

Software Performance Engineering

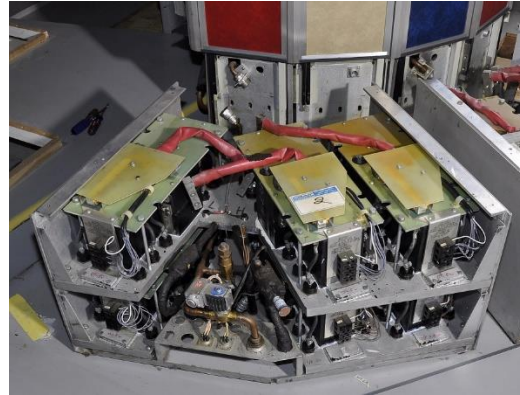
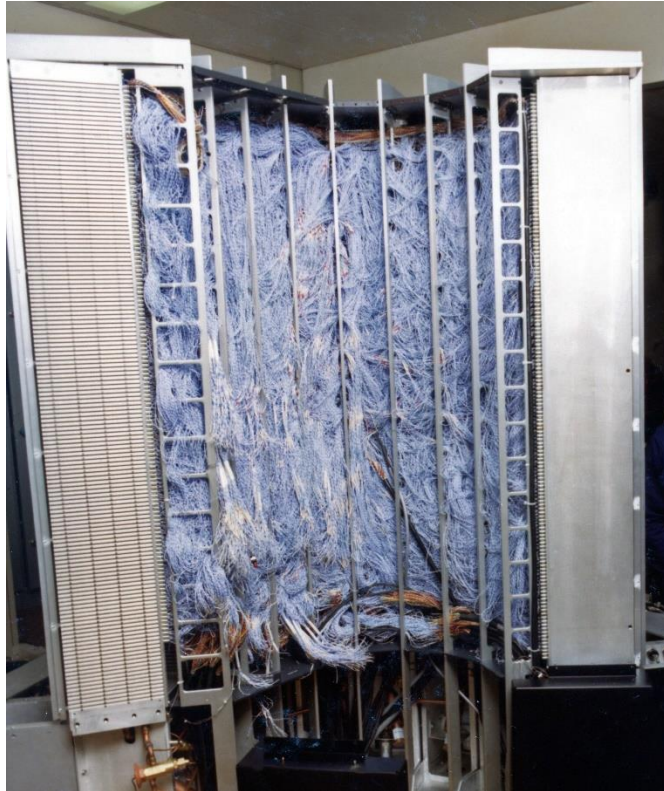
LECTURE 5 Vectorization

Xuhao Chen
September 18, 2025



Vectorization

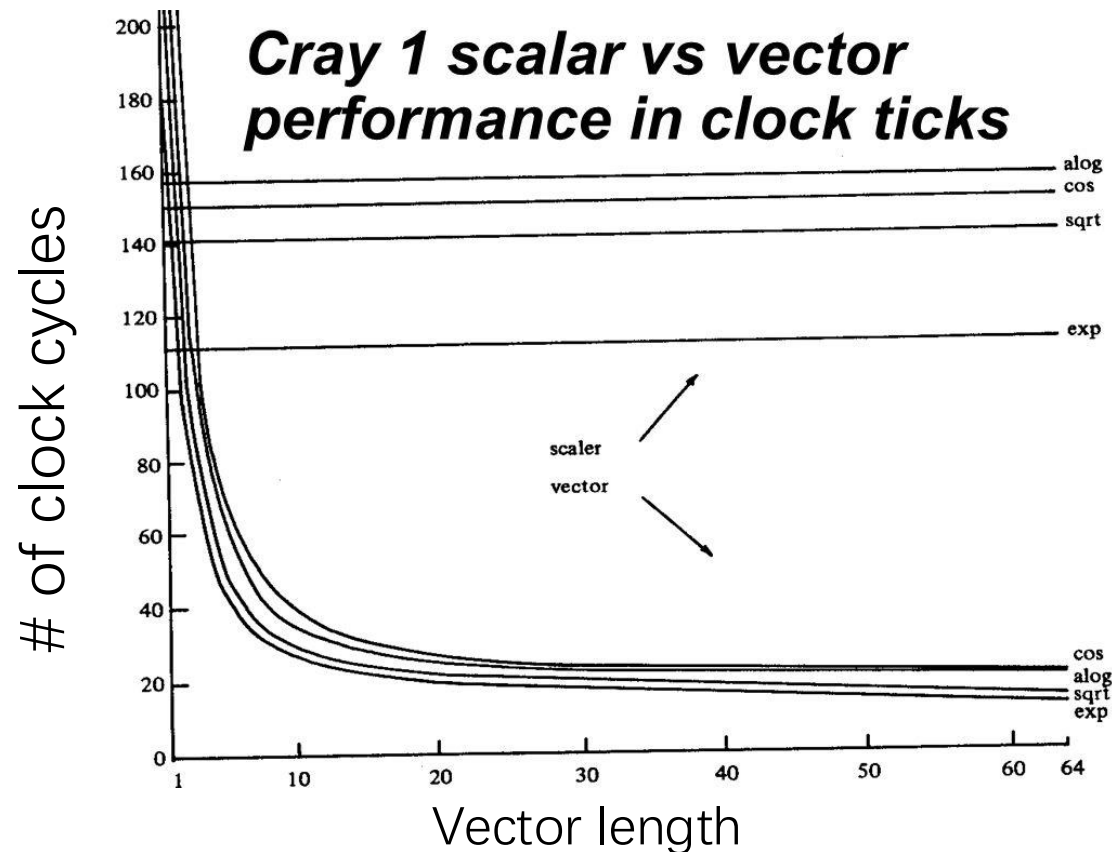
- The First Supercomputers were developed in the 70's
- 200,000 gates, 80 MHz, \$10M



Seymour Cray

Vectorization

- The First Supercomputers were developed in the 70's
- Required long vectors for performance



Seymour Cray

Multimedia Instructions

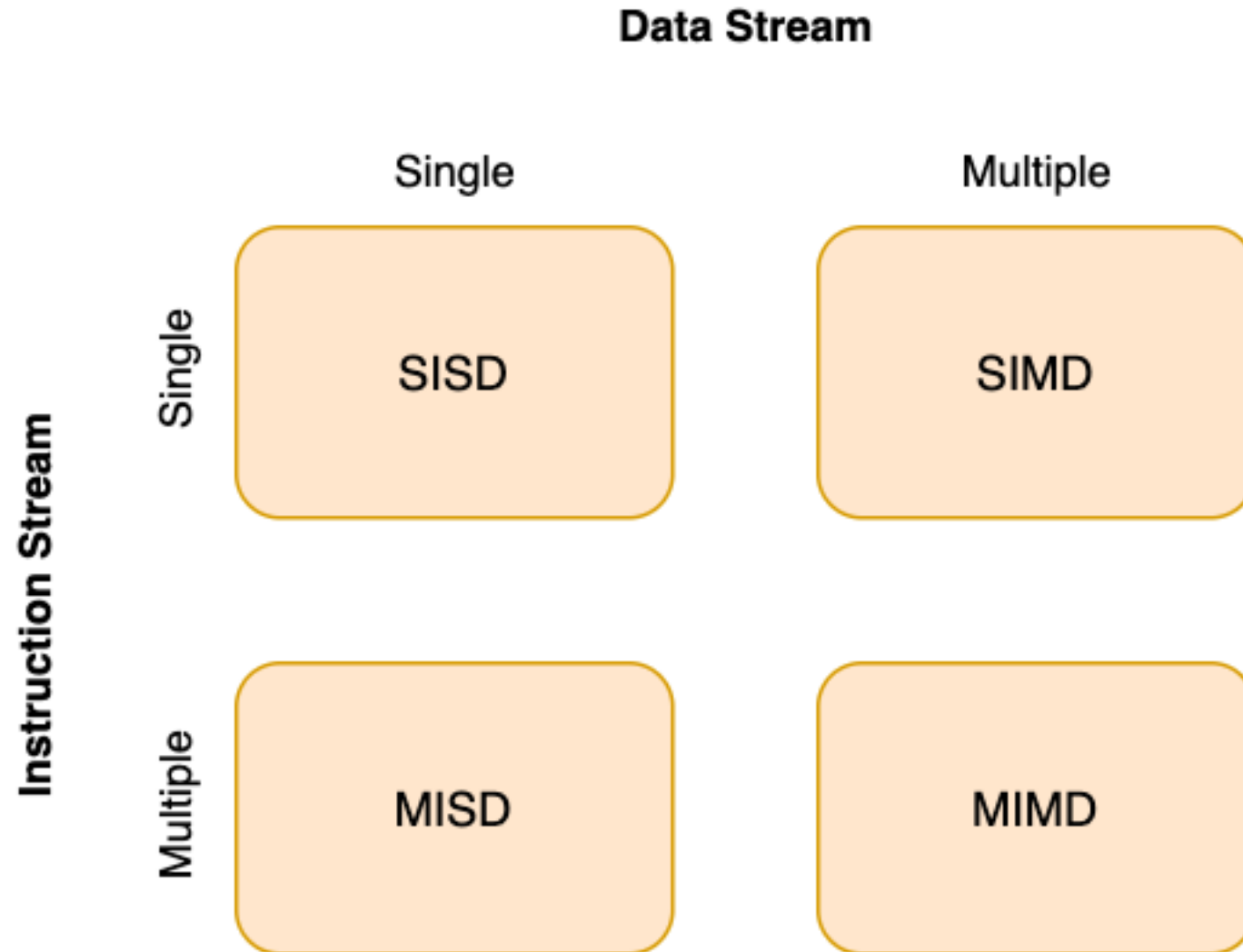
- Multimedia ISA extensions came in the 90's
 - ▣ SIMD: Single Instruction Multiple Data
 - ▣ Integrated into the processor
 - ▣ Short and fast
 - ▣ No startup overhead

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

WHAT IS SIMD?



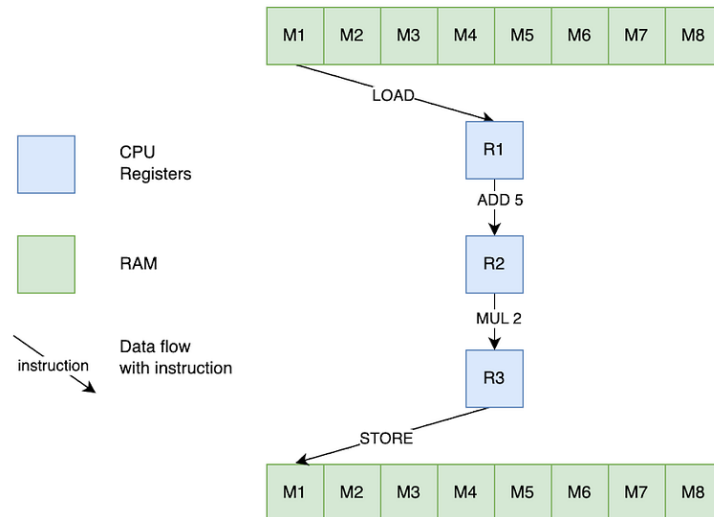
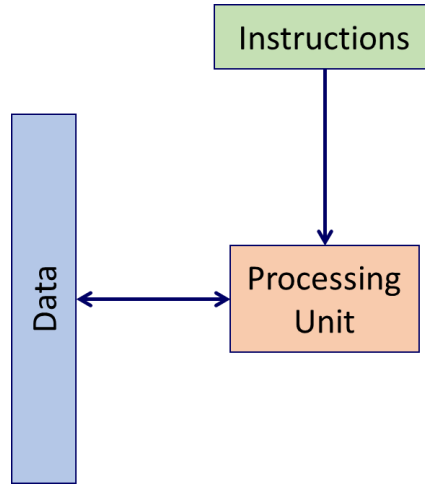
Flynn Taxonomy (1966)



Flynn Taxonomy (1966)

Single-Instruction Single-Data

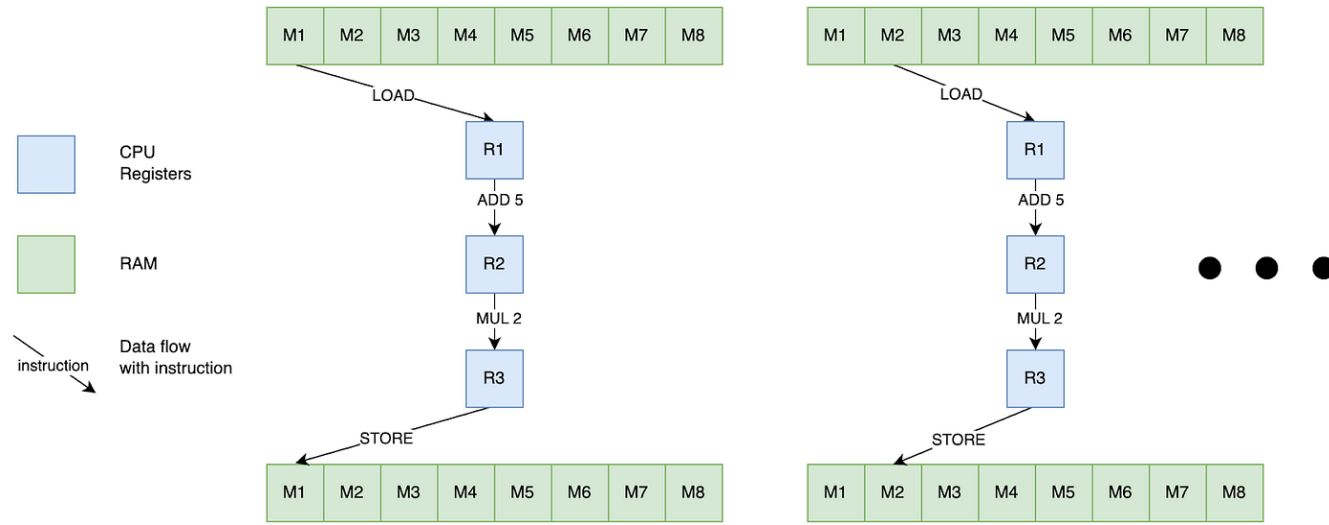
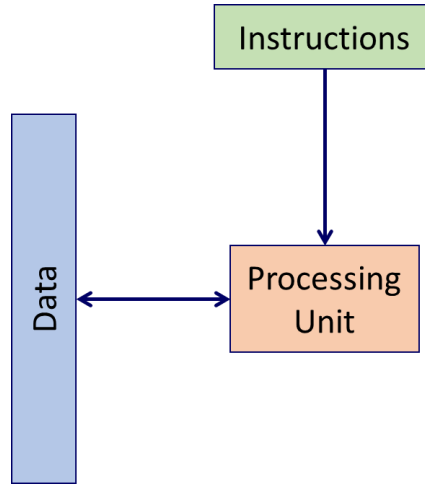
(e.g. Single-Core Processors)



Flynn Taxonomy (1966)

Single-Instruction Single-Data

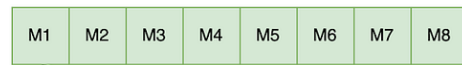
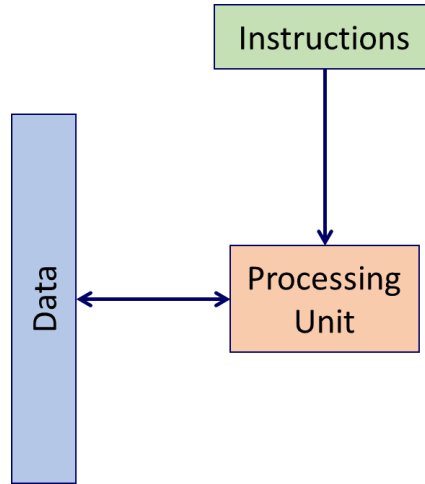
(e.g. Single-Core Processors)



Flynn Taxonomy (1966)

Single-Instruction Single-Data

(e.g. Single-Core Processors)



LOAD



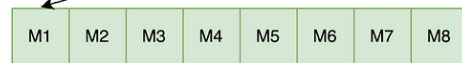
ADD 5



MUL 2



STORE



LOAD



ADD 5



MUL 2

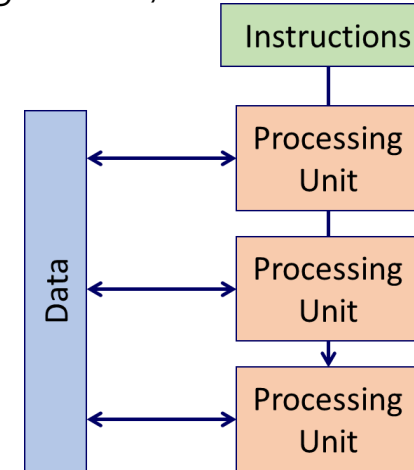


STORE



Single-Instruction Multi-Data

(e.g. GPUs, Intel SIMD Extensions)



VLOAD



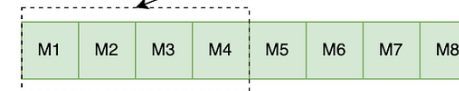
VADD 5



VMUL 2



VSTORE



VLOAD



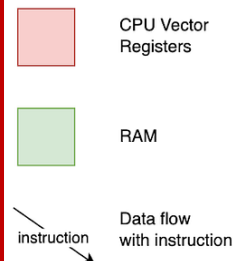
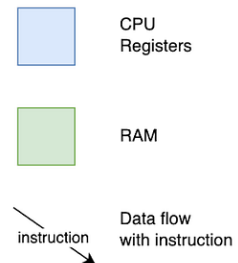
VADD 5



VMUL 2



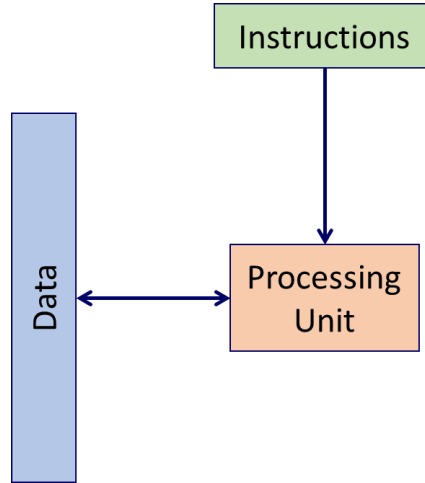
VSTORE



Flynn Taxonomy (1966)

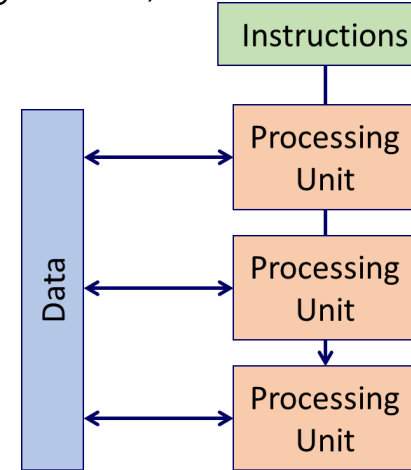
Single-Instruction Single-Data

(e.g. Single-Core Processors)



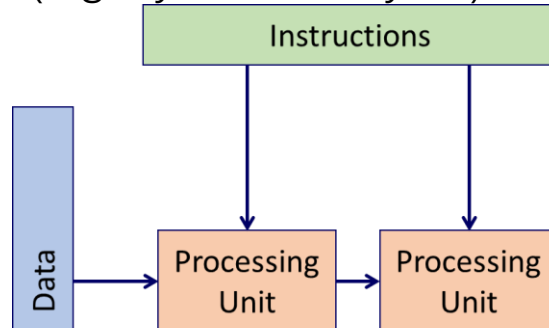
Single-Instruction Multi-Data

(e.g. GPUs, Intel SIMD Extensions)



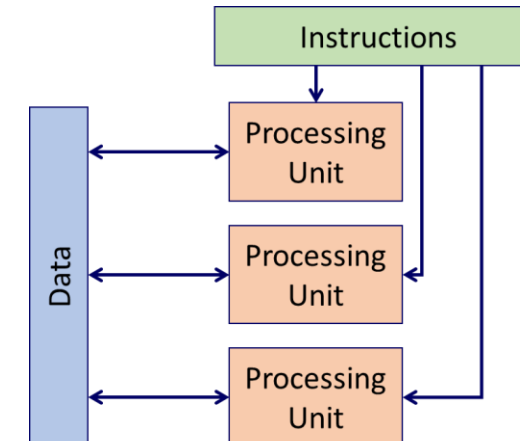
Multi-Instruction Single-Data

(e.g. Systolic Arrays, ...)



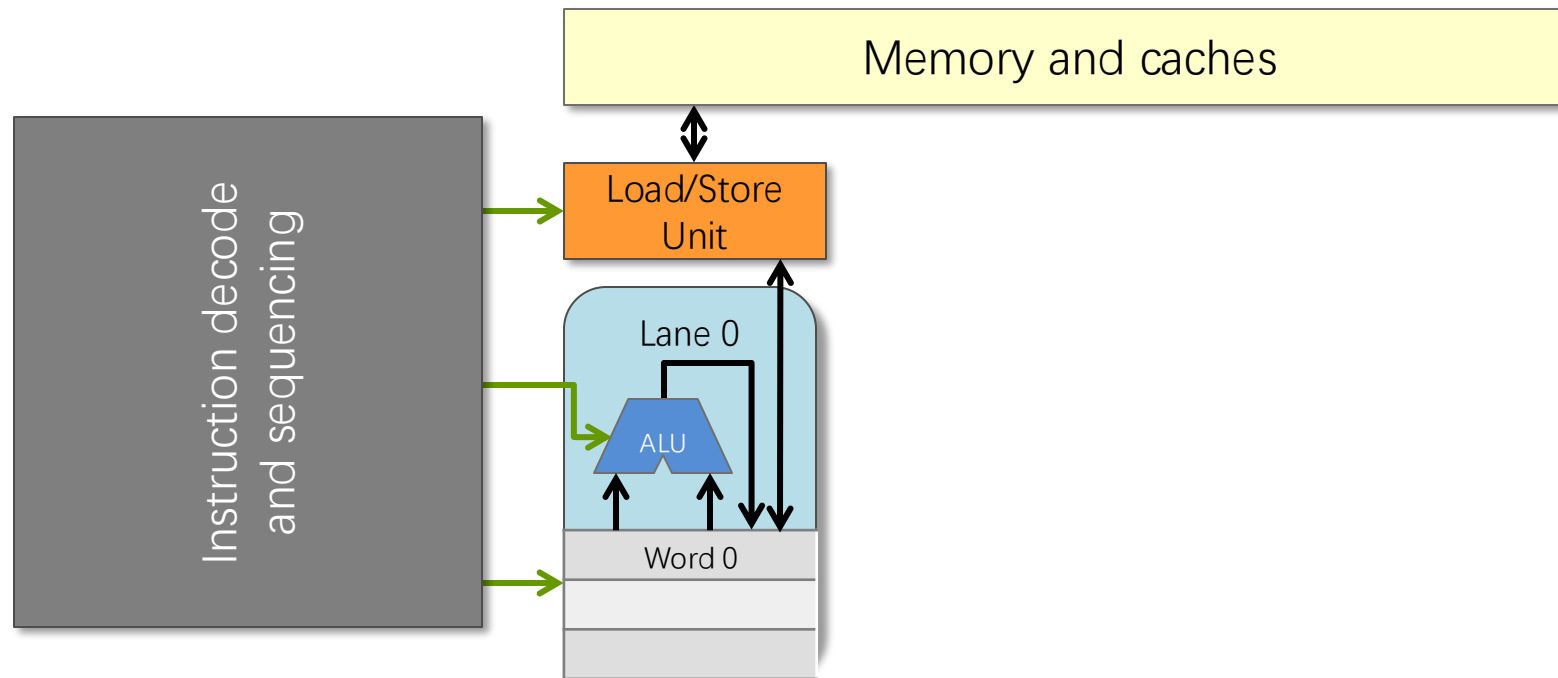
Multi-Instruction Multi-Data

(e.g. Parallel Computers)



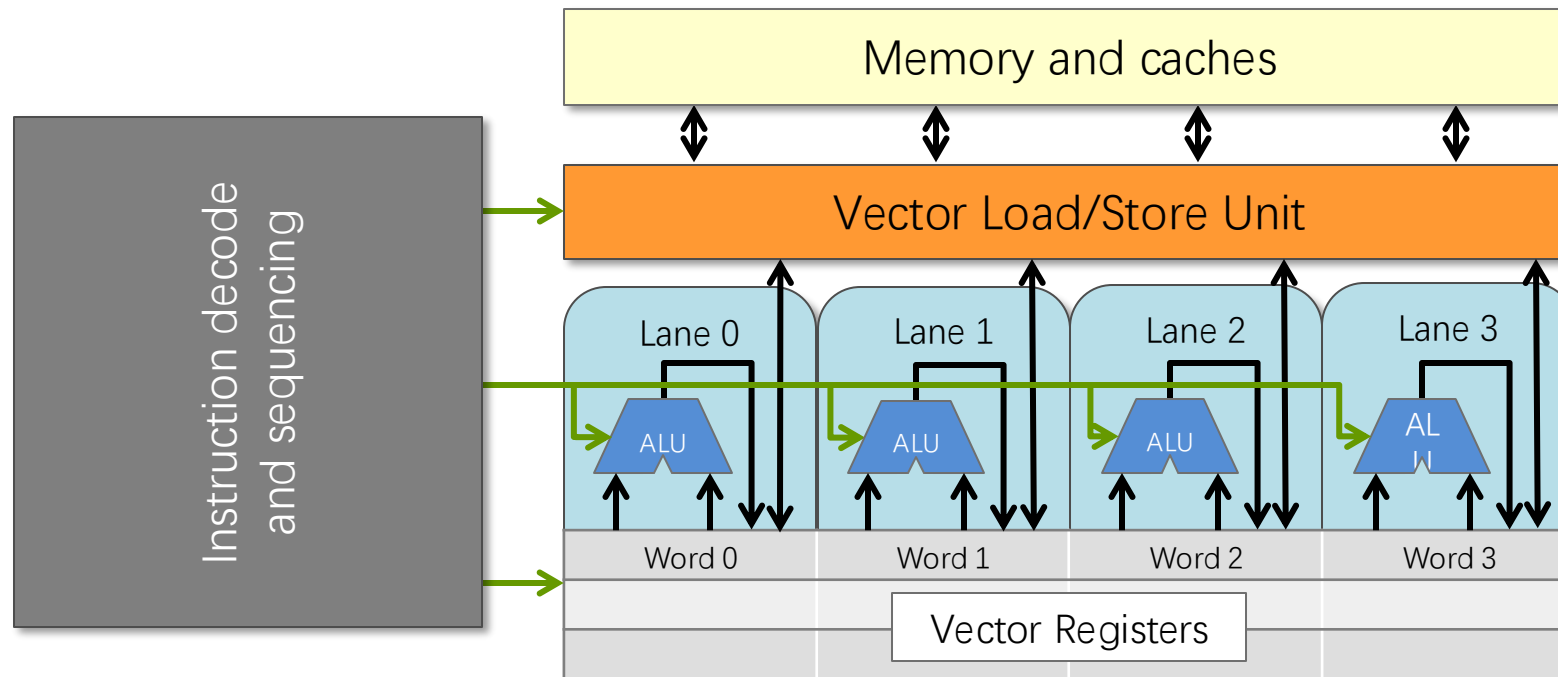
Scalar Hardware

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

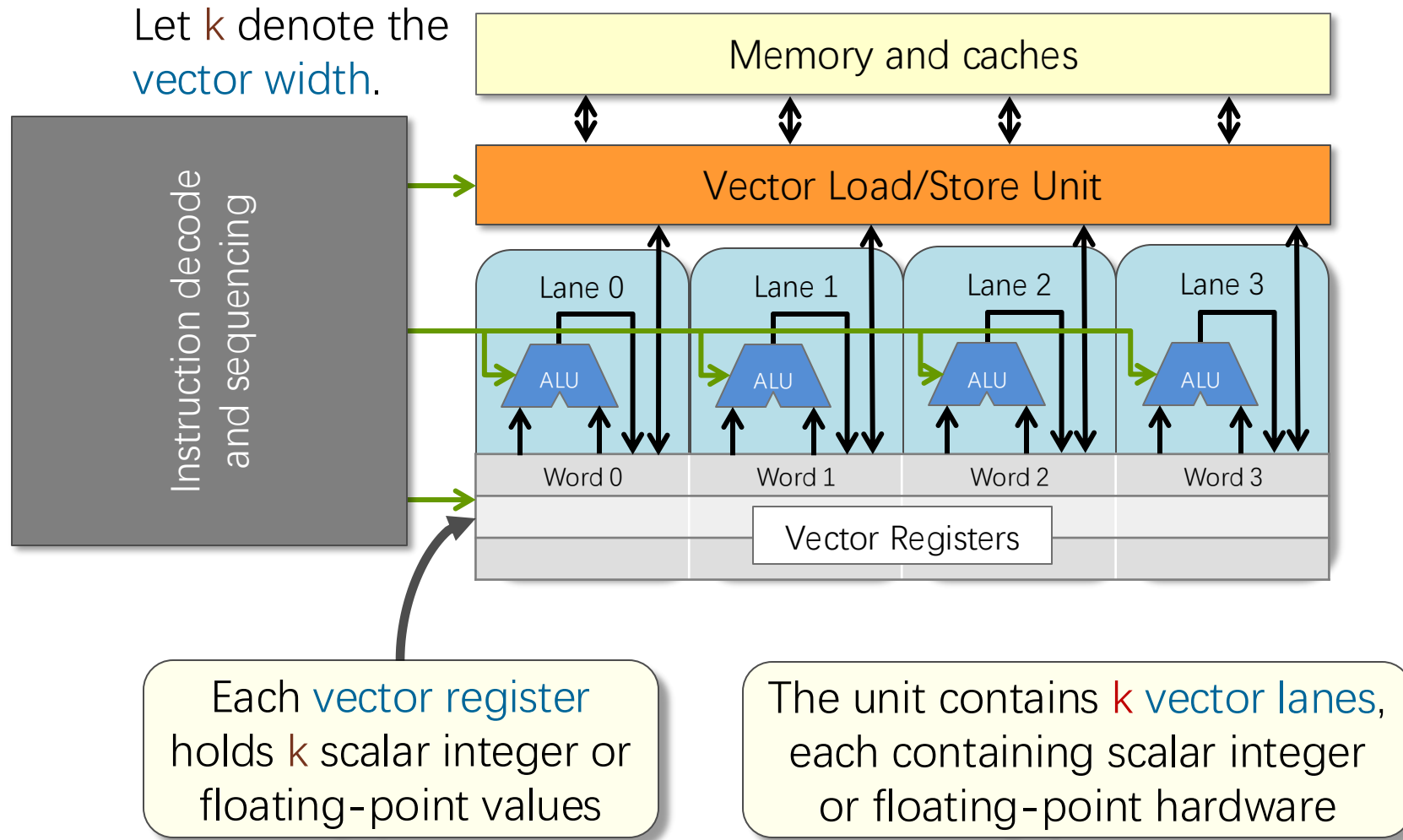


Vector Hardware

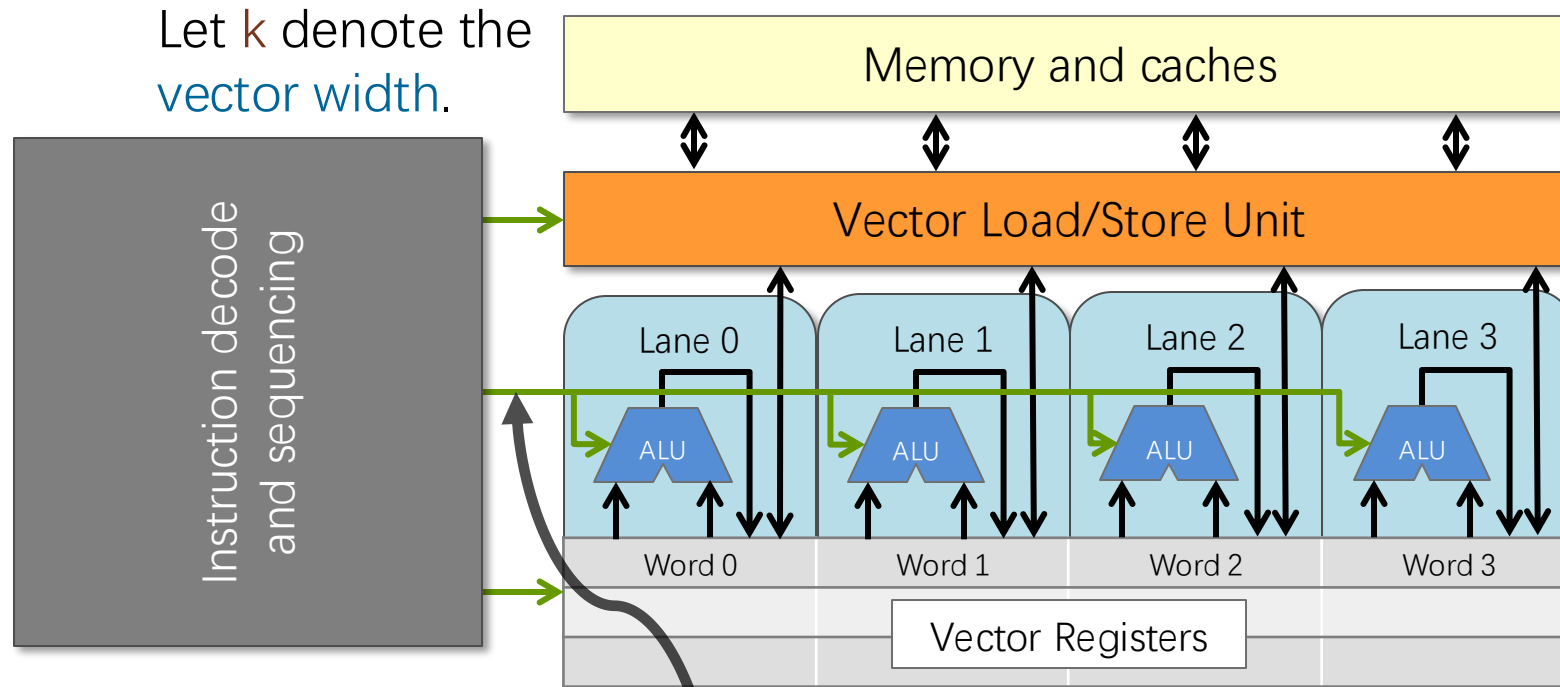
Modern microprocessors incorporate **vector units** to process data in a **single-instruction stream, multiple-data stream (SIMD)** fashion



A k -Wide Vector Unit



A k-Wide Vector Unit



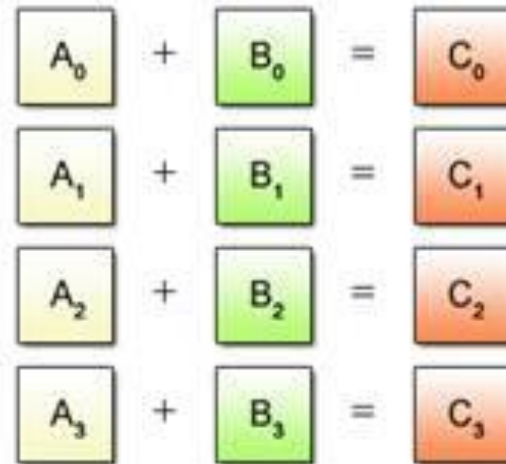
All vector lanes operate in **lock-step** and use the **same** instruction and control signals

Lock step!

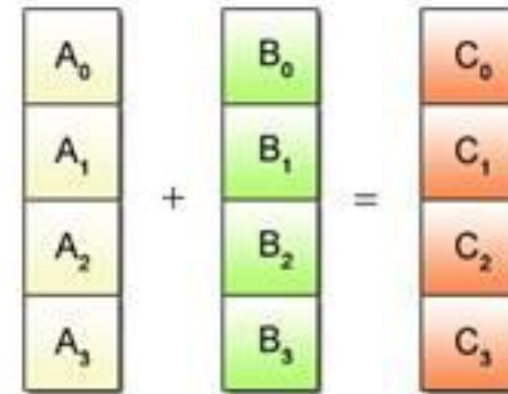
SIMD Instructions

- 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

(a) Scalar Operation

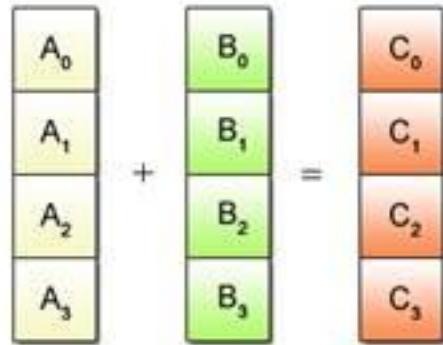


(b) SIMD Operation

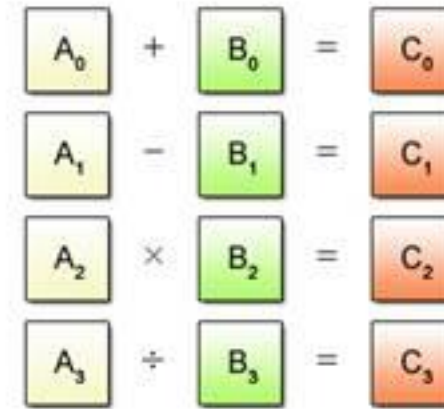


SIMD Instructions

- SIMD instructions typically operate in an **elementwise** fashion
- All lanes perform **exactly the same operation** on each element



Example of SIMD
Processable Patterns

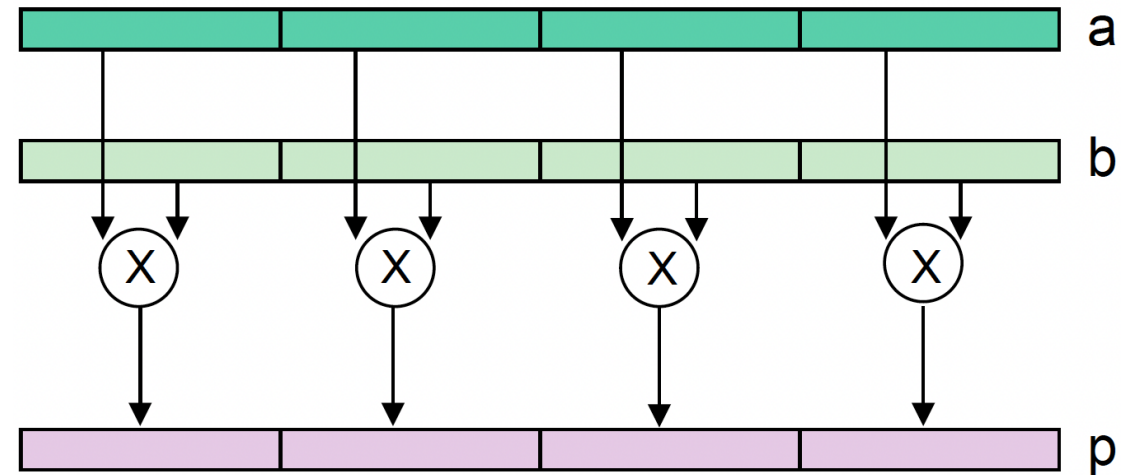
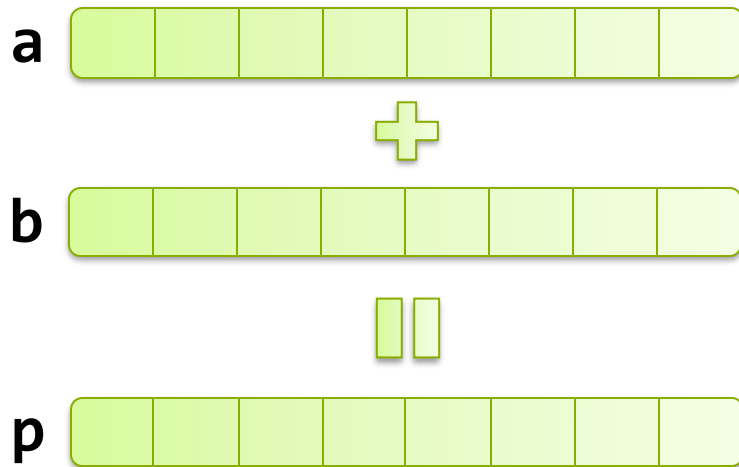


Example of SIMD
Unprocessable Patterns

SIMD Instructions

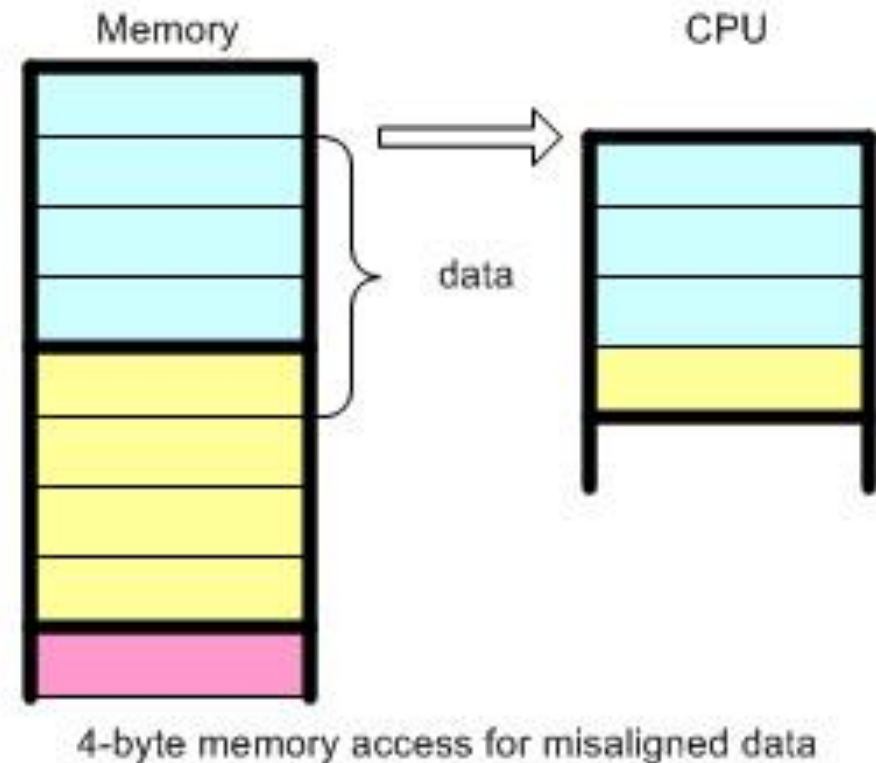
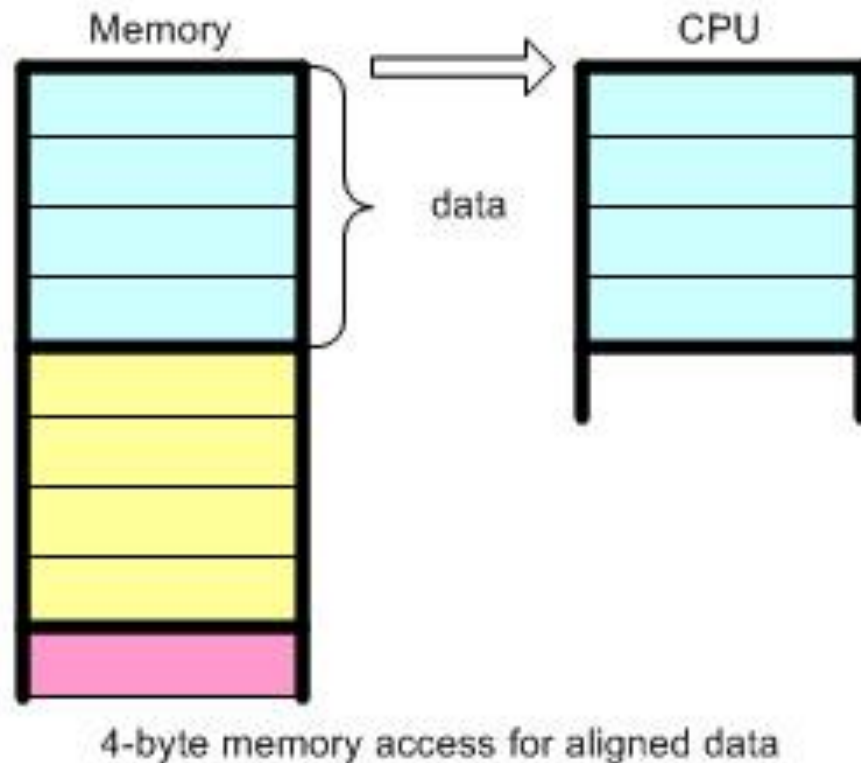
- SIMD instructions typically operate in an **elementwise** fashion
- All lanes perform **exactly the same operation** on each element

Vector Addition



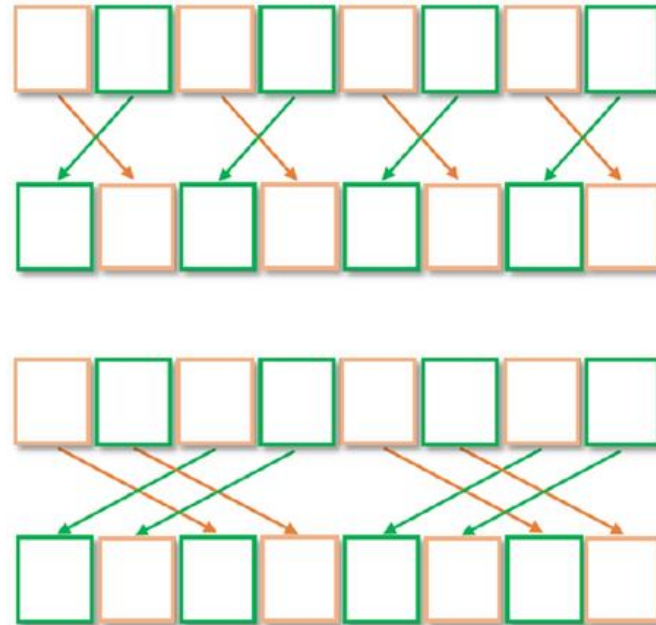
SIMD Instructions

- SIMD instructions typically operate in an **elementwise** fashion
- Vector memory operands might need to be **aligned**



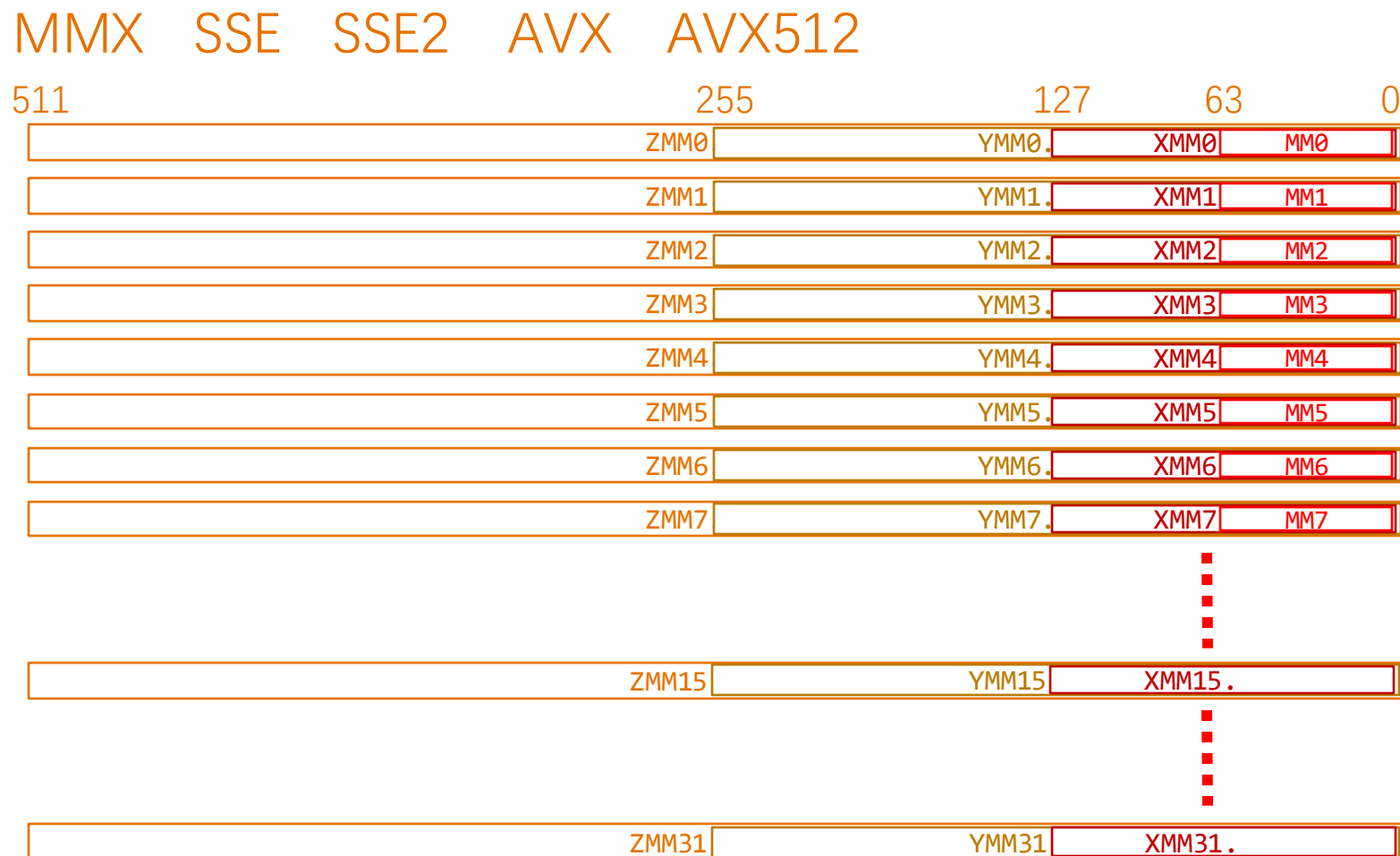
SIMD Instructions

- SIMD instructions typically operate in an **elementwise** fashion
- Some architectures support **cross-lane** operations
 - ▣ e.g. inserting or extracting subsets of vector elements
 - ▣ e.g. permuting (a.k.a., shuffling) the vector
 - ▣ e.g. scatter, or gather



Vector Register Aliasing

- Different names, but same registers



SIMD Data Types

- Each AVX register can have different data types
- Operation on 32 8-bit values in one instruction!

[illegible]

X86 SIMD INSTRUCTIONS



Instruction Naming Convention

- Format: `_mm<vec-width>_<op>_<scalar-ty>`
- `<vec-width>`: 64 / 128 / 256 / 512
- `<scalar-ty>`
 - ▣ `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)

Instruction Naming Convention

- 256 bit 4-wide double precision float addition

Operates on **ymm** registers

Add operation

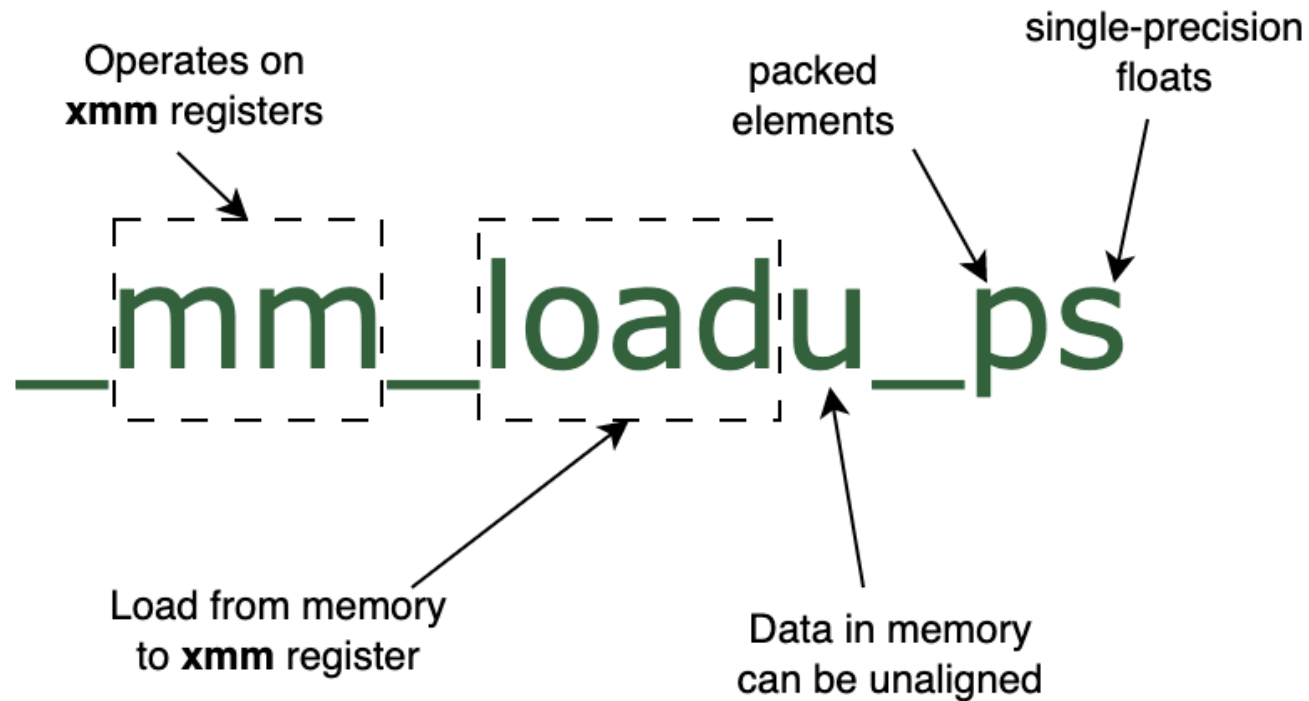
packed elements

double-precision floats

_mm256_add_pd

Instruction Naming Convention

- 128-bit 4-wide load (128 is omitted) for single precision float



Instruction Naming Convention

- More 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

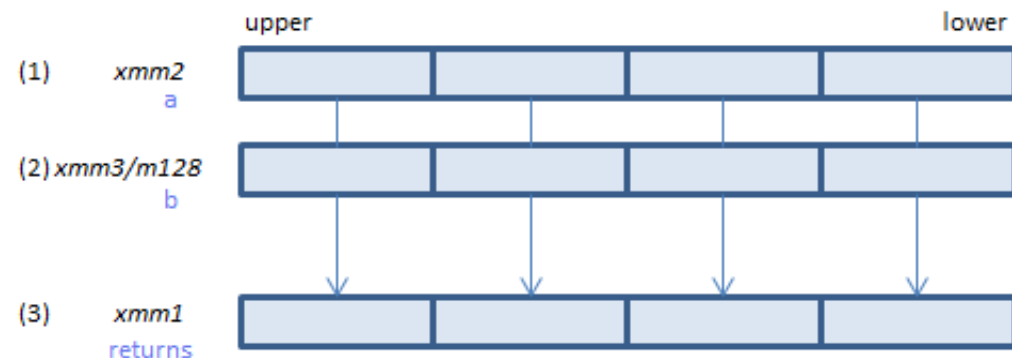
1. Arithmetic and Logic

- Types

- ▣ add, sub, mul, div, and, or, not, 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  
}
```



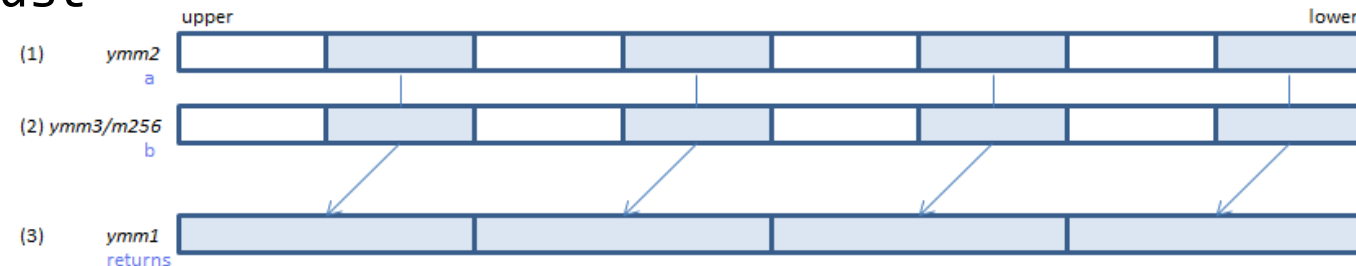
1. Arithmetic and Logic

- Types

- mul

- Example

```
__m256i _mm_mul_epu32 (__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

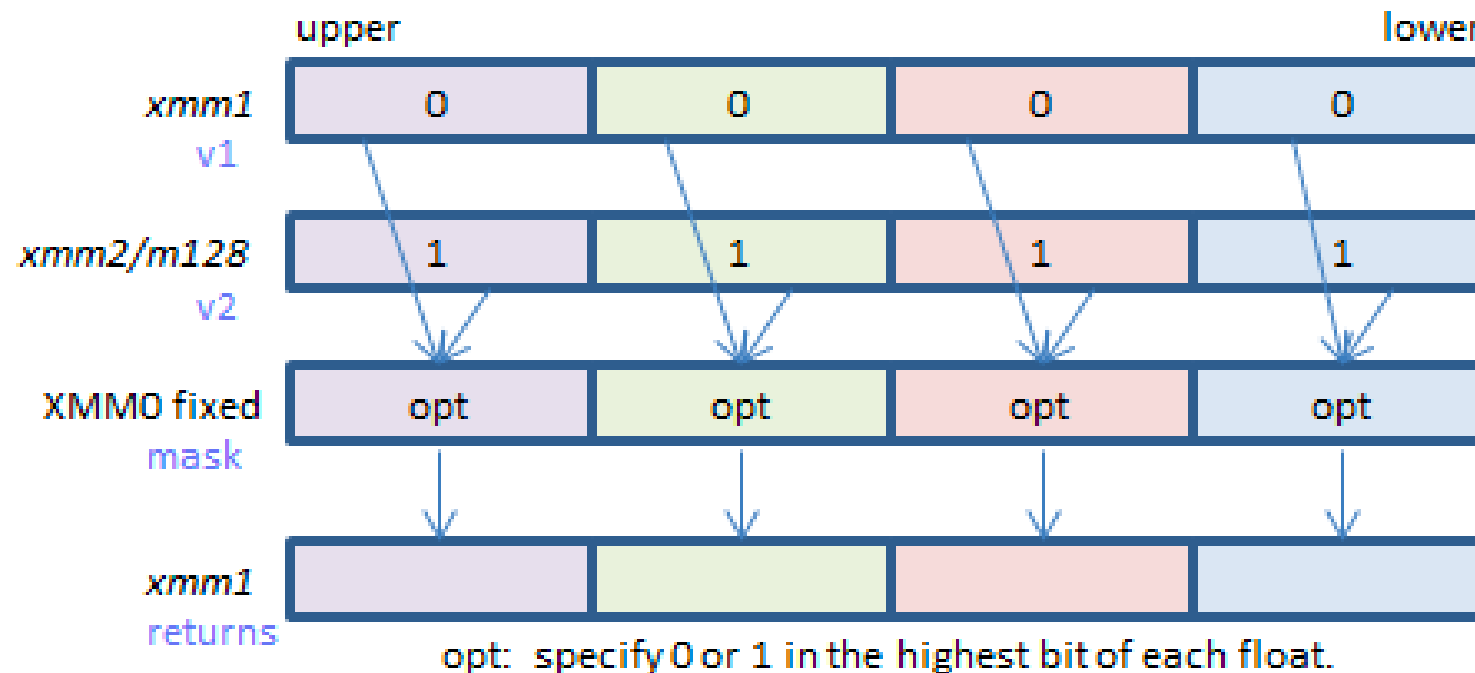
- **Example**

```
__m256 _mm256_loadu_ps (float const * mem_addr) {  
    dst[255:0] := MEM[mem_addr+255:mem_addr]  
    dst[MAX:256] := 0  
}
```

4. Shuffle: Blending

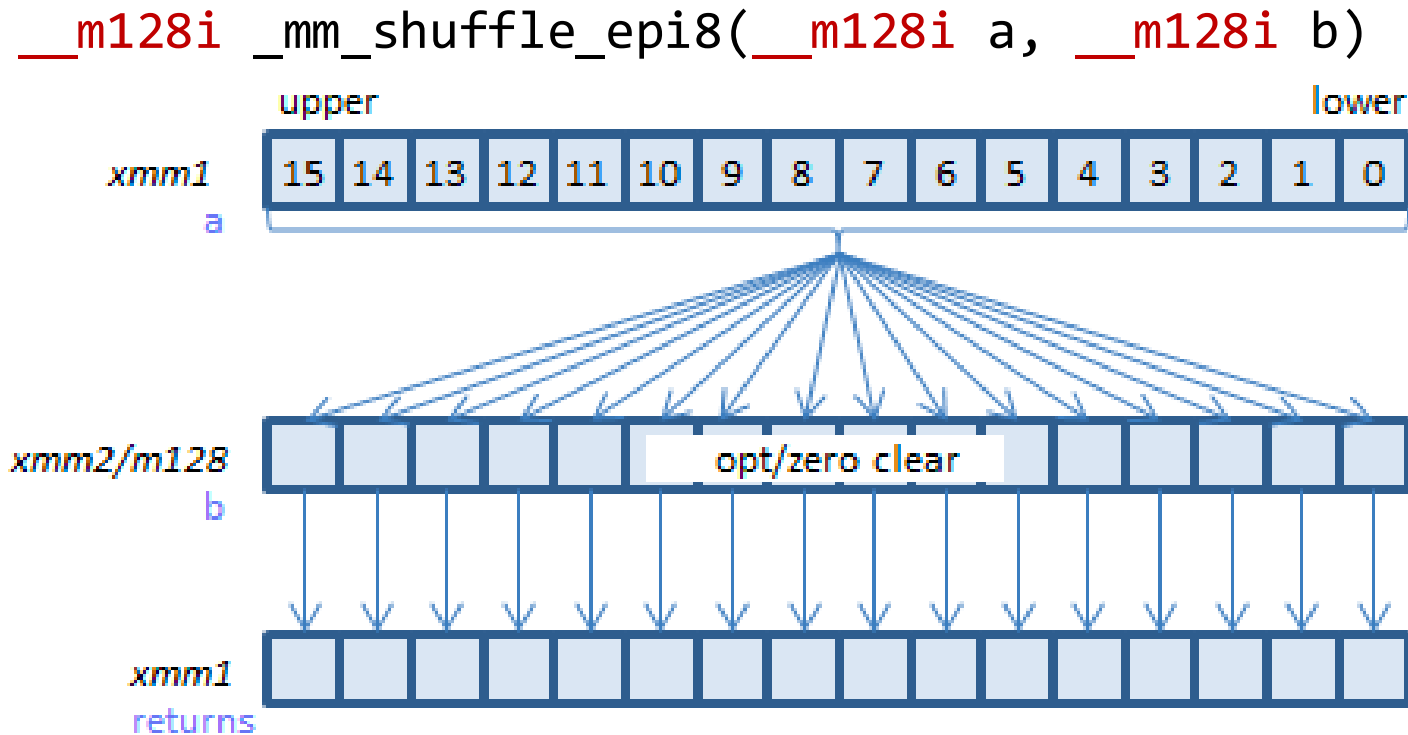
- Shuffle operation: `blend`
- **Example**

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



4. Shuffle: all-to-all shuffle

- Shuffle operation: shuffle
- Example



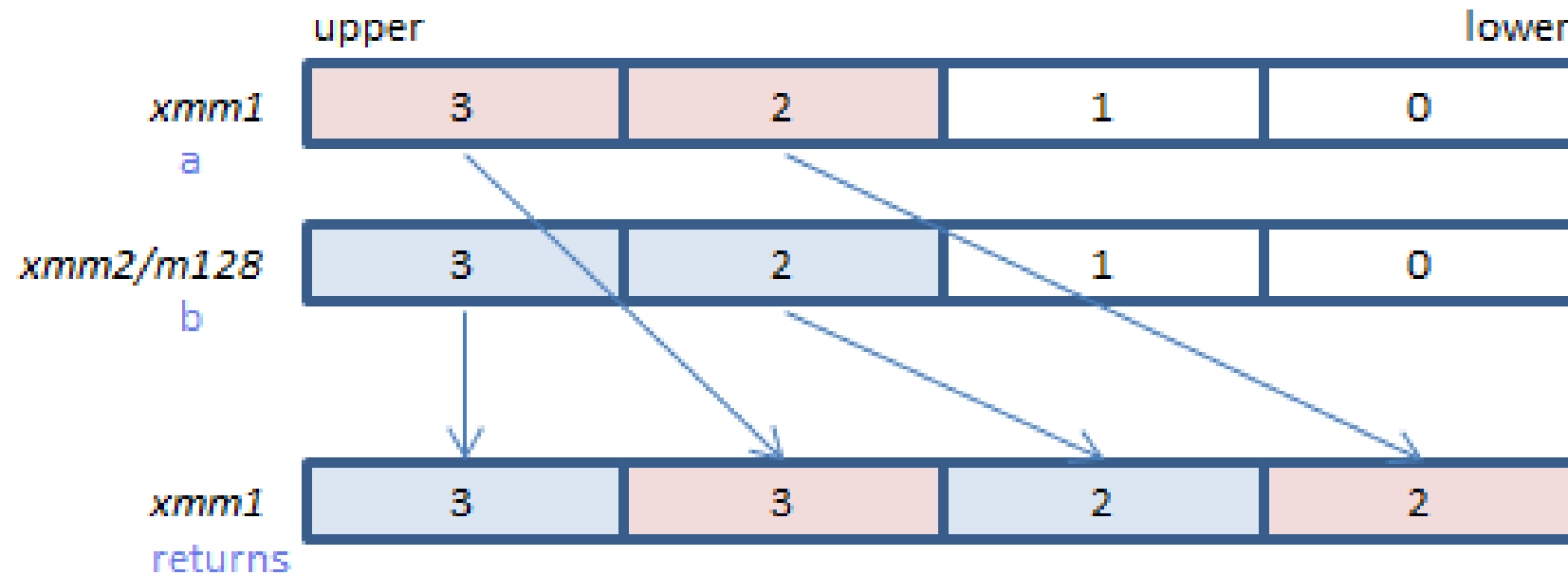
each byte of xmm2/m128:
bit 7 == 0 specifies copying, bit 3:0 specifies 0 to 15
bit 7 == 1 specifies zero clearing

4. Shuffle: unpack

- Shuffle operation: `unpackhi` & `unpacklo`

- **Example**

`__m128 _mm_unpackhi_ps(__m128 a, __m128 b)`

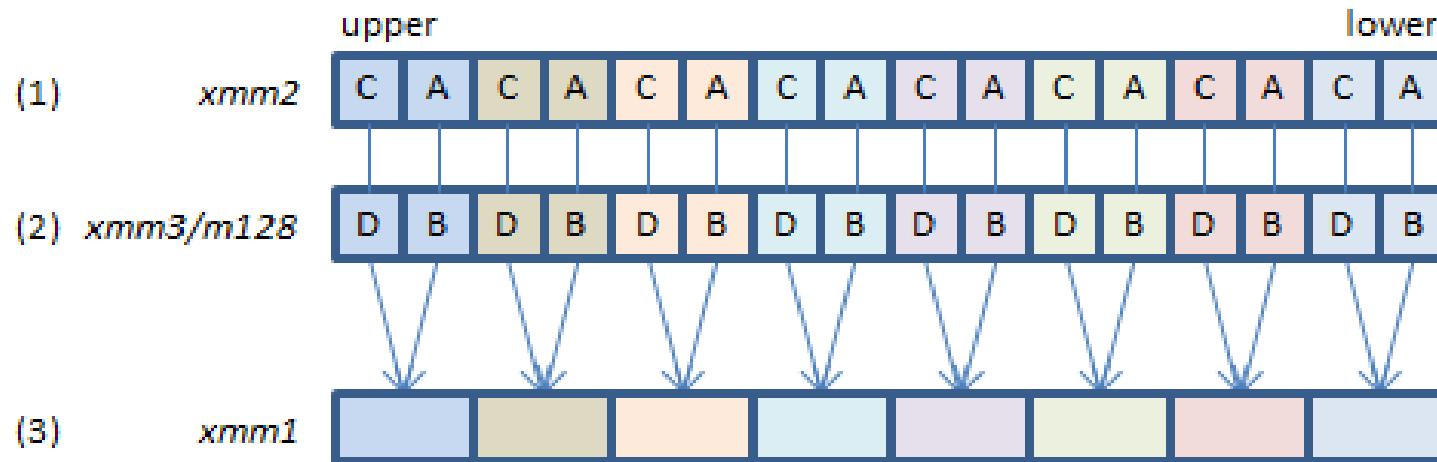


5. Non-SIMD: dot product instructions

- Non-SIMD operation: madd

- Example

`__m128i _mm_maddubs_epi16(__m128i a, __m128i b)`



Calculate $A*B+C*D$ and set to each WORD of (3). Signed saturation: set -32768 / 32767 on overflow.
Each BYTE of (1) is unsigned. Each BYTE of (2) is signed.

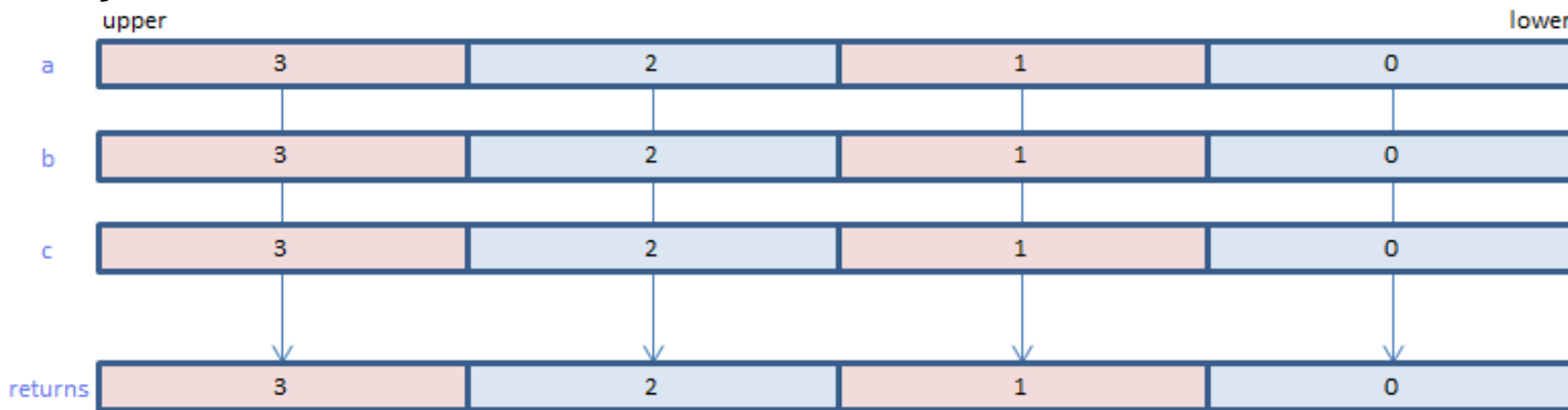
5. Non-SIMD: addsub

$$(a+bi)(c+di) = (ac - bd) + (ad + bc)i$$

- Non-SIMD operation: 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..

Intel® Intrinsic Guide

Updated
08/10/2022

Version
3.6.3

Instruction Set

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☐ AVX
- ☐ AVX2
- ☐ FMA
- ☐ AVX_VNNI
- ☐ AVX-512
- ☐ KNC
- ☐ AMX
- ☐ SVM
- ☐ 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

Search Intel Intrinsics

```
void __mm_2intersect_epi32 (__m128i a, __m128i b, __mmask8* k1, __mmask8* k2) vp2intersectd
void __mm256_2intersect_epi32 (__m256i a, __m256i b, __mmask8* k1, __mmask8* k2) vp2intersectd
void __mm512_2intersect_epi32 (__m512i a, __m512i b, __mmask16* k1, __mmask16* k2) vp2intersectd
void __mm_2intersect_epi64 (__m128i a, __m128i b, __mmask8* k1, __mmask8* k2) vp2intersectq
void __mm256_2intersect_epi64 (__m256i a, __m256i b, __mmask8* k1, __mmask8* k2) vp2intersectq
void __mm512_2intersect_epi64 (__m512i a, __m512i b, __mmask8* k1, __mmask8* k2) vp2intersectq
__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 <immintrin.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
        addr := 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) v4fmaddss
__m512 __mm512_4fmadd_ps (__m512 src, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b) v4fmaddps
```

> 7000 Intrinsics!

Give Feedback

**LET IT BE VECTORIZED
(BY THE COMPILER)**



Vectorizing Compiler

- Two types of Vectorizers
 - ▣ Loop Vectorizer
 - ▣ 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;  
}
```


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

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];  
    tc  = c[i];  
    t   = tb + tc;  
    a[i] = t;  
    tb2 = b[i+1];  
    tc2 = c[i+1];  
    t2  = tb2 + tc2;  
    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;  
}
```

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+1] = 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

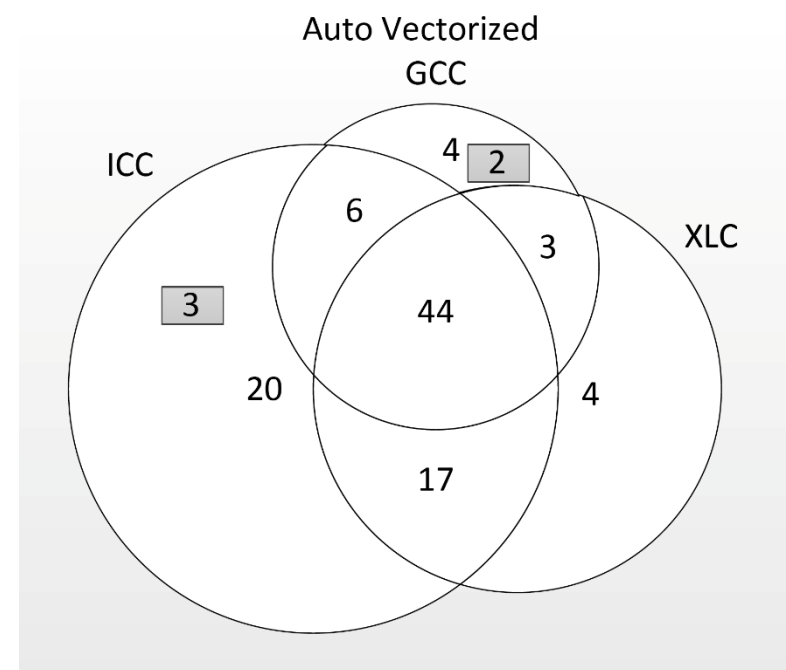
```
for (int i = 0; i < n; i++) {  
    tb    = b[i];  
    tc    = c[i];  
    t     = tb + tc;  
    a[i]  = t;  
}
```

```
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. Garzam, T. Wong and D. A. Padua, "An Evaluation of Vectorizing Compilers," 2011 International Conference on Parallel Architectures and Compilation Techniques, 2011

How good are the Vectorizers: A Case Study

- *A matrix-vector multiplication kernel from TVM*

```
void
dot_16x16x16_uint8_int8_int32(
  uint8_t data[restrict 4],
  int8_t kernel[restrict 16][4],
  int32_t output[restrict 16]) {
  for (int i = 0; i < 16; i++)
    for (int k = 0; k < 4; k++)
      output[i] +=
        data[k] * kernel[i][k];
}
```

ICC (1x)

```
movzx r11d, [rdi]
movsx eax, [rsi]
imul r11d, eax
movsx ecx, [1+rsi]
movzx r8d, [1+rdi]
imul r8d, ecx
movzx r10d, [2+rdi]
movsx r9d, [2+rsi]
imul r10d, r9d
movsx eax, [3+rsi]
movzx ecx, [3+rdi]
imul ecx, eax
add r11d, [rdx]
add r11d, r8d
add r11d, r10d
add r11d, ecx
mov [rdx], r11d
movzx r9d, [rdi]
movsx r11d, [4+rsi]
imul r9d, r11d
movsx eax, [5+rsi]
movzx r8d, [1+rdi]
imul r8d, eax
movzx eax, [2+rdi]
movsx ecx, [6+rsi]
imul eax, ecx
movsx r10d, [7+rsi]
movzx r11d, [3+rdi]
imul r11d, r10d
add r9d, [4+rdx]
add r9d, r8d
add r9d, eax
add r9d, r11d
mov [4+rdx], r9d
movzx r8d, [rdi]
movsx r9d, [8+rsi]
imul r8d, r9d
movsx eax, [9+rsi]
movzx ecx, [1+rdi]
imul ecx, eax
movzx r10d, [2+rdi]
movsx r9d, [10+rsi]
imul r10d, r9d
movsx r11d, [11+rsi]
movzx eax, [3+rdi]
imul eax, r11d
add r8d, [8+rdx]
add r8d, ecx
add r8d, r10d
add r8d, eax
mov [8+rdx], r8d
movzx ecx, [rdi]
movsx r8d, [12+rsi]
```

GCC (1.5x)

```
vmovdqa xmm0, .LC0[rip]
vmovdqu xmm1, [rsi]
vmovdqu xmm3, [rsi+16]
vmovdqu xmm2, [rsi+32]
vmovdqu xmm6, [rsi+48]
vpand xmm10, xmm0, xmm1
vpsrlw xmm1, xmm1, 8
movzx r9d, [rdi]
vpand xmm4, xmm0, xmm3
vpsrlw xmm3, xmm3, 8
movzx ecx, [rdi+2]
movzx eax, [rdi+3]
vpacusbw xmm4, xmm10, xmm4
vpacusbw xmm1, xmm1, xmm3
vpand xmm10, xmm0, xmm2
movzx r8d, [rdi+1]
vpand xmm3, xmm0, xmm6
vpsrlw xmm2, xmm2, 8
vmovd xmm8, r9d
vpacusbw xmm3, xmm10, xmm3
vpsrlw xmm6, xmm6, 8
vpand xmm10, xmm0, xmm4
vpacusbw xmm2, xmm2, xmm6
vpsrlw xmm4, xmm4, 8
vpand xmm6, xmm0, xmm3
vpsrlw xmm3, xmm3, 8
vpacusbw xmm10, xmm10, xmm6
vpacusbw xmm8, xmm8, xmm8
vpacusbw xmm6, xmm4, xmm3
vpand xmm3, xmm0, xmm1
vpand xmm0, xmm0, xmm2
vpacusbw xmm3, xmm3, xmm0
vpsrldq xmm0, xmm10, 8
vpmovsxbw xmm4, xmm10
vmovd xmm7, r8d
vpmullw xmm4, xmm4, xmm8
vpmullw xmm0, xmm0, xmm8
vpsrlw xmm2, xmm2, 8
vpbroadcastw xmm7, xmm7
vpmovsxbw xmm8, xmm3
vpsrldq xmm3, xmm3, 8
vmovd xmm9, ecx
vpmullw xmm8, xmm8, xmm7
vpsrlw xmm1, xmm1, 8
vpbroadcastw xmm9, xmm9
vpmovsxbw xmm3, xmm3
vpacusbw xmm1, xmm1, xmm2
vmovd xmm5, eax
vpmullw xmm3, xmm3, xmm7
vpmovsxbw xmm7, xmm6
vpbroadcastw xmm5, xmm5
vpmullw xmm7, xmm7, xmm9
vpsrldq xmm2, xmm6, 8
vpacusbw xmm6, xmm1
```

Clang/LLVM (2.2x)

```
movzx eax, [rdi + 3]
vpbroadcastd zmm8, eax
movzx eax, [rdi + 2]
vpbroadcastd zmm9, eax
movzx eax, [rdi + 1]
vpbroadcastd zmm10, eax
movzx eax, [rdi]
vpbroadcastd zmm11, eax
vmovdqu xmm0, [rsi]
vmovdqu xmm1, [rsi + 16]
vmovdqu xmm6, [rsi + 32]
vmovdqu xmm7, [rsi + 48]
vmovdqa xmm2, [rip + .LCPI0_0]
vpshufb xmm3, xmm7, xmm2
vpshufb xmm2, xmm6, xmm2
vpunpckldq xmm2, xmm2, xmm3
vmovdqa xmm3, [rip + .LCPI0_1]
vpshufb xmm4, xmm1, xmm3
vpshufb xmm3, xmm0, xmm3
vpunpckldq xmm3, xmm3, xmm4
vpblendd xmm12, xmm3, xmm2, 12
vmovdqa xmm3, [rip + .LCPI0_2]
vpshufb xmm4, xmm7, xmm3
vpshufb xmm3, xmm6, xmm3
vpunpckldq xmm3, xmm3, xmm4
vmovdqa xmm4, [rip + .LCPI0_3]
vpshufb xmm5, xmm1, xmm4
vpshufb xmm4, xmm0, xmm4
vpunpckldq xmm4, xmm4, xmm5
vpblendd xmm3, xmm4, xmm3, 12
vmovdqa xmm4, [rip + .LCPI0_4]
vpshufb xmm5, xmm7, xmm4
vpshufb xmm4, xmm6, xmm4
vpunpckldq xmm4, xmm4, xmm5
vmovdqa xmm5, [rip + .LCPI0_5]
vpshufb xmm2, xmm1, xmm5
vpshufb xmm5, xmm0, xmm5
vpunpckldq xmm2, xmm5, xmm2
vpblendd xmm2, xmm2, xmm4, 12
vmovdqa xmm4, [rip + .LCPI0_6]
vpshufb xmm5, xmm7, xmm4
vpshufb xmm4, xmm6, xmm4
vpunpckldq xmm4, xmm4, xmm5
vmovdqa xmm5, [rip + .LCPI0_7]
vpshufb xmm1, xmm1, xmm5
vpshufb xmm0, xmm0, xmm5
vpunpckldq xmm0, xmm0, xmm1
vpblendd xmm0, xmm0, xmm4, 12
vpmovsxbd zmm1, xmm12
vpmulld zmm1, zmm11, zmm1
vpaddd zmm1, zmm1, [rdx]
vpmovsxbd zmm3, xmm3
vpmulld zmm3, zmm10, zmm3
vpmovsxbd zmm2, xmm2
vpmulld zmm2, zmm9, zmm2
```

VeGen (11x)

```
vmovdqu64 zmm0, [rdx]
vpbroadcastd zmm1, [rdi]
vpdpbusd zmm0, zmm0, [rsi]
vmovdqu64 [rdx], zmm0
```


What makes it hard for the compiler to vectorize

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

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

What makes it hard for the compiler to vectorize

Go to godbolt.org and try

`-O3 -mavx512vnni -ffast-math -fno-unroll-loops`

1	Loop with unknown trip count	https://www.godbolt.org/z/MacxsGWsv
2	Cannot prove independence	https://www.godbolt.org/z/8fYcPz3Wv
3	Loop carried dependences	https://www.godbolt.org/z/z146WG3de
4	Reductions	https://www.godbolt.org/z/4n8zEbbsT
5	Control-flow	https://www.godbolt.org/z/jojKnEnv4
6	Non-contiguous data	https://www.godbolt.org/z/PzYc9jqaf https://www.godbolt.org/z/ME15GTesM
7	Cost model failures	https://www.godbolt.org/z/r5szGsaK4
8	Vectorization of function calls	https://www.godbolt.org/z/5Mcq9Pors
9	Different hardware versions	https://www.godbolt.org/z/Pvn1KM3rx

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;  
}
```

1. Loop with unknown trip count

```
1 bar:                                     # @bar
2     cmpl    %edx, %esi
3     jge     .LBB0_8
4     movslq  %esi, %rsi
5     movslq  %edx, %rax
6     movq    %rax, %rcx
7     subq    %rsi, %rcx
8     cmpq    $8, %rcx
9     jae     .LBB0_3
10    movq    %rsi, %r8
11    jmp     .LBB0_14
12 .LBB0_3:
13    cmpq    $16, %rcx
14    jae     .LBB0_9
15    xorl    %edx, %edx
16    jmp     .LBB0_5
17 .LBB0_9:
18    movq    %rcx, %rdx
19    andq    $-16, %rdx
20    vbroadcastss %xmm0, %zmm1
21    leaq    (%rdi,%rsi,4), %r8
22    xorl    %r9d, %r9d
23 .LBB0_10:                                # =>This Inner Loop Header:
24    vmovups  %zmm1, (%r8,%r9,4)
25    addq    $16, %r9
26    cmpq    %r9, %rdx
27    jne     .LBB0_10
28    cmpq    %rdx, %rcx
29    j       .LBB0_8
30    testb   $8, %cl
31    jne     .LBB0_5
32    addq    %rsi, %rdx
33    movq    %rdx, %r8
34    jmp     .LBB0_14
35 .LBB0_5:
36    movq    %rcx, %r9
37    andq    $-8, %r9
38    leaq    (%r9,%rsi), %r8
39    vbroadcastss %xmm0, %ymm1
40    leaq    (%rdi,%rsi,4), %rsi
41 .LBB0_6:                                # =>This Inner Loop Header:
42    vmovups  %ymm1, (%rsi,%rdx,4)
43    addq    $8, %rdx
44    cmpq    %rdx, %r9
45    jne     .LBB0_6
46    cmpq    %r9, %rcx
47    je      .LBB0_8
48 .LBB0_14:                                # =>This Inner Loop Header:
49    vmovss   %xmm0, (%rdi,%r8,4)
50    incq    %r8
51    cmpq    %r8, %rax
52    jne     .LBB0_14
53 .LBB0_8:
54    vzeroupper
55    retq

1 bar:                                     # @bar
2     testl   %esi, %esi
3     jle     .LBB0_13
4     movl    %esi, %eax
5     cmpl    $8, %esi
6     jae     .LBB0_3
7     xorl    %ecx, %ecx
8     jmp     .LBB0_12
9 .LBB0_3:
10    cmpl    $16, %esi
11    jae     .LBB0_5
12    xorl    %ecx, %ecx
13    jmp     .LBB0_9
14 .LBB0_5:
15    movl    %eax, %ecx
16    andl    $-16, %ecx
17    vbroadcastss %xmm0, %zmm1
18    leaq    (,%rax,4), %rdx
19    andq    $-64, %rdx
20    xorl    %esi, %esi
21 .LBB0_6:                                # =>This Inner Loop Header: Depth=1
22    vmovups  %zmm1, (%rdi,%rsi)
23    addq    $64, %rsi
24    cmpq    %rsi, %rdx
25    jne     .LBB0_6
26    cmpq    %rax, %rcx
27    je      .LBB0_13
28    testb   $8, %al
29    je      .LBB0_12
30 .LBB0_9:
31    movq    %rcx, %rdx
32    movl    %eax, %ecx
33    andl    $-8, %ecx
34    vbroadcastss %xmm0, %ymm1
35 .LBB0_10:                                # =>This Inner Loop Header: Depth=1
36    vmovups  %ymm1, (%rdi,%rdx,4)
37    addq    $8, %rdx
38    cmpq    %rdx, %rcx
39    jne     .LBB0_10
40    cmpq    %rax, %rcx
41    je      .LBB0_13
42 .LBB0_12:                                # =>This Inner Loop Header: Depth=1
43    vmovss   %xmm0, (%rdi,%rcx,4)
44    incq    %rcx
45    cmpq    %rcx, %rax
46    jne     .LBB0_12
47 .LBB0_13:
48    vzeroupper
49    retq

1 bar:                                     # @bar
2     vbroadcastss %xmm0, %zmm0
3     xorl    %eax, %eax
4 .LBB0_1:                                # =>This Inner Loop Header: Depth=1
5     vmovups  %zmm0, (%rdi,%rax,4)
6     addq    $16, %rax
7     cmpq    $1024, %rax                # imm = 0x400
8     jne     .LBB0_1
9     vzeroupper
10    retq
```

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];  
}
```

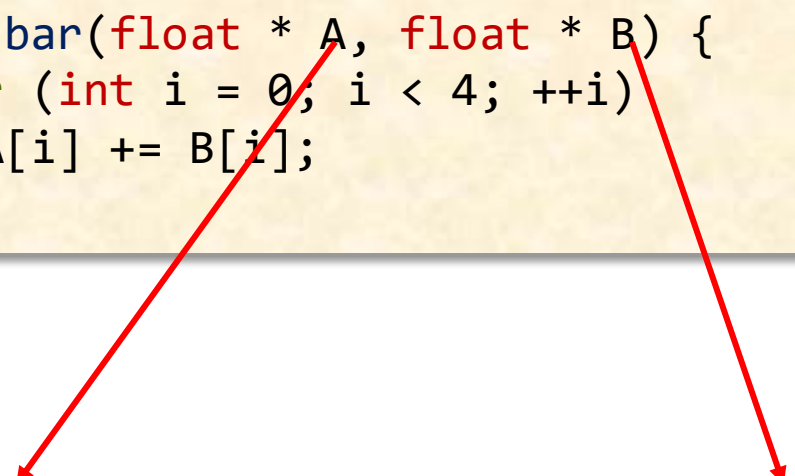


1	2	3	4	5	6	7	8	10	20	30	40	50	60	70	80
---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----

Serial execution: in scalar

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
---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----

Parallel execution: in vector of width 2

2. Cannot Prove Independence

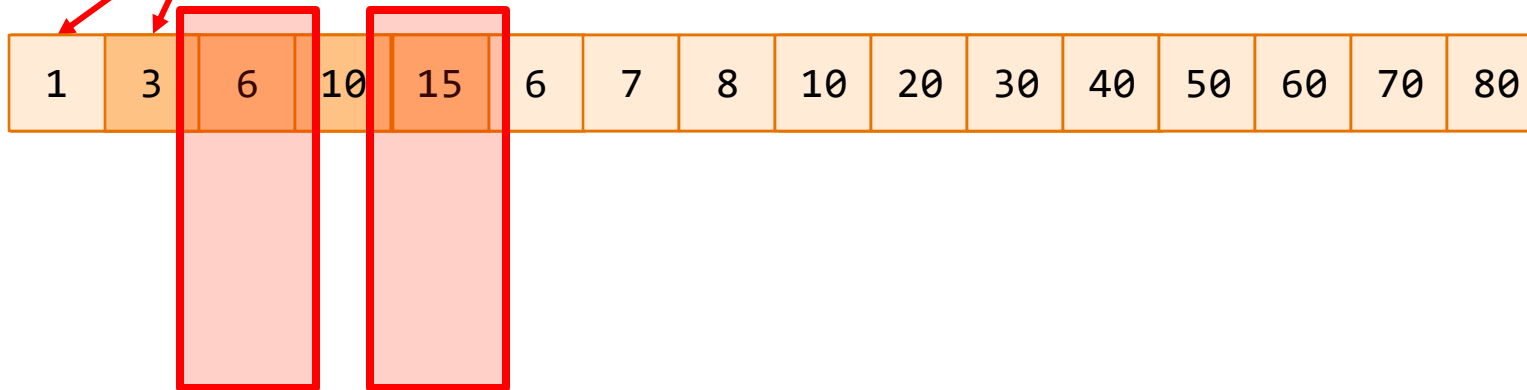
```
void bar(float * A, float * B) {  
    for (int i = 0; i < 4; ++i)  
        A[i] += B[i];  
}
```

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

Serial execution: in scalar

2. Cannot Prove Independence

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



Parallel execution: in vector of width 2

- Vector Results $\neq$ Scalar Results
- Memory overlap $\rightarrow$ 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!

2. Cannot Prove Independence

```
1  bar:                                     # @bar
2      leaq    4096(%rsi), %rax
3      cmpq    %rdi, %rax
4      jbe     .LBB0_2
5      leaq    4096(%rdi), %rax
6      cmpq    %rsi, %rax
7      jbe     .LBB0_2
8      xorl    %eax, %eax
9  .LBB0_6:                                # =>This Inner Loop Header: Depth=1
10     vmovss   (%rdi,%rax,4), %xmm0        # xmm0 = mem[0],zero,zero,zero
11     vaddss   (%rsi,%rax,4), %xmm0, %xmm0
12     vmovss   %xmm0, (%rdi,%rax,4)
13     incq     %rax
14     cmpq     $1024, %rax                 # imm = 0x400
15     jne      .LBB0_6
16     jmp      .LBB0_4
17 .LBB0_2:
18     xorl     %eax, %eax
19 .LBB0_3:                                # =>This Inner Loop Header: Depth=1
20     vmovups   (%rdi,%rax,4), %zmm0
21     vaddps   (%rsi,%rax,4), %zmm0, %zmm0
22     vmovups   %zmm0, (%rdi,%rax,4)
23     addq     $16, %rax
24     cmpq     $1024, %rax                 # imm = 0x400
25     jne      .LBB0_3
26 .LBB0_4:
27     vzeroupper
28     retq
```

sequential

parallel

2. Cannot Prove Independence

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

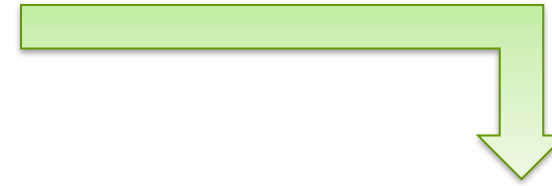
- LLVM can still vectorize!
- By checking if the address ranges overlap
 - ▣ Check the ranges first
 - ▣ If overlapping, use scalar implementation
 - ▣ If not overlapping, use the vectorized implementation
- However, Can save all these hassles by...

```
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      leaq    4096(%rsi), %rax
3      cmpq    %rdi, %rax
4      jbe     .LBB0_2
5      leaq    4096(%rdi), %rax
6      cmpq    %rsi, %rax
7      jbe     .LBB0_2
8      xorl    %eax, %eax
9  .LBB0_6:                                # =>This Inner Loop Header: Depth=1
10     vmovss   (%rdi,%rax,4), %xmm0        # xmm0 = mem[0],zero,zero,zero
11     vaddss   (%rsi,%rax,4), %xmm0, %xmm0
12     vmovss   %xmm0, (%rdi,%rax,4)
13     incq     %rax
14     cmpq     $1024, %rax                 # imm = 0x400
15     jne     .LBB0_6
16     jmp      .LBB0_4
17 .LBB0_2:
18     xorl     %eax, %eax
19 .LBB0_3:                                # =>This Inner Loop Header: Depth=1
20     vmovups   (%rdi,%rax,4), %zmm0
21     vaddps    (%rsi,%rax,4), %zmm0, %zmm0
22     vmovups   %zmm0, (%rdi,%rax,4)
23     addq      $16, %rax
24     cmpq      $1024, %rax                # imm = 0x400
25     jne      .LBB0_3
26 .LBB0_4:
27     vzeroupper
28     retq
```

Using restrict



```
bar:                                     # @bar
      xorl     %eax, %eax
.LBB0_1:                                # =>This Inner Loop Header
      vmovups   (%rdi,%rax,4), %zmm0
      vaddps    (%rsi,%rax,4), %zmm0, %zmm0
      vmovups   %zmm0, (%rdi,%rax,4)
      addq      $16, %rax
      cmpq      $1024, %rax                # imm = 0x400
      jne      .LBB0_1
      vzeroupper
      retq
```

3. Loop Carried Dependence

```
void bar(float * restrict A, int n) {  
    for (int i = 1; i < 1024; ++i)  
        A[i] += A[i-1];  
                A[i-2];  
                A[i-100];  
                A[i-128];  
}
```

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

3. Loop Carried Dependence

```
bar:                                # @bar
    vmovss    -4(%rdi), %xmm0        # xmm0 = mem[0],zero,zero,zero
    xorl     %eax, %eax
.LBB0_1:                             # =>This Inner Loop Header: Depth=1
    vaddss    (%rdi,%rax,4), %xmm0, %xmm0
    vmovss    %xmm0, (%rdi,%rax,4)
    incq     %rax
    cmpq     $1024, %rax             # imm = 0x400
    jne      .LBB0_1
    retq
```

$A[i] += A[i-1];$

```
bar:                                # @bar
    vmovsd    -8(%rdi), %xmm0        # xmm0 = mem[0],zero
    xorl     %eax, %eax
.LBB0_1:                             # =>This Inner Loop Header: Depth=1
    vmovsd    (%rdi,%rax,4), %xmm1    # xmm1 = mem[0],zero
    vaddps    %xmm0, %xmm1, %xmm0
    vmovlps   %xmm0, (%rdi,%rax,4)
    addq     $2, %rax
    cmpq     $1024, %rax             # imm = 0x400
    jne      .LBB0_1
    retq
```

$A[i] += A[i-2];$

```
bar:                                # @bar
    xorl     %eax, %eax
.LBB0_1:                             # =>This Inner Loop Header: Depth=1
    vmovups    (%rdi,%rax,4), %xmm0
    vaddps    -400(%rdi,%rax,4), %xmm0, %xmm0
    vmovups    %xmm0, (%rdi,%rax,4)
    addq     $4, %rax
    cmpq     $1024, %rax             # imm = 0x400
    jne      .LBB0_1
    retq
```

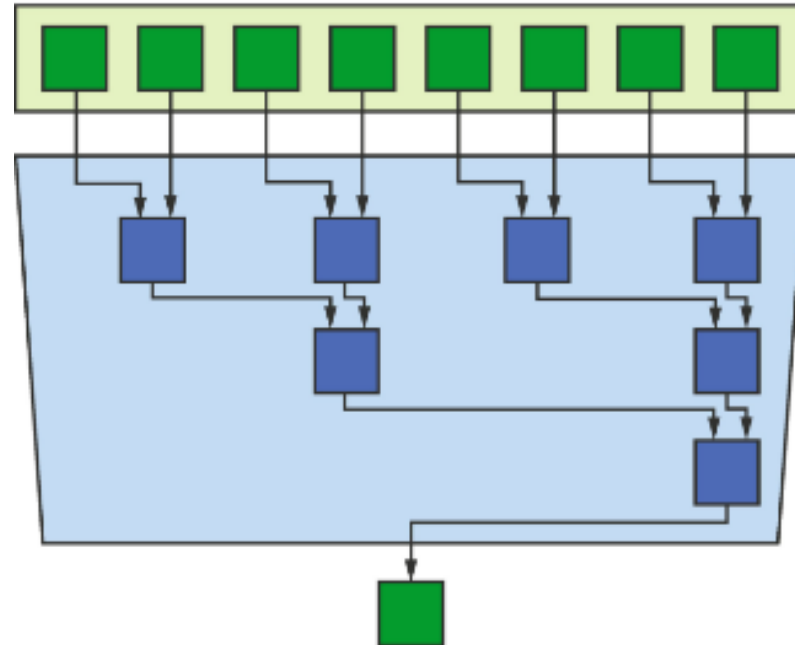
$A[i] += A[i-100];$

```
bar:                                # @bar
    xorl     %eax, %eax
.LBB0_1:                             # =>This Inner Loop Header: Depth=1
    vmovups    (%rdi,%rax,4), %zmm0
    vaddps    -512(%rdi,%rax,4), %zmm0, %zmm0
    vmovups    %zmm0, (%rdi,%rax,4)
    addq     $16, %rax
    cmpq     $1024, %rax             # imm = 0x400
    jne      .LBB0_1
    vzeroupper
    retq
```

$A[i] += A[i-128];$

4. Reductions

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



4. Reductions

```
.LCPI0_0:
    .long    5                                # 0x5
foo:
    # @foo
    vpxor    %xmm0, %xmm0, %xmm0
    xorl     %eax, %eax
    vpbroadcastd .LCPI0_0(%rip), %zmm1    # zmm1 = [5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5]
.LBB0_1:
    # =>This Inner Loop Header. Depth=1
    vpaddd   (%rdi,%rax,4), %zmm0, %zmm0
    vpaddd   %zmm1, %zmm0, %zmm0
    addq     $16, %rax
    cmpq     $1024, %rax                    # imm = 0x400
    jne      .LBB0_1
    vextracti64x4 $1, %zmm0, %ymm1
    vpaddd   %zmm1, %zmm0, %zmm0
    vextracti128 $1, %ymm0, %xmm1
    vpaddd   %xmm1, %xmm0, %xmm0
    vpshufd  $238, %xmm0, %xmm1             # xmm1 = xmm0[2,3,2,3]
    vpaddd   %xmm1, %xmm0, %xmm0
    vpshufd  $85, %xmm0, %xmm1              # xmm1 = xmm0[1,1,1,1]
    vpaddd   %xmm1, %xmm0, %xmm0
    vmovd    %xmm0, %eax
    vzeroupper
    retq
```

broadcast

16 elements at
a time

Reduction tree
on 16 elements

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;  
}
```

5. Control Flow

```
.LCPI0_0:
    .long    5                # 0x5
foo:
    vpxor    %xmm0, %xmm0, %xmm0
    xorl     %eax, %eax
    vpxor    %xmm1, %xmm1, %xmm1

.LBB0_1:
    # =>This Inner Loop Header: Depth=1
    vmovdqu64    (%rdi,%rax,4), %zmm2
    vpcmpgtd     %zmm0, %zmm2, %k1
    vpaddd      %zmm1, %zmm2, %zmm2
    vpaddd      .LCPI0_0(%rip){1to16}, %zmm2, %zmm1 {%k1}
    addq       $16, %rax
    cmpq       $1024, %rax          # imm = 0x400
    jne        .LBB0_1
    vextracti64x4    $1, %zmm1, %ymm0
    vpaddd      %zmm0, %zmm1, %zmm0
    vextracti128     $1, %ymm0, %xmm1
    vpaddd      %xmm1, %xmm0, %xmm0
    vpshufd     $238, %xmm0, %xmm1    # xmm1 = xmm0[2,3,2,3]
    vpaddd      %xmm1, %xmm0, %xmm0
    vpshufd     $85, %xmm0, %xmm1     # xmm1 = xmm0[1,1,1,1]
    vpaddd      %xmm1, %xmm0, %xmm0
    vmovd      %xmm0, %eax
    vzeroupper
    retq
```

add w/o
condition

Compute mask
 $A[i] > 0$?

Update value
only if mask set

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;  
    }  
}
```

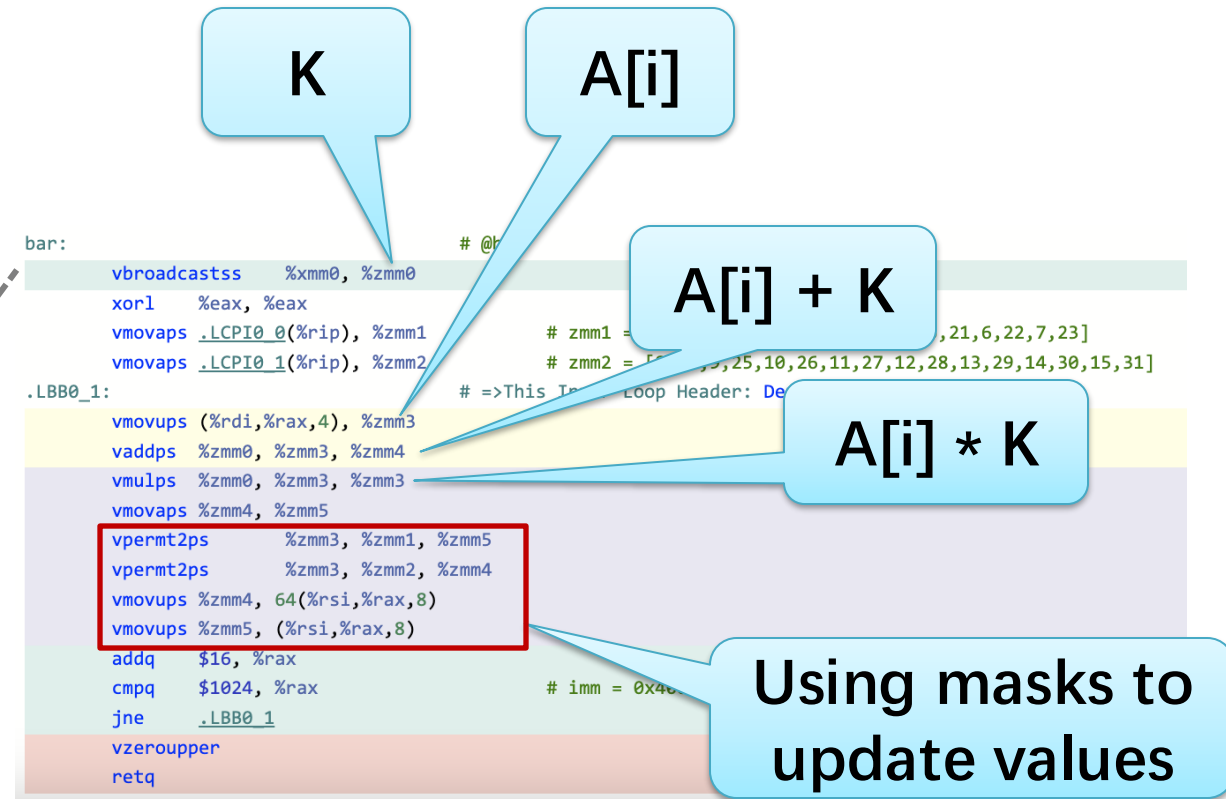
6. Non-contiguous data

```
.LCPI0_0:
.long 0          # 0x0
.long 16         # 0x10
.long 1          # 0x1
.long 17         # 0x11
.long 2          # 0x2
.long 18         # 0x12
.long 3          # 0x3
.long 19         # 0x13
.long 4          # 0x4
.long 20         # 0x14
.long 5          # 0x5
.long 21         # 0x15
.long 6          # 0x6
.long 22         # 0x16
.long 7          # 0x7
.long 23         # 0x17

.LCPI0_1:
.long 8          # 0x8
.long 24         # 0x18
.long 9          # 0x9
.long 25         # 0x19
.long 10         # 0xa
.long 26         # 0x1a
.long 11         # 0xb
.long 27         # 0x1b
.long 12         # 0xc
.long 28         # 0x1c
.long 13         # 0xd
.long 29         # 0x1d
.long 14         # 0xe
.long 30         # 0x1e
.long 15         # 0xf
.long 31         # 0x1f

bar:
    vbroadcastss    %xmm0, %zmm0
    xorl    %eax, %eax
    vmovaps .LCPI0_0(%rip), %zmm1    # zmm1 = [0,16,1,17,2,18,3,19,4,20,5,21,6,22,7,23]
    vmovaps .LCPI0_1(%rip), %zmm2    # zmm2 = [8,24,9,25,10,26,11,27,12,28,13,29,14,30,15,31]

.LBB0_1:
    vmovups (%rdi,%rax,4), %zmm3
    vaddps %zmm0, %zmm3, %zmm4
    vmulps %zmm0, %zmm3, %zmm3
    vmovaps %zmm4, %zmm5
    vpermt2ps %zmm3, %zmm1, %zmm5
    vpermt2ps %zmm3, %zmm2, %zmm4
    vmovups %zmm4, 64(%rsi,%rax,8)
    vmovups %zmm5, (%rsi,%rax,8)
    addq $16, %rax
    cmpq $1024, %rax    # imm = 0x400
    jne .LBB0_1
    vzeroupper
    retq
```



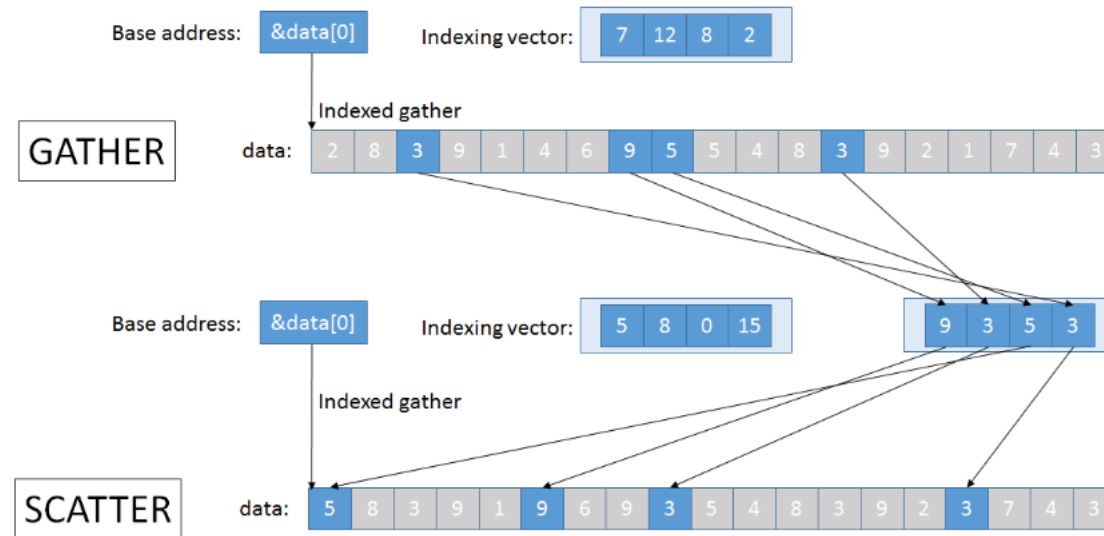
6. Non-contiguous data

```
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;  
    }  
}
```

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

Indirect Access: Use gather instruction

6. Non-contiguous data



```
bar:                                     # @bar
    xorl    %eax, %eax
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
    vmovups (%rdx,%rax,4), %zmm0
    kxnorw  %k0, %k0, %k1
    vxorps  %xmm1, %xmm1, %xmm1
    vgatherdps (%rdi,%zmm0,4), %zmm1 {%k1}
    vmovups %zmm1, (%rsi,%rax,4)
    addq    $16, %rax
    cmpq    $1024, %rax                  # imm = 0x400
    jne     .LBB0_1
    vzeroupper
    retq
```

gather from
memory

7. Cost Model Failures

- 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

```
void matmul(float a[restrict][N], float b[restrict][N],  
            float c[restrict][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];  
}
```


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_12
12    mov     r9d, edi
13    mov     r8d, r9d
14    and     r8d, 48
15    lea     r13, [4*r9]
16    mov     r10, r9
17    shl     r10, 5
18    xor     eax, eax
19    vscmps  xmm0, xmm0, xmm0
20    jmp     .LBB0_2
21    .LBB0_11:                             # In Loop: Header=BB0_2 Depth=1
22        mov     rax, qword ptr [rsp - 16]    # 8-byte Reload
23        add     rax, 1
24        add     r13, r13
25        cmp     rax, r9
26        je     .LBB0_12
27    .LBB0_2:                             # =>This Loop Header: Depth=1
28        mov     qword ptr [rsp - 16], rax    # 8-byte Spill
29        imul    rax, r9
30        mov     rcx, qword ptr [rsp - 24]    # 8-byte Reload
31        lea     rdx, [rcx + 4*rax]
32        mov     r15, qword ptr [rsp - 32]    # 8-byte Reload
33        xor     r12d, r12d
34        mov     qword ptr [rsp - 8], rdx    # 8-byte Spill
35        jmp     .LBB0_1
36    .LBB0_10:                             # In Loop: Header=BB0_3 Depth=2
37        vmovss  dword ptr [rdx + 4*r12], xmm1
38        add     r12, 1
39        add     r15, 4
40        cmp     r12, r9
41        je     .LBB0_11
42    .LBB0_3:                             # Parent Loop BB0_2 Depth=1
43        vmovss  xmm1, dword ptr [rdx + 4*r12]    # xmm1 = mem[0],zero,zero,zero
44        cmp     edi, 8
45        jne     .LBB0_5
46        xor     eax, eax
47        jmp     .LBB0_8
48    .LBB0_5:                             # In Loop: Header=BB0_3 Depth=2
49        vblendps xmm1, xmm0, xmm1, 1    # xmm1 = xmm1[0],xmm0[1,2,3]
50        mov     rbx, r15
51        xor     eax, eax
52    .LBB0_6:                             # Parent Loop BB0_2 Depth=1
53        lea     rbp, [rbx + r13]
54        mov     r15, rbp
55        add     r13, r13
56        lea     rcx, [r13 + r13]
57        lea     rdx, [rcx + r13]
58        lea     r14, [rdx + r13]
59        vmovss  xmm2, dword ptr [r13 + rcx]    # xmm2 = mem[0],zero,zero,zero
60        vinsertps xmm2, xmm2, dword ptr [r13 + rdx], 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[3]
62        add     r14, r13
63        vinsertps xmm2, xmm2, dword ptr [r13 + r14], 48 # xmm2 = xmm2[0,1,2],mem[0]
64        vmovss  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 [r13 + rbp], 32 # xmm3 = xmm3[0,1],mem[0],xmm3[3]
67        vinsertps xmm3, xmm3, dword ptr [r13 + r13], 48 # xmm3 = xmm3[0,1,2],mem[0]
68        vinsertf128 ymm2, ymm3, xmm2, 1
69        vmulps  ymm2, ymm2, ymmword ptr [rsi + 4*rax]
70        vaddps  ymm1, ymm1, ymm2
71        add     rax, 4
72        add     rbx, r10
73        cmp     r8, rax
74        jne     .LBB0_6
75        vextractf128 xmm2, ymm1, 1
76        vaddps  xmm1, xmm1, xmm2
77        vpermilpd xmm2, xmm1, 1    # xmm2 = xmm1[1,0]
78        vaddps  xmm1, xmm1, xmm2
79        vmovshdup xmm2, xmm1    # xmm2 = xmm1[1,1,3,3]
80        vaddss  xmm1, xmm1, xmm2
81        mov     rax, r8
82        cmp     r9, r9
83        mov     rdx, 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     rbx, [r15 + 4*rcx]
89    .LBB0_9:                             # Parent Loop BB0_2 Depth=1
90        vmovss  xmm2, dword ptr [rbx]    # xmm2 = mem[0],zero,zero,zero
91        vmulss  xmm2, xmm2, dword ptr [rsi + 4*rax]
92        vaddss  xmm1, xmm1, xmm2
93        add     rax, 1
94        add     rbx, r13
95        cmp     r9, rax
96        jne     .LBB0_2
97        jmp     .LBB0_10
98    .LBB0_12:
99        pop     rbx
100       pop     r12
101       pop     r13
102       pop     r14
103       pop     r15
104       pop     rbp
```

8. Vectorization of Function Calls

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

```
#include <math.h>

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

8. Vectorization of Function Calls

```
bar:                                     # @bar
    pushq    %r15
    pushq    %r14
    pushq    %rbx
    movq     %rsi, %rbx
    movq     %rdi, %r14
    xorl     %r15d, %r15d
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
    vmovups  (%r14,%r15,4), %ymm0
    callq    _ZGVdN8v_sinf@PLT
    vmovups  %ymm0, (%rbx,%r15,4)
    addq     $8, %r15
    cmpq     $1024, %r15                # imm = 0x400
    jne      .LBB0_1
    popq     %rbx
    popq     %r14
    popq     %r15
    vzeroupper
    retq
```

9. Different Hardware Versions

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

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

9. Different Hardware Versions

The image displays four screenshots of x86-64 assembly code generated by clang 16.0.0, illustrating different hardware versions and their corresponding assembly instructions. Each screenshot shows a loop with a red box highlighting the increment instruction.

Top Left Screenshot: x86-64 clang 16.0.0 (Editor #1) -O3 -fno-vectorize -ffast-math -fno-unroll-loops. The assembly code shows a loop header with `Depth=1`. The increment instruction is `incq %rax`.

Top Right Screenshot: x86-64 clang 16.0.0 (Editor #1) -O3 -msse2 -ffast-math -fno-unroll-loops. The assembly code shows a loop header with `Depth=1`. The increment instruction is `addq $4, %rax`.

Bottom Left Screenshot: x86-64 clang 16.0.0 (Editor #1) -O3 -mavx2 -ffast-math -fno-unroll-loops. The assembly code shows a loop header with `Depth=1`. The increment instruction is `addq $8, %rax`.

Bottom Right Screenshot: x86-64 clang 16.0.0 (Editor #1) -O3 -mavx512vnni -ffast-math -fno-unroll-loops. The assembly code shows a loop header with `Depth=1`. The increment instruction is `addq $16, %rax`.

Aside: Do I Have SIMD Capabilities?

```
$ less /proc/cpuinfo
```

```
flags              : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat p
se36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm con
stant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmp
erf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx1
6 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f
16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd ibrs ibp
b stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2
erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves
dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp flush_l1d
```

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++) {  
        y[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 % vec_len; 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++) {  
        y[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) {  
    // v_0_4 = {v3, v2, v1, v0}  
    __m128 v_0_4 = _mm256_extracti128_si256(v, 0);  
    // v_4_8 = {v7, v6, v5, v4}  
    __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_cvtss_f32(t4);  
}
```

Vectorizing Control Flow

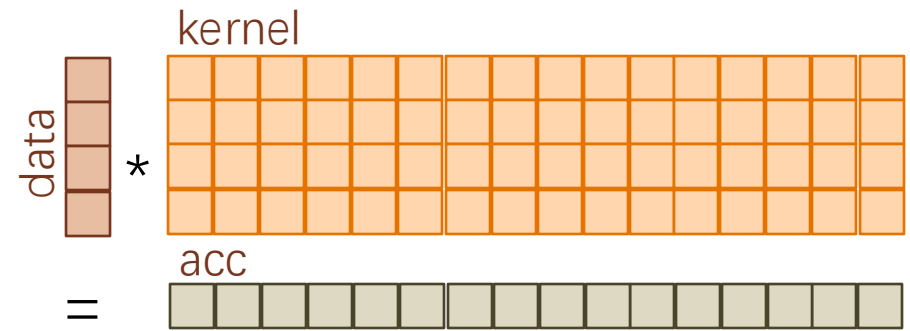
```
void foo(float *restrict x, float *restrict y) {  
    for (int i = 0; i < 8; i++) {  
        if (x[i] < y[i])  
            x[i] += y[i];  
        else  
            x[i] -= y[i];  
    }  
}
```

```
__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];  
}
```

- Compiler vectorizes the inner loop
 - ▣ Similar to a reduction

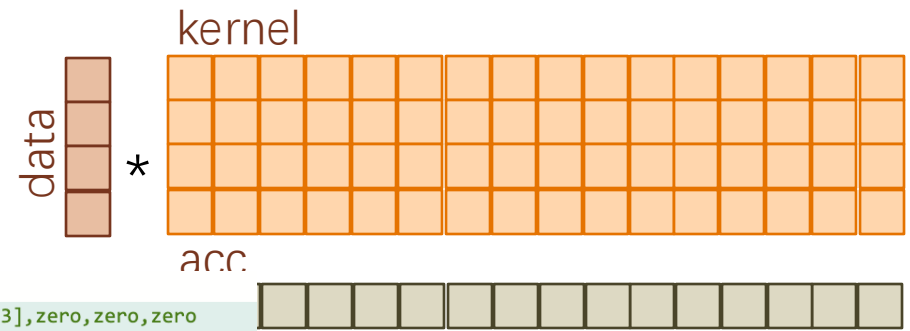


2.2x faster than scalar

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];  
}
```

- Compiler vectorizes the inner loop
 - ▣ Similar to a reduction



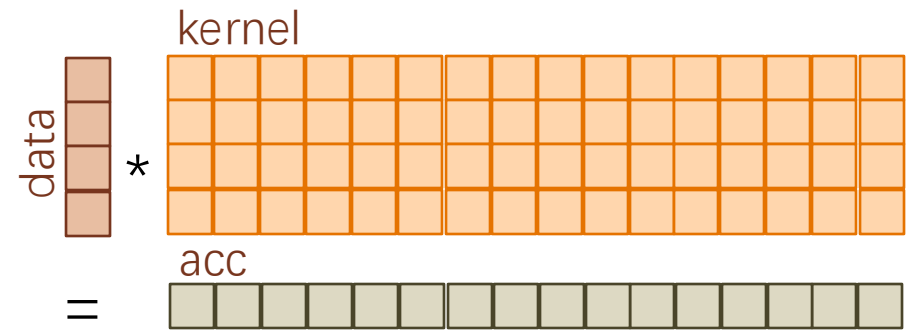
```
dot:                                # @dot  
    vpmovzxbd    (%rdi), %xmm0      # xmm0 = mem[0],zero,zero,zero,mem[1],zero,zero,zero,mem[2],zero,zero,zero,mem[3],zero,zero,zero  
    xorl        %eax, %eax  
.  
LBB0_1:                            # =>This Inner Loop Header: Depth=1  
    vmovd       (%rdx,%rax,4), %xmm1 # xmm1 = mem[0],zero,zero,zero  
    vpmovsxbd    (%rsi,%rax,4), %xmm2  
    vpmaddwd     %xmm0, %xmm2, %xmm2  
    vpaddq      %xmm1, %xmm2, %xmm1  
    vpsltd      $238, %xmm1, %xmm2   # xmm2 = xmm1[2,3,2,3]  
    vpaddq      %xmm2, %xmm1, %xmm1  
    vpsltd      $85, %xmm1, %xmm2    # xmm2 = xmm1[1,1,1,1]  
    vpaddq      %xmm2, %xmm1, %xmm1  
    vmovd       %xmm1, (%rdx,%rax,4)  
    incq        %rax  
    cmpq        $16, %rax  
    jne         .LBB0_1  
    retq
```

faster than scalar

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];  
}
```

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



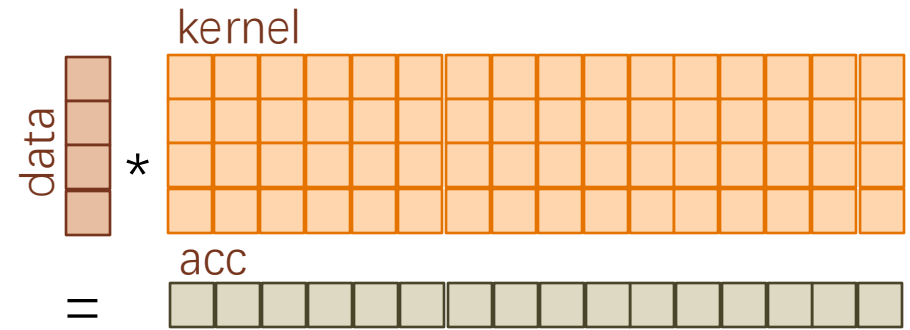
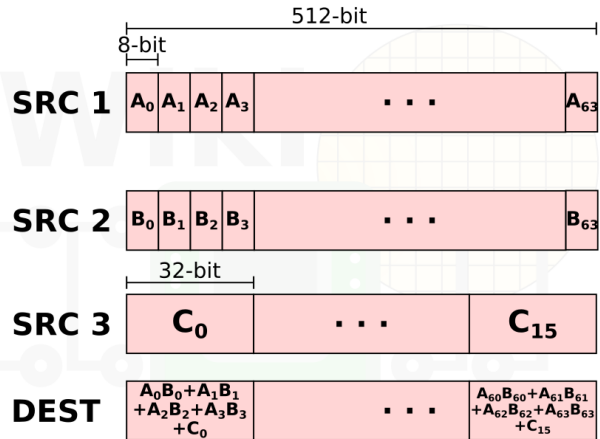
2.2x faster than scalar

```
__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_epi32(acc_vec, data_vec, kernel_vec);
```

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

```
__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_epi32(acc_vec, data_vec, kernel_vec);
```

11x faster than scalar

Summary

- Vector hardware is great for performance
 - ▣ SIMD: Single Instruction Multiple Data
 - ▣ Architecture is changing / improving quickly (latest SIMD generation)
- 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