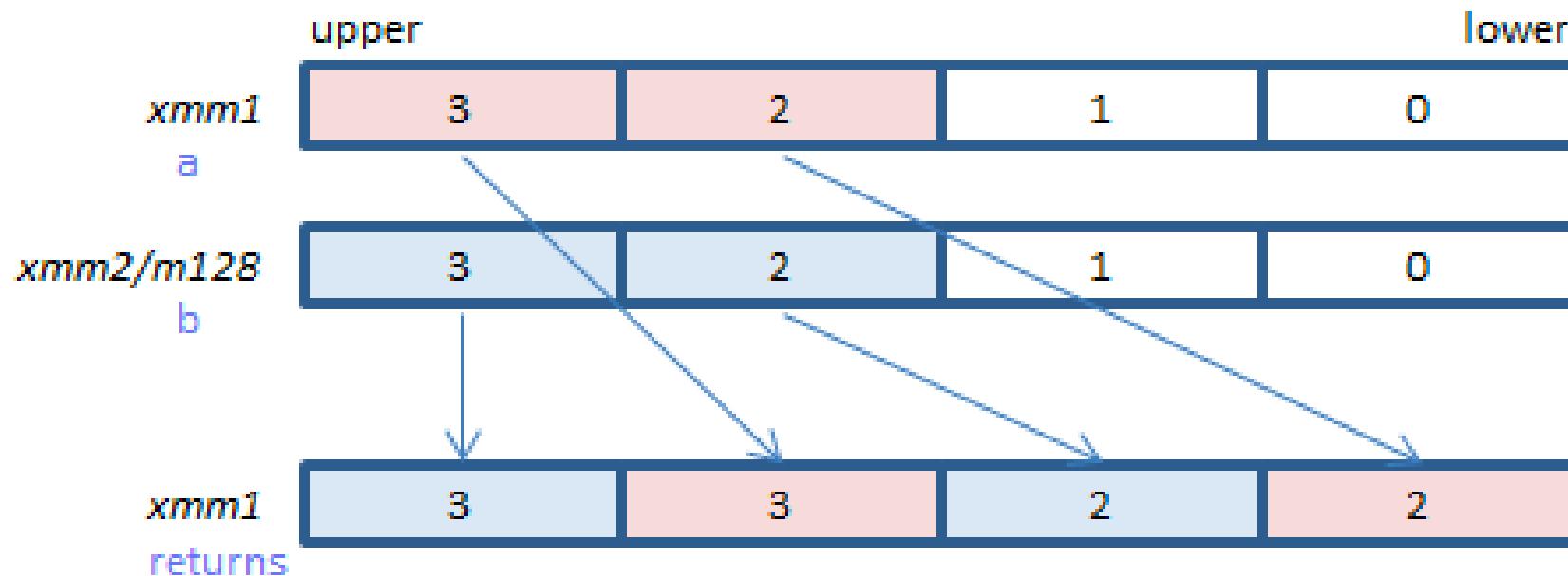
each byte of xmm2/m128:
bit 7 == 0 specifies copying, bit 3:0 specifies 0 to 15
bit 7 == 1 specifies zero clearing

4. Shuffle: unpack

- Unpackhi
- Unpacklo

★ Example

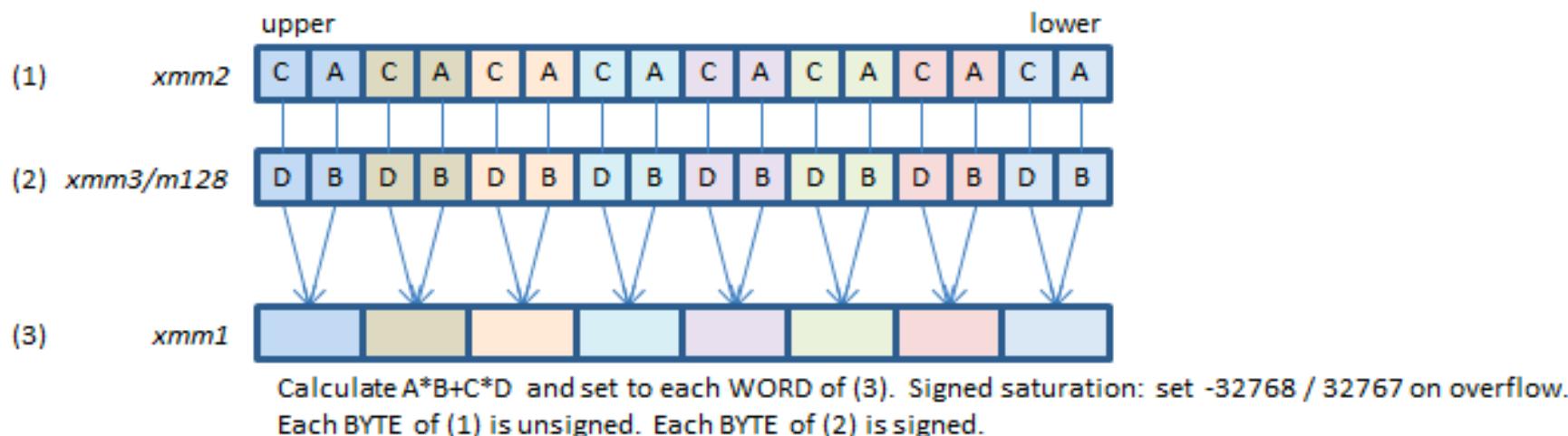
```
__m128 __mm_unpackhi_ps(__m128 a, __m128 b)
```



5. Non-SIMD: dot product instructions

★ Example

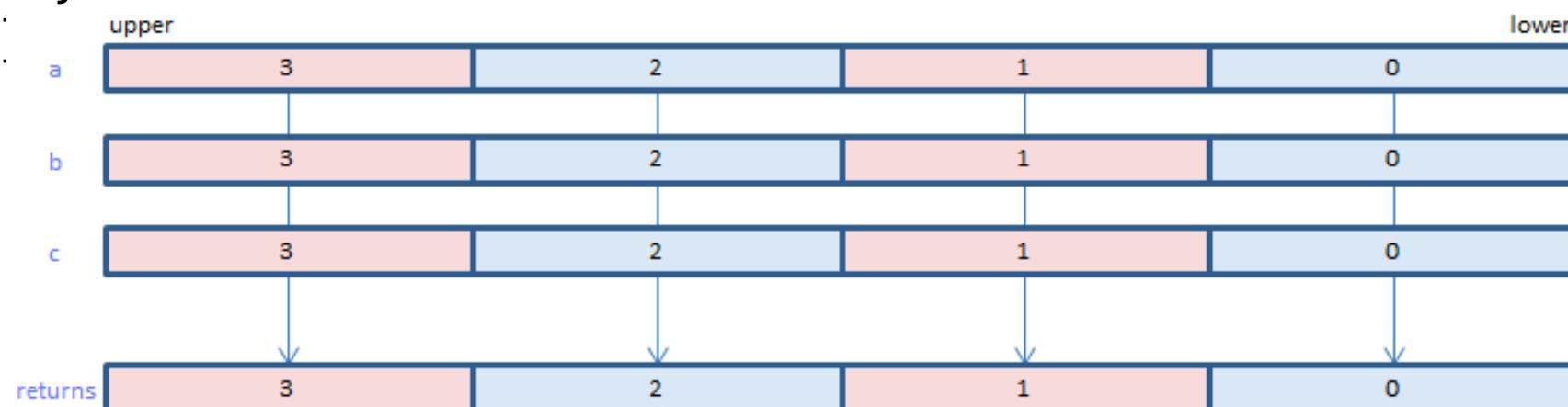
`__m128i _mm_maddubs_epi16(__m128i a, __m128i b)`



5. Non-SIMD: addsub

★ Example

```
__m256d __mm256_addsub_pd(__m256d a, __m256d b) {  
    for j := 0 to 3 {  
        i := j*64  
        if ((j & 1) == 0)  
            dst[i+63:i] := a[i+63:i] - b[i+63:i]  
        else  
            dst[i+63:i] := a[i+63:i] + b[i+63:i]  
    }  
}
```



And a lot more instructions..

Google "Intel Intrinsics Guide"

Intel® Intrinsics Guide

Updated
08/10/2022 Version
3.6.3

Instruction Set

- MMX
- SSE
- SSE2
- SSE3
- SSE3
- SSE4.1
- SSE4.2
- AVX
- AVX2
- FMA
- AVX_VNNI
- AVX-512
- KNC
- AMX
- SVML
- Other

Categories

- Application-Targeted
- Arithmetic
- Bit Manipulation
- Cast
- Compare
- Convert
- Cryptography
- Elementary Math Functions
- General Support
- Load
- Logical
- Mask
- Miscellaneous
- Move
- OS-Targeted
- Probability/Statistics
- Random
- Set
- Shift

[Give Feedback](#)

| Search Intel Intrinsics | |
|---|---------------|
| void _mm_2intersect_ep32 (_m128i a, _m128i b, _mmask8* k1, _mmask8* k2) | vp2intersecte |
| void _mm256_2intersect_ep32 (_m256i a, _m256i b, _mmask8* k1, _mmask8* k2) | vp2intersecte |
| void _mm512_2intersect_ep32 (_m512i a, _m512i b, _mmask16* k1, _mmask16* k2) | vp2intersecte |
| void _mm_2intersect_ep64 (_m128i a, _m128i b, _mmask8* k1, _mmask8* k2) | vp2intersecte |
| void _mm256_2intersect_ep64 (_m256i a, _m256i b, _mmask8* k1, _mmask8* k2) | vp2intersecte |
| void _mm512_2intersect_ep64 (_m512i a, _m512i b, _mmask8* k1, _mmask8* k2) | vp2intersecte |
| _m512i _mm512_4dpwssd_epi32 (_m512i src, _m512i a0, _m512i a1, _m512i a2, _m512i a3, _m128i * b) | vp4dpwssd |
| _m512i _mm512_mask_4dpwssd_epi32 (_m512i src, _mmask16 k, _m512i a0, _m512i a1, _m512i a2, _m512i a3, _m128i * b) | vp4dpwssd |
| _m512i _mm512_maskz_4dpwssd_epi32 (_mmask16 k, _m512i src, _m512i a0, _m512i a1, _m512i a2, _m512i a3, _m128i * b) | vp4dpwssd |
| _m512i _mm512_4dpwssds_epi32 (_m512i src, _m512i a0, _m512i a1, _m512i a2, _m512i a3, _m128i * b) | vp4dpwssds |
| _m512i _mm512_mask_4dpwssds_epi32 (_m512i src, _mmask16 k, _m512i a0, _m512i a1, _m512i a2, _m512i a3, _m128i * b) | vp4dpwssds |
| _m512i _mm512_maskz_4dpwssds_epi32 (_mmask16 k, _m512i src, _m512i a0, _m512i a1, _m512i a2, _m512i a3, _m128i * b) | vp4dpwssds |
| _m512 _mm512_4fmadd_ps (_m512 src, _m512 a0, _m512 a1, _m512 a2, _m512 a3, _m128 * b) | v4fmaddps |
| _m512 _mm512_mask_4fmadd_ps (_m512 src, _mmask16 k, _m512 a0, _m512 a1, _m512 a2, _m512 a3, _m128 * b) | v4fmaddps |
| _m512 _mm512_maskz_4fmadd_ps (_mmask16 k, _m512 src, _m512 a0, _m512 a1, _m512 a2, _m512 a3, _m128 * b) | v4fmaddps |
| _m128 _mm_4fmadd_ss (_m128 src, _m128 a0, _m128 a1, _m128 a2, _m128 a3, _m128 * b) | v4fmaddss |
| _m128 _mm_mask_4fmadd_ss (_m128 src, _mmask8 k, _m128 a0, _m128 a1, _m128 a2, _m128 a3, _m128 * b) | v4fmaddss |

Synopsis

```
_m128 _mm_mask_4fmadd_ss (_m128 src, _mmask8 k, _m128 a0, _m128 a1, _m128 a2, _m128 a3, _m128 * b)
#include <i mmintrin.h>
Instruction: v4fmaddss xmm (k), xmm, m128
CPUID Flags: AVX512_4FMAPS
```

Description

Multiply the lower single-precision (32-bit) floating-point elements specified in 4 consecutive operands `a0` through `a3` by corresponding element in `b`, accumulate with the lower element in `a`, and store the result in the lower element of `dst` using writemask `k` (the element is copied from `a` when mask bit 0 is not set).

Operation

```
dst[127:0] := src[127:0]
IF k[0]
  FOR m := 0 to 3
    add := b + m * 32
    dst.fp32[0] := dst.fp32[0] + a(m).fp32[0] * Cast_FP32(MEM[addr+31:addr])
  ENDFOR
FI
dst[MAX:128] := 0
```

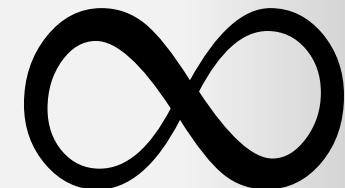
_m128 _mm_maskz_4fmadd_ss (_mmask8 k, _m128 src, _m128 a0, _m128 a1, _m128 a2, _m128 a3, _m128 * b)

_m512 _mm512_4fmadd_ps (_m512 src, _m512 a0, _m512 a1, _m512 a2, _m512 a3, _m128 * b)

Over 7000 Intrinsics!



SPEED
LIMIT



PER ORDER OF 6.106

LET IT BE VECTORIZED
(BY THE COMPILER)

Vectorizing Compiler

- Two types of Vectorizers
 1. Loop Vectorizer
 2. Superword Level Parallelism(SLP) Vectorizer

Loop Vectorization

```
for (int i = 0; i < n; i++) {  
    tb = b[i];  
    tc = c[i];  
    t = tb + tc;  
    a[i] = t;  
}
```

* Randy Allen and Ken Kennedy. Automatic Translation
of FORTRAN Programs to Vector Form. TOPLAS 1987

Loop Vectorization

```
for (int i = 0; i < n; i++) {  
    tb = b[i];  
    tc = c[i];  
    t = tb + tc;  
    a[i] = t;  
}
```

```
for (int i = 0; i < n; i += 2) {  
    tb = b[i:i+2];  
    tc = c[i:i+2];  
    t = tb + tc;  
    a[i:i+2] = t;  
}
```

The entire loop body is vectorized.

In order to do this the loop has to be ‘parallelizable’

Non trivial analysis by the compiler

* Randy Allen and Ken Kennedy. Automatic
Translation of FORTRAN Programs to Vector Form.
TOPLAS 1987

SLP Vectorization

```
for (int i = 0; i < n; i++) {  
    tb = b[i];  
    tc = c[i];  
    t = tb + tc;  
    a[i] = t;  
}
```

```
for (int i = 0; i < n; i += 2) {  
    tb = b[i];  
    tc = c[i];  
    t = tb + tc;  
    a[i] = t;  
    tb2 = b[i+1];  
    tc2 = c[i+1];  
    t2 = tb2 + tc2;  
    a[i] = t2;  
}
```

First Unroll the loop

SLP Vectorization

```
for (int i = 0; i < n; i++) {  
    tb   = b[i];  
    tc   = c[i];  
    t    = tb + tc;  
    a[i] = t;  
}
```

```
for (int i = 0; i < n; i += 2) {  
    tb   = b[i];  
    tb2  = b[i+1];  
    tc   = c[i];  
    tc2  = c[i+1];  
    t    = tb + tc;  
    t2   = tb2 + tc2;  
    a[i] = t;  
    a[i] = t2;  
}
```

Next, Group Isomorphic Instructions

SLP Vectorization

```
for (int i = 0; i < n; i++) {  
    tb = b[i];  
    tc = c[i];  
    t = tb + tc;  
    a[i] = t;  
}
```

```
for (int i = 0; i < n; i += 2) {  
    tb = b[i]; }  
    tb2 = b[i+1]; }  
    tc = c[i]; }  
    tc2 = c[i+1]; }  
    t = tb + tc; }  
    t2 = tb2 + tc2; }  
    a[i] = t; }  
    a[i] = t2; }  
}
```

```
tb      = b[i:i+1];  
tc      = c[i:i+1];  
t       = tb + tc;  
a[i:i+1] = t;
```

Finally, pack into vector instructions

SLP Vectorization

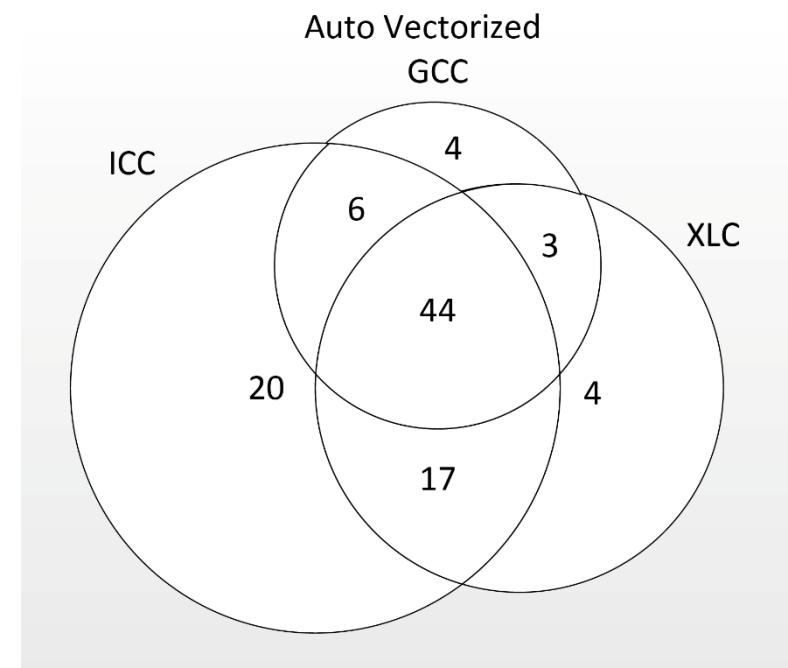
```
for (int i = 0; i < n; i++) {  
    tb = b[i];  
    tc = c[i];  
    t = tb + tc;  
    a[i] = t;  
}
```

```
tb      = b[i:i+1];  
tc      = c[i:i+1];  
t       = tb + tc;  
a[i:i+1] = t;
```

The loop body can be partially vectorized.
No loop analysis is needed, only the loop body.
Need to unroll to be effective

How good are the Vectorizers?

- Vectorizers are good, but, not ubiquitous
 - Some complex code get vectorized
 - But... some very simple code does not!
 - Need to pay attention



S. Maleki, Y. Gao, M. J. Garzarn, T. Wong and D. A. Padua,
"An Evaluation of Vectorizing Compilers," 2011 International
Conference on Parallel Architectures and Compilation
Techniques, 2011

What makes it hard for the compiler to vectorize

- 1 Loop with unknown trip count
- 2 Cannot prove independence
- 3 Loop carried dependences
- 4 Reductions
- 5 Control-flow
- 6 Non-inner loop
- 7 Non-contiguous data
- 8 Cost model failures
- 9 Vectorization of function calls

See <https://llvm.org/docs/Vectorizers.html> for more info.

What makes it hard for the compiler to vectorize

Go to www.godbolt.org and try -O3 -mavx512vnni -ffast-math -fno-unroll-loops

- | | | |
|---|---------------------------------|--|
| 1 | Loop with unknown trip count | https://www.godbolt.org/z/oh15M6baK |
| 2 | Cannot prove independence | https://www.godbolt.org/z/eYYadjraP |
| 3 | Loop carried dependences | https://www.godbolt.org/z/6WEcrMaM4 |
| 4 | Reductions | https://www.godbolt.org/z/q7s xv8cEa |
| 5 | Control-flow | https://www.godbolt.org/z/5v66a8Mq6 |
| 6 | Non-contiguous data | https://www.godbolt.org/z/bGc1GvThM https://www.godbolt.org/z/31bzd4PxY |
| 7 | Cost model failures | https://godbolt.org/z/b3cYo6388 |
| 8 | Vectorization of function calls | https://www.godbolt.org/z/ra8Esjx9 |
| 9 | Different hardware versions | https://www.godbolt.org/z/dd53r1Mf7 |

See <https://llvm.org/docs/Vectorizers.html> for more info.

1. Loop with unknown trip count

```
void bar(float * A, float K, int start, int end) {  
    for (int i = start; i < end; ++i)  
        A[i] = K;  
}
```

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

```
void bar(float * A, float K) {  
    for (int i = 0; i < 1024; ++i)  
        A[i] = K;  
}
```

<https://www.godbolt.org/z/oh15M6baK>

1. Loop with unknown trip count

```
1 bar: # @bar
2   cmp    esi, edx
3   jge    .LBB0_8
4   movsxd r8, esi
5   movsxd r10, edx
6   mov    r9, r10
7   sub    r9, r8
8   cmp    r9, 8
9   jae    .LBB0_3
10  mov    rsi, r8
11  jmp    .LBB0_14
12 .LBB0_3:
13  cmp    r9, 16
14  jae    .LBB0_9
15  xor    edx, edx
16  jmp    .LBB0_5
17 .LBB0_9:
18  mov    rdx, r9
19  and    rdx, -16
20  vbroadcastss zmm1, xmm0
21  lea    rcx, [rdi + 4*r8]
22  xor    esi, esi
23 .LBB0_10: # =>This Inner Loop Header: Depth=22
24  vmovups zmmword ptr [rcx + 4*rsi], zmm1
25  add    rsi, 16
26  cmp    rdx, rsi
27  jne    .LBB0_10
28  cmp    r9, rdx
29  je     .LBB0_8
30  test   r9b, 8
31  jne    .LBB0_5
32  add    rdx, r8
33  mov    rsi, rdx
34  jmp    .LBB0_14
35 .LBB0_5:
36  mov    rcx, r9
37  and    rcx, -8
38  lea    rsi, [rcx + 8]
39  vbroadcastss ymm1, xmm0
40  lea    rax, [rdi + 4*r8]
41 .LBB0_6: # =>This Inner Loop Header: Depth=40
42  vmovups ymmword ptr [rax + 4*rdx], ymm1
43  add    rdx, 8
44  cmp    rcx, rdx
45  jne    .LBB0_6
46  cmp    r9, rcx
47  je     .LBB0_8
48 .LBB0_14: # =>This Inner Loop Header: Depth=47
49  vmovss dword ptr [rdi + 4*rsi], xmm0
50  inc    rsi
51  cmp    r10, rsi
52  jne    .LBB0_14
53 .LBB0_8:
54  vzeroupper
55  ret
```

```
1 bar: # @bar
2   test   edx, edx
3   jle    .LBB0_13
4   mov    eax, edx
5   cmp    edx, 8
6   jae    .LBB0_3
7   xor    ecx, ecx
8   jmp    .LBB0_12
9 .LBB0_3:
10  cmp    edx, 16
11  jae    .LBB0_5
12  xor    ecx, ecx
13  jmp    .LBB0_9
14 .LBB0_5:
15  mov    ecx, eax
16  and    ecx, -16
17  vbroadcastss zmm1, xmm0
18  lea    rdx, [4*rax]
19  and    rdx, -64
20  xor    esi, esi
21 .LBB0_6: # =>This Inner Loop Header: Depth=1
22  vmovups zmmword ptr [rdi + rsi], zmm1
23  add    rsi, 64
24  cmp    rdx, rsi
25  jne    .LBB0_6
26  cmp    rcx, rax
27  je     .LBB0_13
28  test   al, 8
29  je     .LBB0_12
30 .LBB0_9:
31  mov    rdx, rcx
32  mov    ecx, eax
33  and    ecx, -8
34  vbroadcastss ymm1, xmm0
35 .LBB0_10: # =>This Inner Loop Header: Depth=1
36  vmovups ymmword ptr [rdi + 4*rdx], ymm1
37  add    rdx, 8
38  cmp    rcx, rdx
39  jne    .LBB0_10
40  cmp    rcx, rax
41  je     .LBB0_13
42 .LBB0_12: # =>This Inner Loop Header: Depth=1
43  vmovss dword ptr [rdi + 4*rcx], xmm0
44  inc    rcx
45  cmp    rax, rcx
46  jne    .LBB0_12
47 .LBB0_13:
48  vzeroupper
49  ret
```

```
1 bar: # @bar
2  vbroadcastss zmm0, xmm0
3  xor    eax, eax
4 .LBB0_1: # =>This Inner Loop Header: Depth=1
5  vmovups zmmword ptr [rdi + 4*rax], zmm0
6  add    rax, 16
7  cmp    rax, 1024
8  jne    .LBB0_1
9  vzeroupper
10 ret
```

2. Cannot Prove Independence

```
void bar(float * A, float * B) {  
    for (int i = 0; i < 4; ++i)  
        A[i] += B[i];  
}
```

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |
|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|

2. Cannot Prove Independence

```
void bar(float * A, float * B) {  
    for (int i = 0; i < 4; ++i)  
        A[i] += B[i];  
}
```

The diagram illustrates the execution of the `bar` function. Two red arrows originate from the `i < 4` condition in the `for` loop and point to the values 10 and 20 in the array. The array is represented as a horizontal row of 16 orange boxes, each containing a numerical value.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |
|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|

2. Cannot Prove Independence

```
void bar(float * A, float * B) {  
    for (int i = 0; i < 4; ++i)  
        A[i] += B[i];  
}
```

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |
|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|

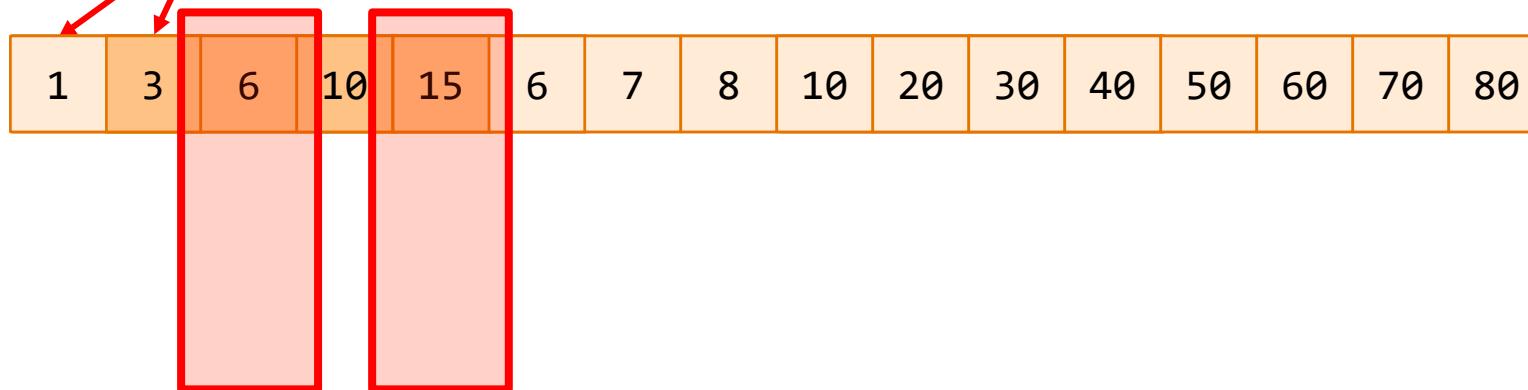
2. Cannot Prove Independence

```
void bar(float * A, float * B) {  
    for (int i = 0; i < 4; ++i)  
        A[i] += B[i];  
}
```

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |
|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|

2. Cannot Prove Independence

```
void bar(float * A, float * B) {  
    for (int i = 0; i < 4; ++i)  
        A[i] += B[i];  
}
```



- Vector Results \neq Scalar Results
- Dependence violated

2. Cannot Prove Independence

```
void bar(float * A, float * B) {  
    for (int i = 0; i < 1024; ++i)  
        A[i] += B[i];  
}
```

LLVM can still vectorize!

- How?

By checking if the address ranges overlap

- Check the ranges first
- If overlapping, use scalar implementation
- If not overlapping, use the vectorized implementation

However, Can save all these hassles by….

```
void bar(float * restrict A, float * restrict B) {  
    for (int i = 0; i < 1024; ++i)  
        A[i] += B[i];  
}
```

2. Cannot Prove Independence

```
1 √ bar:                                # @bar
2     mov    rax, rdi
3     sub    rax, rsi
4     cmp    rax, 255
5     ja     .LBB0_3
6     xor    eax, eax
7 √ .LBB0_2:                                # =>This Inner Loop Header: Depth=1
8     vaddss xmm1, xmm0, dword ptr [rsi + 4*rax]
9     vmovss dword ptr [rdi + 4*rax], xmm1
10    inc   rax
11    cmp   rax, 1024
12    jne   .LBB0_2
13    jmp   .LBB0_5
14 √ .LBB0_3:
15     vbroadcastss zmm0, xmm0
16     xor    eax, eax
17 √ .LBB0_4:                                # =>This Inner Loop Header: Depth=1
18     vaddps zmm1, zmm0, zmmword ptr [rsi + 4*rax]
19     vmovups zmmword ptr [rdi + 4*rax], zmm1
20     add   rax, 16
21     cmp   rax, 1024
22     jne   .LBB0_4
23 √ .LBB0_5:
24     vzeroupper
25     ret
```

```
1 bar:                                # @bar
2     vbroadcastss zmm0, xmm0
3     xor    eax, eax
4 .LBB0_1:                                # =>This Inner Loop Header: Depth=1
5     vaddps zmm1, zmm0, zmmword ptr [rsi + 4*rax]
6     vmovups zmmword ptr [rdi + 4*rax], zmm1
7     add   rax, 16
8     cmp   rax, 1024
9     jne   .LBB0_1
10    vzeroupper
11    ret
```

3. Loop Carried Dependence

```
void bar(float * restrict A, int n) {
    for (int i = 1; i < n; ++i)
        A[i] += A[i-1];      1024
    }                      A[i-2];
                           A[i-100];
```

```
void bar(float * restrict A, int n) {
    for (int i = 0; i < n-1; ++i)
        A[i] += A[i+1];
    }
```

<https://www.godbolt.org/z/6WEcrMaM4>

3. Loop Carried Dependence

```
1 bar:                                # @bar
2     vmovss  xmm0, dword ptr [rdi - 4]    # xmm0 = mem[0],zero,zero,zero
3     xor     eax, eax
4 .LBB0_1:                           # =>This Inner Loop Header: Depth=1
5     vaddss  xmm0, xmm0, dword ptr [rdi + 4*rax]
6     vmovss  dword ptr [rdi + 4*rax], xmm0
7     inc     rax
8     cmp     rax, 1024
9     jne     .LBB0_1
10    ret
11
1 bar:                                # @bar
2     vmovsd  xmm0, qword ptr [rdi - 8]      # xmm0 = mem[0],zero
3     xor     eax, eax
4 .LBB0_1:                           # =>This Inner Loop Header: Depth=1
5     vmovsd  xmm1, qword ptr [rdi + 4*rax]  # xmm1 = mem[0],zero
6     vaddps  xmm0, xmm1, xmm0
7     vmovlps qword ptr [rdi + 4*rax], xmm0
8     add     rax, 2
9     cmp     rax, 1024
10    jne     .LBB0_1
11    ret
1 bar:                                # @bar
2     xor     eax, eax
3 .LBB0_1:                           # =>This Inner Loop Header: Depth=1
4     vmovsd  xmm0, qword ptr [rdi + 4*rax - 40] # xmm0 = mem[0],zero
5     vmovsd  xmm1, qword ptr [rdi + 4*rax]      # xmm1 = mem[0],zero
6     vaddps  xmm0, xmm1, xmm0
7     vmovlps qword ptr [rdi + 4*rax], xmm0
8     add     rax, 2
9     cmp     rax, 1024
10    jne     .LBB0_1
11    ret
```

4. Reductions

```
int foo(int *A) {  
    unsigned sum = 0;  
    for (int i = 0; i < 1024; ++i)  
        sum += A[i] + 5;  
    return sum;  
}
```

<https://www.godbolt.org/z/q7s xv8cEa>

4. Reductions

```
1 .LCPI0_0:  
2     .long 5                                # 0x5  
3 foo:                                     # @foo  
4     vpxor    xmm0, xmm0, xmm0  
5     xor     eax, eax  
6     vpbroadcastd zmm1, dword ptr [rip + .LCPI0_0] # zmm1 = [5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5]  
7 .LBB0_1:                                    # =>This Inner Loop Header: Depth=1  
8     vpaddd  zmm0, zmm0, zmmword ptr [rdi + 4*rax]  
9     vpaddd  zmm0, zmm0, zmm1  
10    add    rax, 16  
11    cmp    rax, 1024  
12    jne    .LBB0_1  
13    vextracti64x4 ymm1, zmm0, 1  
14    vpaddd  zmm0, zmm0, zmm1  
15    vextracti128 xmm1, ymm0, 1  
16    vpaddd  xmm0, xmm0, xmm1  
17    vpshufd xmm1, xmm0, 238                 # xmm1 = xmm0[2,3,2,3]  
18    vpaddd  xmm0, xmm0, xmm1  
19    vpshufd xmm1, xmm0, 85                  # xmm1 = xmm0[1,1,1,1]  
20    vpaddd  xmm0, xmm0, xmm1  
21    vmovd   eax, xmm0  
22    vzeroupper  
23    ret
```

5. Control Flow

```
int foo(int *A) {
    unsigned sum = 0;
    for (int i = 0; i < 1024; ++i)
        if (A[i] > 0)
            sum += A[i] + 5;
    return sum;
}
```

<https://www.godbolt.org/z/5v66a8Mq6>

5. Control Flow

```
1 .LCPI0_0:  
2     .long 5                                # 0x5  
3 foo:                                         # @foo  
4     vpxor  xmm0, xmm0, xmm0  
5     xor    eax, eax  
6     vpxor  xmm1, xmm1, xmm1  
7 .LBB0_1:                                     # =>This Inner Loop Header: Depth=1  
8     vmovdqu64      zmm2, zmmword ptr [rdi + 4*rax]  
9     vpcmpgtd      k1, zmm2, zmm0  
10    vpaddd   zmm2, zmm2, zmm1  
11    vpaddd   zmm1 {k1}, zmm2, dword ptr [rip + .LCPI0_0]{1to16}  
12    add    rax, 16  
13    cmp    rax, 1024  
14    jne    .LBB0_1  
15    vextracti64x4  ymm0, zmm1, 1  
16    vpaddd   zmm0, zmm1, zmm0  
17    vextracti128   xmm1, ymm0, 1  
18    vpaddd   xmm0, xmm0, xmm1  
19    vpshufd  xmm1, xmm0, 238                 # xmm1 = xmm0[2,3,2,3]  
20    vpaddd   xmm0, xmm0, xmm1  
21    vpshufd  xmm1, xmm0, 85                  # xmm1 = xmm0[1,1,1,1]  
22    vpaddd   xmm0, xmm0, xmm1  
23    vmovd   eax, xmm0  
24    vzeroupper  
25    ret
```

6. Non-contiguous data

```
void bar(float * restrict A, float * restrict B, float K) {
    for (int i = 0; i < 1024; ++i) {
        B[2*i]    = A[i] + K;
        B[2*i+1] = A[i] * K;
    }
}
```

<https://www.godbolt.org/z/bGc1GvThM>

```
void bar(float * restrict A, float * restrict B, int * restrict X) {
    for (int i = 0; i < 1024; ++i)
        B[i] *= A[X[i]];
}
```

<https://www.godbolt.org/z/31bzd4PxY>

6. Non-contiguous data

```
1 .LCPI0_0:  
2     .long 0          # 0x0  
3     .long 16         # 0x10  
4     .long 1          # 0x1  
5     .long 17         # 0x11  
6     .long 2          # 0x2  
7     .long 18         # 0x12  
8     .long 3          # 0x3  
9     .long 19         # 0x13  
10    .long 4          # 0x4  
11    .long 20         # 0x14  
12    .long 5          # 0x5  
13    .long 21         # 0x15  
14    .long 6          # 0x6  
15    .long 22         # 0x16  
16    .long 7          # 0x7  
17    .long 23         # 0x17  
18 .LCPI0_1:  
19     .long 8          # 0x8  
20     .long 24         # 0x18  
21     .long 9          # 0x9  
22     .long 25         # 0x19  
23     .long 10         # 0xa  
24     .long 26         # 0x1a  
25     .long 11         # 0xb  
26     .long 27         # 0x1b  
27     .long 12         # 0xc  
28     .long 28         # 0x1c  
29     .long 13         # 0xd  
30     .long 29         # 0x1d  
31     .long 14         # 0xe  
32     .long 30         # 0x1e  
33     .long 15         # 0xf  
34     .long 31         # 0x1f  
35 bar:                      # @bar  
36     vbroadcastss zmm0, xmm0  
37     xor    eax, eax  
38     vmovaps zmm1, zmmword ptr [rip + .LCPI0_0] # zmm1 = [0,16,1,17,2,18,3,19,4,20,5,21,6,22,7,23]  
39     vmovaps zmm2, zmmword ptr [rip + .LCPI0_1] # zmm2 = [8,24,9,25,10,26,11,27,12,28,13,29,14,30,15,31]  
40 .LBB0_1:                   # =>This Inner Loop Header: Depth=1  
41     vmovups zmm3, zmmword ptr [rdi + 4*rax]  
42     vaddps zmm4, zmm3, zmm0  
43     vmulps zmm3, zmm3, zmm0  
44     vmovaps zmm5, zmm4  
45     vperm2ps   zmm5, zmm1, zmm3  
46     vperm2ps   zmm4, zmm2, zmm3  
47     vmovups zmmword ptr [rsi + 8*rax + 64], zmm4  
48     vmovups zmmword ptr [rsi + 8*rax], zmm5  
49     add    rax, 16  
50     cmp    rax, 1024  
51     jne    .LBB0_1  
52     vzeroupper  
53     ret
```

7. Cost Model Failures

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

- Clang v13 vs Clang v14
 - v13 is capable of vectorizing but thinks not profitable
 - v14's updated cost model now thinks it's profitable to vectorize
 - You can get about 30% speedup

<https://godbolt.org/z/b3cYo6388>

7. Cost Model Failures

```

1 matmul:                                # @matmul
2     push r15
3     push r14
4     push rbx
5     test edi, edi
6     jle .LBB0_7
7     mov eax, edi
8     lea r15, [4*rax]
9     xor r8d, r8d
10    .LBB0_2:                           # =>This Loop Header: Depth=1
11    mov r9, r8
12    imul r9, rax
13    mov r10, rdx
14    xor r11d, r11d
15    .LBB0_3:                           # Parent Loop BB0_2 Depth=1
16    lea r14, [r9 + r11]
17    vmovss xmm0, dword ptr [rcx + 4*r14]  # xmm0 = mem[0],zero,zero,zero
18    mov rbx, r10
19    xor edi, edi
20    .LBB0_4:                           # Parent Loop BB0_2 Depth=1
21    vmovss xmm1, dword ptr [rbx]          # xmm1 = mem[0],zero,zero,zero
22    vmulss xmm1, xmm1, dword ptr [rsi + 4*rdi]
23    vaddss xmm0, xmm0, xmm1
24    add rdi, 1
25    add rbx, r15
26    cmp rax, rdi
27    jne .LBB0_4
28    vmovss dword ptr [rcx + 4*r14], xmm0
29    add r11, 1
30    add r10, 4
31    cmp r11, rax
32    jne .LBB0_3
33    add r8, 1
34    add rsi, r15
35    cmp r8, rax
36    jne .LBB0_2
37    .LBB0_7:
38    pop rbx
39    pop r14
40    pop r15
41    ret

2     push rbp
3     push r15
4     push r14
5     push r13
6     push r12
7     push rbx
8     mov qword ptr [rsp - 24], rcx      # 8-byte Spill
9     mov qword ptr [rsp - 32], rdx      # 8-byte Spill
10    test edi, edi
11    jle .LBB0_2
12    mov r8d, r8d
13    and r8d, -8
14    les r13, [4*r9]
15    mov r10, r9
16    shl r10, 5
17    xor eax, eax
18    vxorps xmm0, xmm0, xmm0
19    jmp .LBB0_2
20    .LBB0_11:                          # in Loop: Header=BB0_2 Depth=1
21    mov rax, qword ptr [rsp - 36]       # 8-byte Reload
22    add rax, 1
23    add rsi, r13
24    cmp rax, r9
25    je .LBB0_12
26    .LBB0_2:                           # =>This Loop Header: Depth=1
27    mov qword ptr [rsp - 16], rax      # 8-byte Spill
28    imul rax, r9
29    mov rcx, qword ptr [rsp - 24]       # 8-byte Reload
30    lea rdx, [rcx + 4*rax]
31    mov r15, qword ptr [rsp - 32]       # 8-byte Reload
32    xor r12d, r12d
33    mov qword ptr [rsp - 8], rdx       # 8-byte Spill
34    jmp .LBB0_3
35    .LBB0_10:                          # in Loop: Header=BB0_3 Depth=2
36    vnmovss dword ptr [rdx + 4*r12], xmm1
37    add r12, 1
38    add r15, 4
39    cmp r12, r9
40    je .LBB0_11
41    .LBB0_3:                           # Parent Loop BB0_2 Depth=1
42    vnmovss word, dword ptr [rdx + 4*r12] # xmm1 = mem[0],zero,zero,zero
43    cmp edi, $8
44    jae .LBB0_5
45    xor eax, eax
46    jmp .LBB0_3
47    .LBB0_5:                           # in Loop: Header=BB0_3 Depth=2
48    vblendps xmm1, xmm0, xmm1, 1        # xmm1 = xmm1[0],xmm0[1,2,3]
49    mov rbx, r15
50    xor eax, eax
51    .LBB0_6:                           # Parent Loop BB0_2 Depth=1
52    lea rbp, [rbx + r13]
53    mov r15, rbp
54    add r11, r13
55    lea rcx, [r11 + r13]
56    lea rdx, [rcx + r13]
57    lea rsi, [rdx + r13]
58    lea r14, [rsi + r13]
59    vnmovss xmm2, dword ptr [r13 + rcx] # xmm2 = mem[0],zero,zero,zero
60    vinsertps xmm2, xmm2, dword ptr [r13 + r14], 16 # xmm2 = xmm2[0],mem[0],xmm2[2,3]
61    vinsertps xmm2, xmm2, dword ptr [r13 + r14], 32 # xmm2 = xmm2[0,1],mem[0],xmm2[1,2]
62    ror r14, r13
63    vinsertps xmm2, xmm2, dword ptr [r13 + r14], 48 # xmm2 = xmm2[1,2],mem[0]
64    vnmovss xmm3, dword ptr [rbx]         # xmm3 = mem[0],zero,zero,zero
65    vinsertps xmm3, xmm3, dword ptr [rbx + r13], 16 # xmm3 = xmm3[0],mem[0],xmm3[2,3]
66    vinsertps xmm3, xmm3, dword ptr [rbx + r13], 32 # xmm3 = xmm3[0,1],mem[0],xmm3[1,2]
67    vinsertps xmm3, xmm3, dword ptr [r13 + r14], 48 # xmm3 = xmm3[1,2],mem[0]
68    vinsertf28 ymm2, ymm2, ymm2, 1
69    vmlps ymm2, ymm1, ymm2
70    vaddps ymm1, ymm1, ymm2
71    add rax, 8
72    add rbx, r10
73    cmp r10, rax
74    je .LBB0_6
75    vextractf128 ymm2, ymm1, 1
76    vaddps ymm1, ymm1, ymm2
77    vpermilpd ymm2, ymm1, 1           # ymm2 = ymm1[1,0]
78    vaddps ymm1, ymm1, ymm2
79    vnmovshup ymm2, ymm1             # ymm2 = ymm1[1,1,3,3]
80    vaddss ymm1, ymm1, ymm2
81    mov rax, r8
82    cmp r8, r9
83    mov r9, qword ptr [rsp - 8]       # 8-byte Reload
84    je .LBB0_10
85    .LBB0_8:                           # in Loop: Header=BB0_3 Depth=2
86    mov rcx, r9
87    imul rcx, rax
88    lea rdx, [r15 + 4*rcx]
89    vnmovss xmm2, dword ptr [rbx]       # xmm2 = mem[0],zero,zero,zero
90    vmlps xmm2, xmm2, dword ptr [rsi + 4*rax]
91    vaddss xmm2, xmm2, xmm1
92    add rax, 1
93    add rbx, r13
94    cmp r9, rax
95    jne .LBB0_9
96    jmp .LBB0_10
97    .LBB0_12:
98    pop rbx
99    pop r12
100   pop r13
101   pop r14
102   pop r15
103   pop rbp

```

8. Vectorization of Function Calls

```
#include <math.h>

void bar(float * restrict A, float * restrict B) {
    for (int i = 0; i < 1024; ++i) {
        B[i] = sinf(A[i]);
    }
}
```

- Works for a few built-in functions
- Note, need to add the exact library to the command line

<https://www.godbolt.org/z/ra8Esjx9>

8. Vectorization of Function Calls

```
1 bar:                                # @bar
2     push    r15
3     push    r14
4     push    rbx
5     mov     r14, rsi
6     mov     r15, rdi
7     xor     ebx, ebx
8 .LBB0_1:                           # =>This Inner Loop Header: Depth=1
9     vmovups ymm0, ymmword ptr [r15 + 4*rbx]
10    call    _ZGVdN8v_sinf@PLT
11    vmovups ymmword ptr [r14 + 4*rbx], ymm0
12    add    rbx, 8
13    cmp    rbx, 1024
14    jne    .LBB0_1
15    pop    rbx
16    pop    r14
17    pop    r15
18    vzeroupper
19    ret
```

9. Different Hardware Versions

```
void bar(float * restrict A, float * restrict B) {  
    for (int i = 0; i < 4; ++i)  
        A[i] += B[i];  
}
```

- Different Intel chips have different vector hardware
MMX, SSE, SSE2, SSE3, AVX, AVX2, AVX512, AVX512VNNI etc.

<https://www.godbolt.org/z/dd53r1Mf7>

9. Different Hardware Versions

The image shows four separate windows of the GDB assembly editor, each displaying assembly code for the same C function named `bar`. The code is identical across all four versions, consisting of an inner loop header at address `.LBB0_1` and a loop body from address 4 to 10. The assembly instructions include `xor eax, eax`, `movss xmm0, dword ptr [rdi + 4*rax]`, `addss xmm0, dword ptr [rsi + 4*rax]`, `movss dword ptr [rdi + 4*rax], xmm0`, `inc rax`, `cmp rax, 1024`, `jne .LBB0_1`, and `ret`.

The differences between the four versions are in the floating-point operations:

- Top Left (Compiler 1):** Uses SSE2 instructions (`movss`, `addss`, `movss`).
- Top Right (Compiler 2):** Uses SSE4.2 instructions (`movups`, `movups`, `addps`, `movups`).
- Bottom Left (Compiler 3):** Uses AVX2 instructions (`vmovups`, `vaddps`, `vmovups`).
- Bottom Right (Compiler 4):** Uses AVX512VNNI instructions (`vmovups`, `vaddps`, `vmovups`).

Each window includes a status bar at the bottom showing the compiler version, build time, and memory usage.

```
1 bar: # @bar
2 xor eax, eax
3 .LBB0_1: # =>This Inner Loop Header: Depth=1
4 movss xmm0, dword ptr [rdi + 4*rax] # xmm0 = mem[0],zero,zero,zero
5 addss xmm0, dword ptr [rsi + 4*rax]
6 movss dword ptr [rdi + 4*rax], xmm0
7 inc rax
8 cmp rax, 1024
9 jne .LBB0_1
10 ret
```

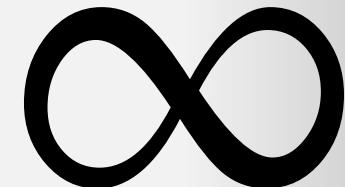
```
1 bar: # @bar
2 xor eax, eax
3 .LBB0_1: # =>This Inner Loop Header: Depth=1
4 movups xmm0, xmmword ptr [rsi + 4*rax]
5 movups xmm1, xmmword ptr [rdi + 4*rax]
6 addps xmm1, xmm0
7 movups xmmword ptr [rdi + 4*rax], xmm1
8 add rax, 4
9 cmp rax, 1024
10 jne .LBB0_1
11 ret
```

```
1 bar: # @bar
2 xor eax, eax
3 .LBB0_1: # =>This Inner Loop Header: Depth=1
4 vmovups ymm0, ymmword ptr [rdi + 4*rax]
5 vaddps ymm0, ymm0, ymmword ptr [rsi + 4*rax]
6 vmovups ymmword ptr [rdi + 4*rax], ymm0
7 add rax, 8
8 cmp rax, 1024
9 jne .LBB0_1
10 vzeroupper
11 ret
```

```
1 bar: # @bar
2 xor eax, eax
3 .LBB0_1: # =>This Inner Loop Header: Depth=1
4 vmovups zmm0, zmmword ptr [rdi + 4*rax]
5 vaddps zmm0, zmm0, zmmword ptr [rsi + 4*rax]
6 vmovups zmmword ptr [rdi + 4*rax], zmm0
7 add rax, 16
8 cmp rax, 1024
9 jne .LBB0_1
10 vzeroupper
11 ret
```



SPEED
LIMIT



PER ORDER OF 6.106

BLOOD SWEAT AND TEARS: HAND VECTORIZATION USING INTRINSICS

Example: SAXPY

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

```
void saxpy(int n, float a, float *x, float *y) {
    int vec_len = 8;
    for (int i = 0; i < n % vec_len; i++)
        y[i] += a * x[i];

    __m256 va = _mm256_set1_ps(a);
    for (int i = n % num_elems; i < n; i += vec_len) {
        __m256 vx    = _mm256_loadu_ps(x + i);
        __m256 vy    = _mm256_loadu_ps(y + i);
        __m256 ax    = _mm256_mul_ps(va, vx);
        __m256 axpy = _mm256_add_ps(ax, vy);
        _mm256_storeu_ps(y + i, axpy);
    }
}
```

Example: SAXPY

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

Compiler can do this automatically too.

Running time ($n=1024$)

- Vectorized (256-bit vector) = 494 cycles
- Scalar = 3309 cycles
- ... about 6.7x faster.

Vectorizing Reductions

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

```
float sum(int n, float *x) {
    int vec_len = 8;
    float s = 0.0;
    for (int i = 0; i < n % vec_len; i++)
        s += x[i];

    __m256 v = _mm256_set_ps(0., 0., 0., 0., 0., 0., 0., s);
    for (int i = n % vec_len; i < n; i += vec_len) {
        __m256 vx = _mm256_loadu_ps(x + i);
        v = _mm256_add_ps(v, vx);
    }
    return reduce_vector(v);
}
```

Vectorizing Reductions (cont.)

```
float reduce_vector(__m256 v) {
    __m128 v_0_4 = _mm256_extracti128_si256(v, 0);
    __m128 v_4_8 = _mm256_extracti128_si256(v, 1);
    // t = {v3+v7, v2+v6, v1+v5, v0+v4}
    __m128 t = _mm_add_ps(v_0_4, v_4_8);
    // t1 = {v2+v6, v3+v7, v0+v4, v1+v5}
    __m128 t1 = _mm_shuffle_ps(t, t, _MM_SHUFFLE(2, 3, 0, 1));
    // t2 = {_, v2+v6+v3+v7, _, v0+v4+v1+v5}
    __m128 t2 = _mm_add_ps(t, t1);
    // t3 = {_, _, _, v2+v6+v3+v7}
    __m128 t3 = _mm_shuffle_ps(t2, t2, _MM_SHUFFLE(0, 0, 0, 2));
    // t4 = {_, _, _, v0+v4+v1+v5+v2+v6+v3+v7}
    __m128 t4 = _mm_add_ps(t2, t3);
    // Extract the first lane
    return _mm_cvtsi32_f32(t4);
}
```

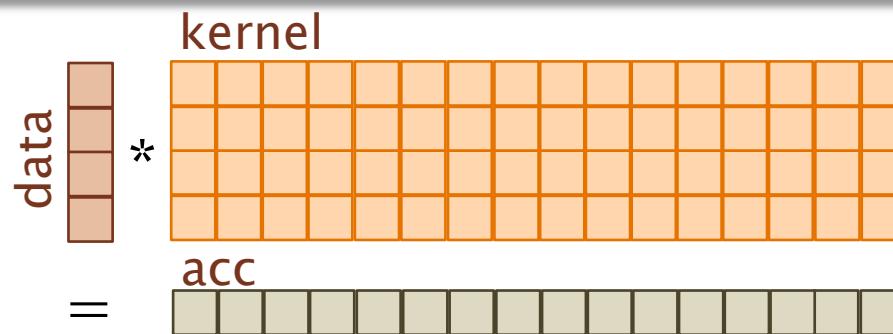
Vectorizing Control Flow

```
void foo(float *restrict x, float *restrict y) {
    for (int i = 0; i < 8; i++) {
        if (x[i] < y[i])
            x[i] += y[i];
        else
            x[i] -= y[i];
    }
}
```

```
__m256 x_vec      = _mm256_loadu_ps(x);
__m256 y_vec      = _mm256_loadu_ps(y);
__m256 x_lt_y     = _mm256_cmp_ps(x_vec, y_vec, _CMP_LT_OS);
__m256 if_true    = _mm256_add_ps(x_vec, y_vec);
__m256 if_false   = _mm256_sub_ps(x_vec, y_vec);
__m256 result     = _mm256_blendv_ps(if_true, if_false, x_lt_y);
_mm256_storeu_ps(x, result);
```

Vectorizing the Dot Product

```
void dot_16x1x16(uint8_t *data, int8_t kernel[][4], int32_t *acc) {  
    for (int i = 0; i < 16; i++)  
        for (int j = 0; j < 4; j++)  
            acc[i] += data[j] * kernel[i][j];  
}
```



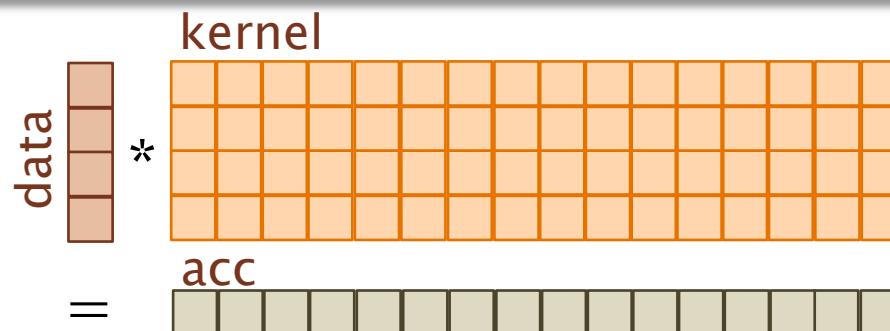
2.2x faster than scalar.

- Compiler vectorizes the inner loop
 - Similar to a reduction

<https://www.godbolt.org/z/8ecxrThWK>

Vectorizing the Dot Product

```
void dot_16x1x16(uint8_t *data, int8_t kernel[][4], int32_t *acc) {  
    for (int i = 0; i < 16; i++)  
        for (int j = 0; j < 4; j++)  
            acc[i] += data[j] * kernel[i][j];  
}
```

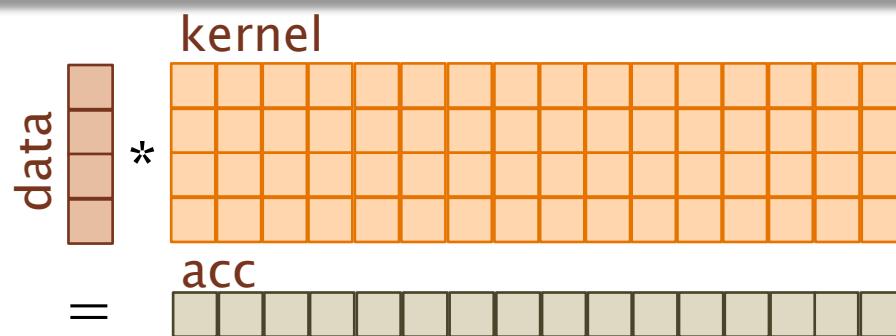


2.2x faster than scalar.

```
1  dot:                                # @dot  
2      vpmovzxbd    xmm0, dword ptr [rdi]    # xmm0 = mem[0],zero,zero,zero,mem[1],zero,zero,zero,mem[2],zero,zero,zero,mem[3],zero,zero,zero  
3      xor    eax, eax  
4 .LBB0_1:                           # =>This Inner Loop Header: Depth=1  
5      vmovd  xmm1, dword ptr [rdx + 4*rax]  # xmm1 = mem[0],zero,zero,zero  
6      vpmovsxbd    xmm2, dword ptr [rsi + 4*rax]  
7      vpmaddwd  xmm2, xmm2, xmm0  
8      vpaddd  xmm1, xmm2, xmm1  
9      vpshufd xmm2, xmm1, 238                # xmm2 = xmm1[2,3,2,3]  
10     vpaddd  xmm1, xmm1, xmm2  
11     vpshufd xmm2, xmm1, 85                # xmm2 = xmm1[1,1,1,1]  
12     vpaddd  xmm1, xmm1, xmm2  
13     vmovd  dword ptr [rdx + 4*rax], xmm1  
14     inc    rax  
15     cmp    rax, 16  
16     jne    .LBB0_1  
17     ret
```

Vectorizing the Dot Product

```
void dot_16x1x16(uint8_t *data, int8_t kernel[][4], int32_t *acc) {  
    for (int i = 0; i < 16; i++)  
        for (int j = 0; j < 4; j++)  
            acc[i] += data[j] * kernel[i][j];  
}
```



2.2x faster than scalar.

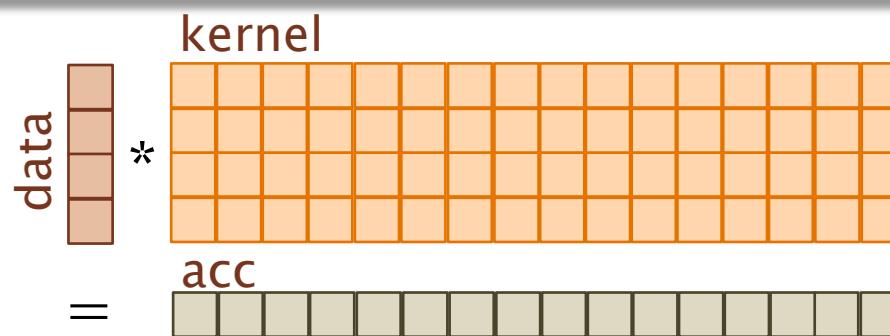
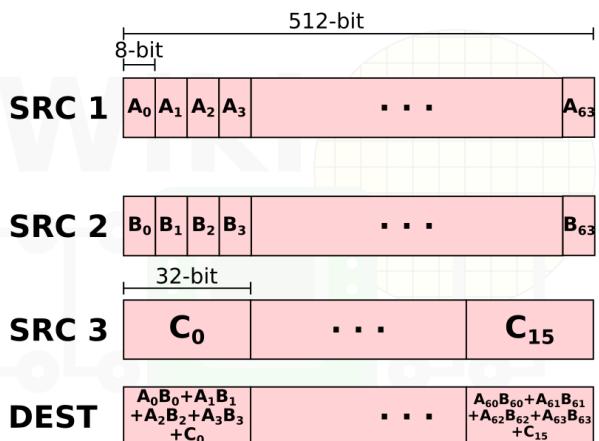
- Compiler vectorizes the inner loop
 - Similar to a reduction
- Can we do better?

<https://www.godbolt.org/z/8ecxrThWK>

Vectorizing the Dot Product

```
void dot_16x1x16(uint8_t *data, int8_t kernel[][4], int32_t *acc) {  
    for (int i = 0; i < 16; i++)  
        for (int j = 0; j < 4; j++)  
            acc[i] += data[j] * kernel[i][j];  
}
```

VPDPBUSD



2.2x faster than scalar.

11x faster than scalar.

```
_m512i acc_vec = _mm512_loadu_epi32(acc);  
// {data0, data1, data2, data3, ..., data0, data1, data2, data3}  
_m512i data_vec = _mm512_set1_epi32(*(int32_t*)data);  
_m512i kernel_vec = _mm512_loadu_epi32(&kernel);  
_mm512_dpbusd_epu32(acc_vec, data_vec, kernel_vec);
```

Conclusion

- Vector hardware is great for performance
 - Architecture is changing/improving quickly
- Compilers can vectorize your code
 - However, compilers are finicky
 - May have to coax the compiler
- Can also use intrinsics to hand vectorize
 - ‘Assembly programming’
 - Hard to keep-up with hardware updates