Software Performance Engineering

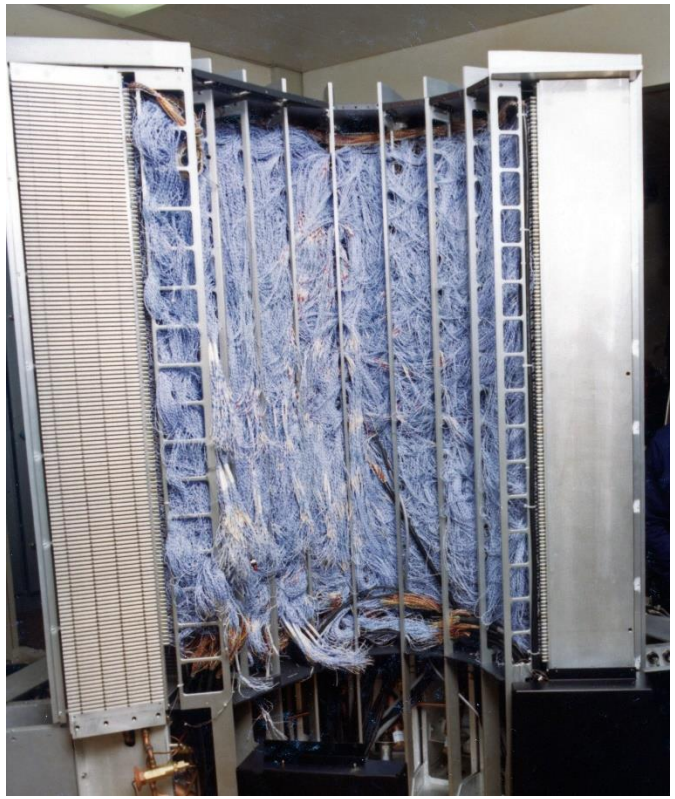
LECTURE 5
Vectorization

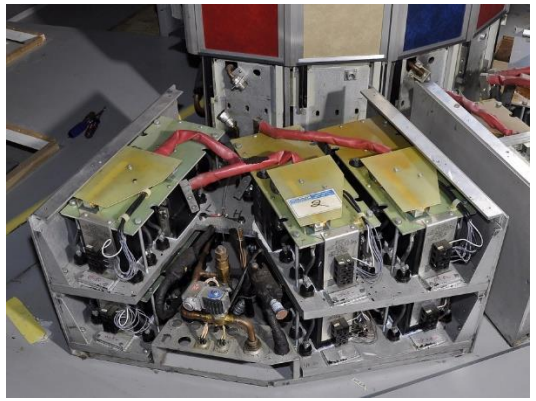
Xuhao Chen

September 18, 2025

Vectorization

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

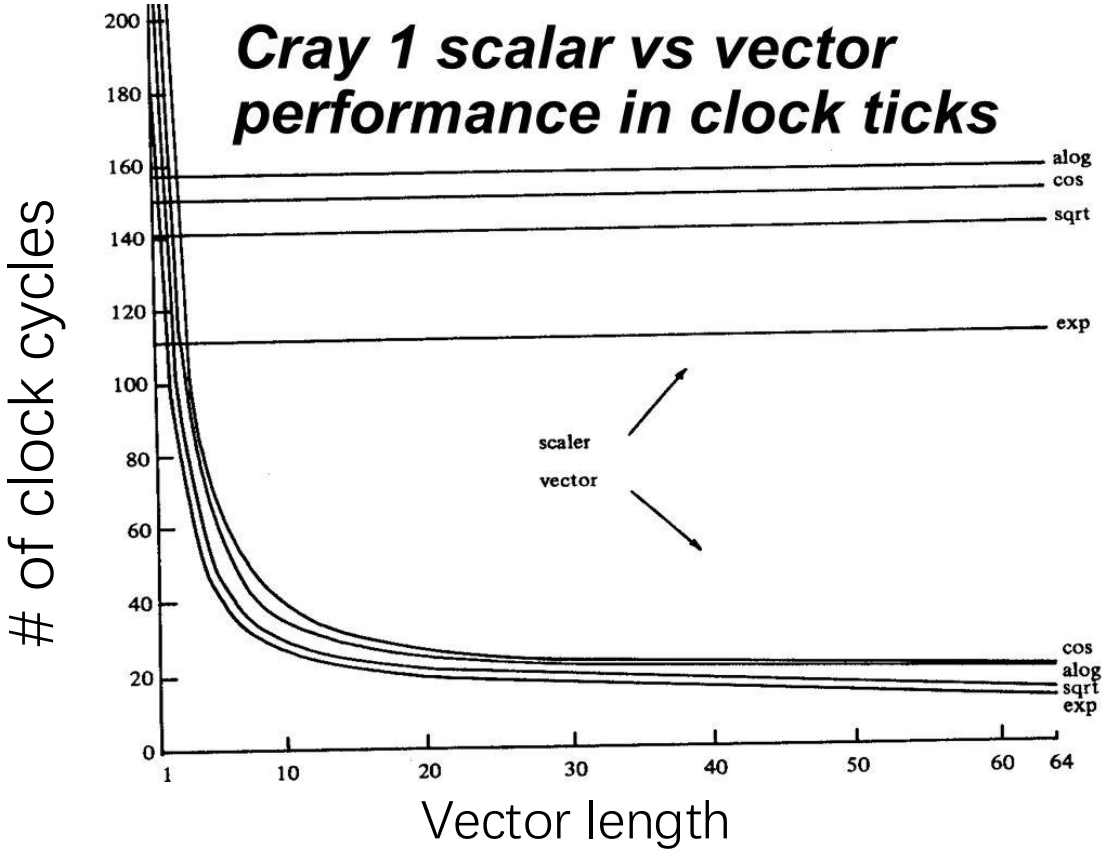




Seymour Cray

Vectorization

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



Seymour Cray

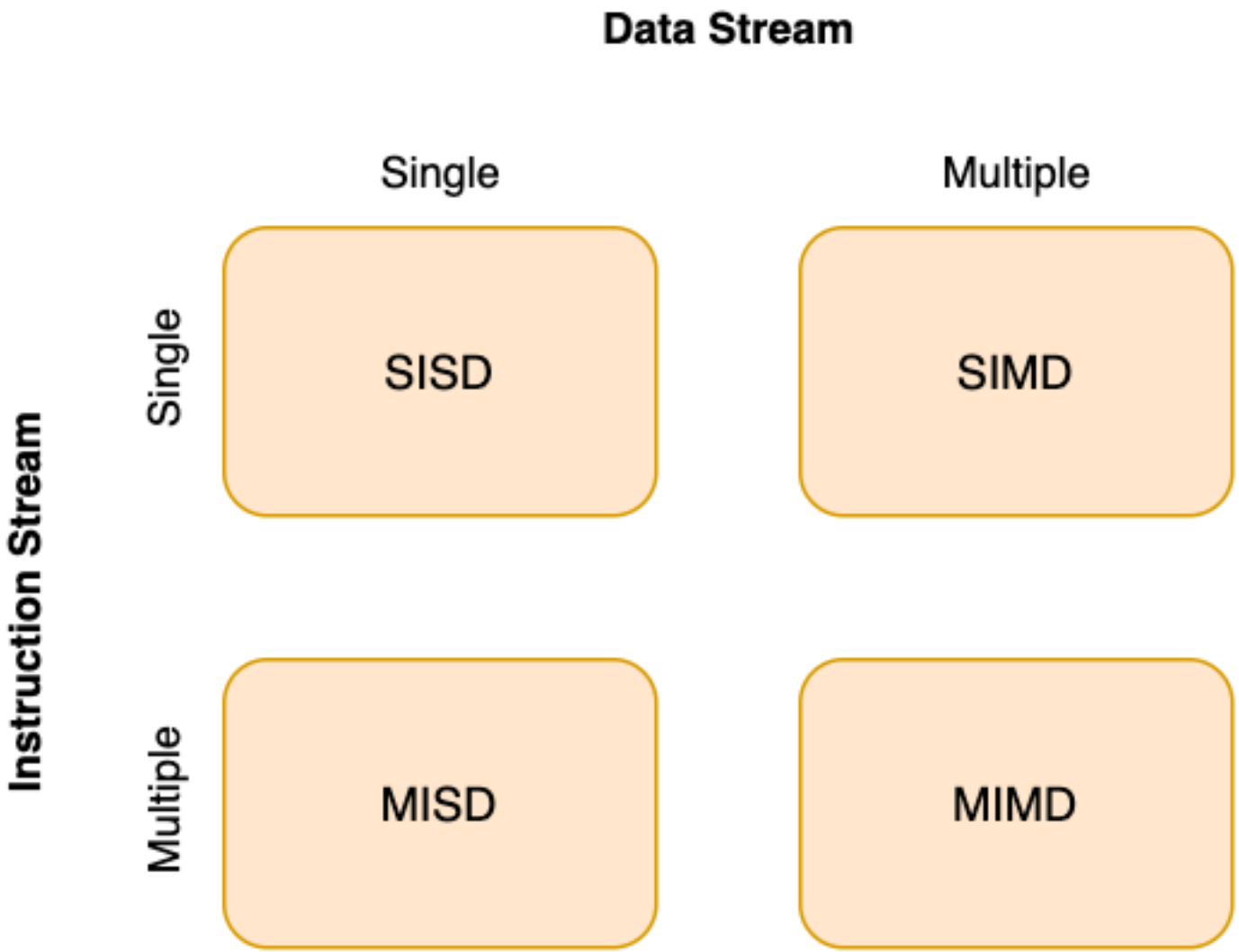
Multimedia Instructions

- 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	1013	256 bits
Intel	AVX512	2015	512 bits
Intel	AVX512-VNNI	2021	512 bits

What is SIMD?

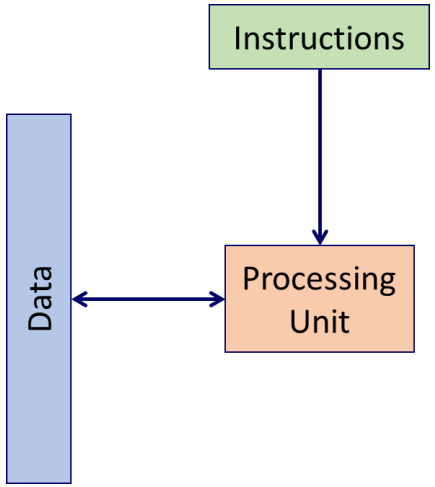
Flynn Taxonomy (1966)

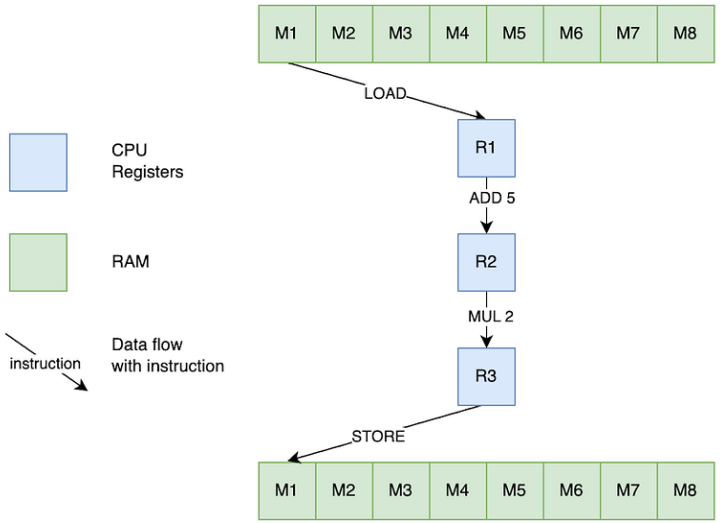


Flynn Taxonomy (1966)

Single-Instruction Single-Data

(e.g. Single-Core Processors)

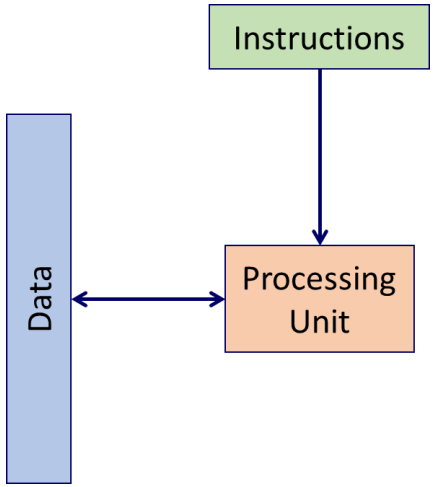


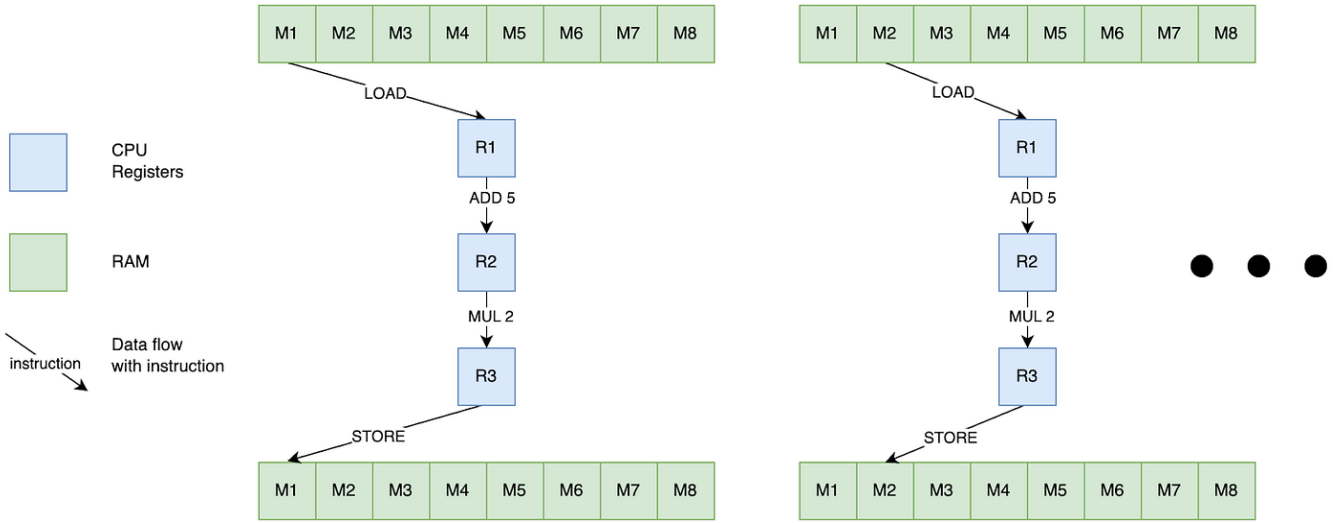


Flynn Taxonomy (1966)

Single-Instruction Single-Data

(e.g. Single-Core Processors)

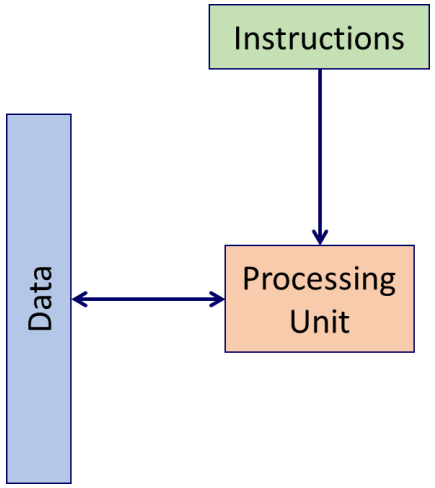


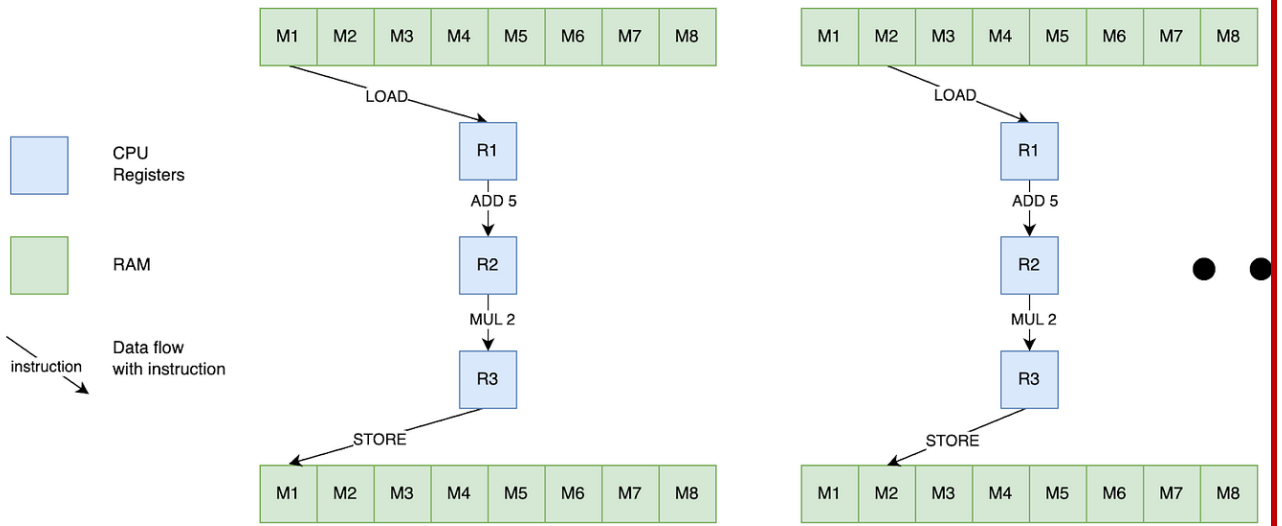


Flynn Taxonomy (1966)

Single-Instruction Single-Data

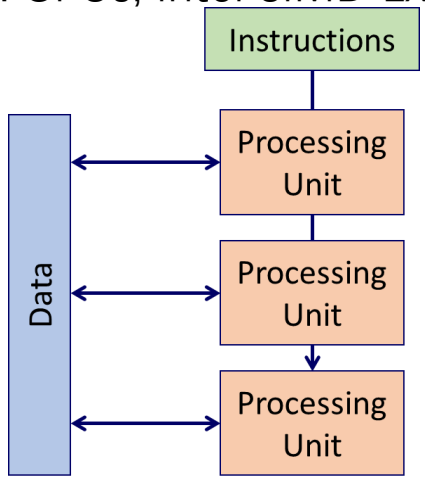
(e.g. Single-Core Processors)

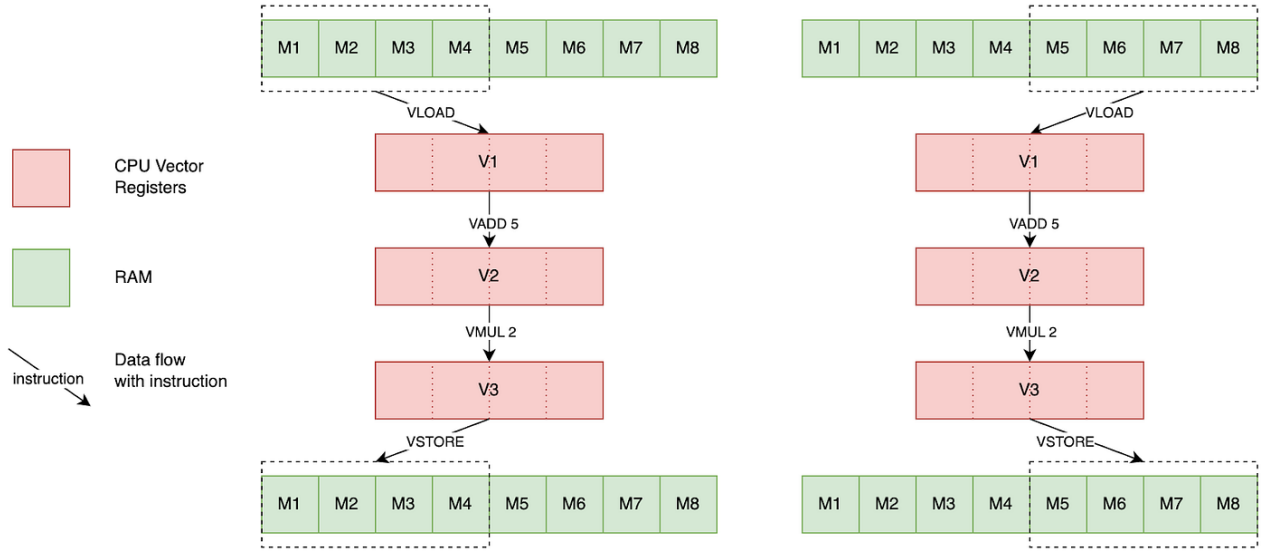




Single-Instruction Multi-Data

(e.g. GPUs, Intel SIMD Extensions)

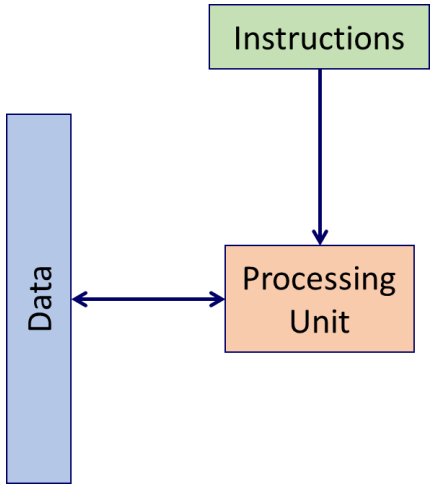




Flynn Taxonomy (1966)

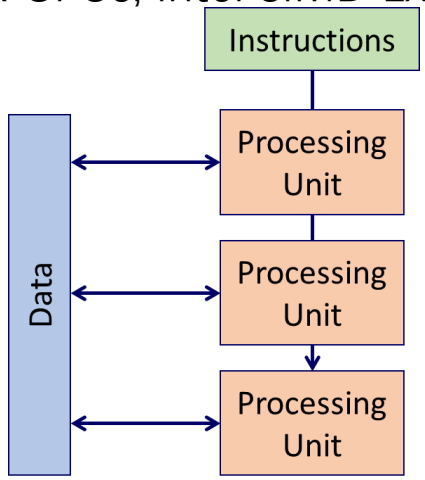
Single-Instruction Single-Data

(e.g. Single-Core Processors)



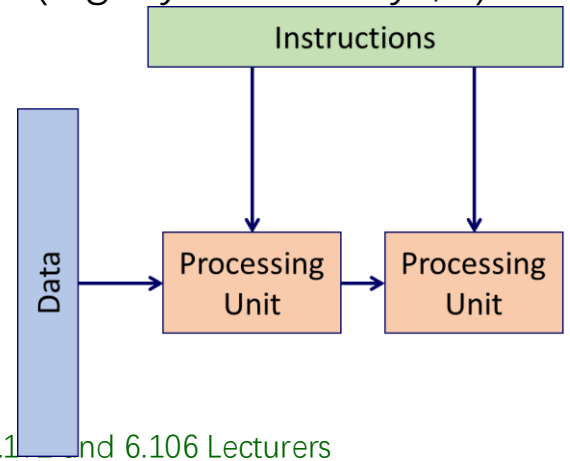
Single-Instruction Multi-Data

(e.g. GPUs, Intel SIMD Extensions)



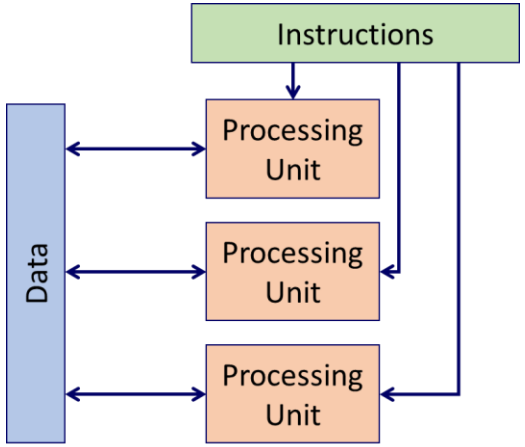
Multi-Instruction Single-Data

(e.g. Systolic Arrays,⋯)



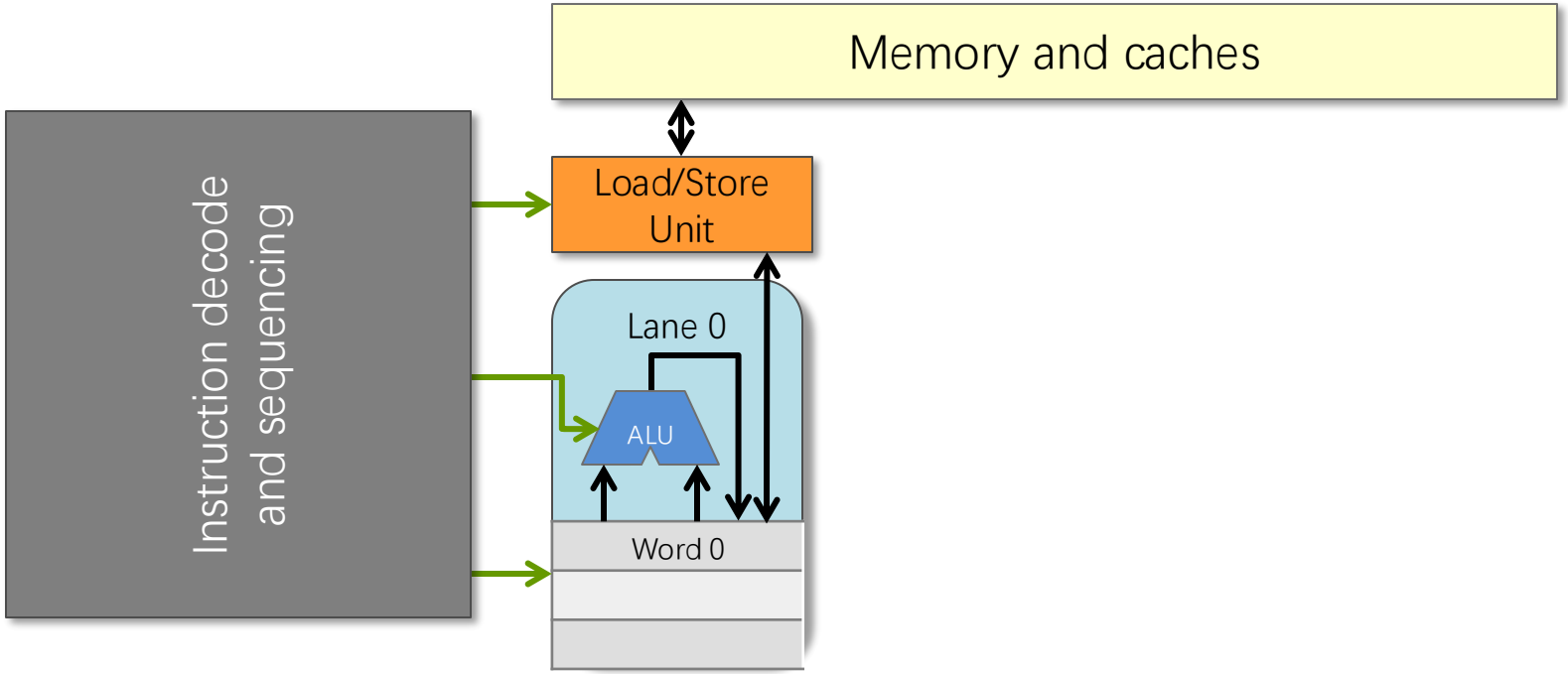
Multi-Instruction Multi-Data

(e.g. Parallel Computers)



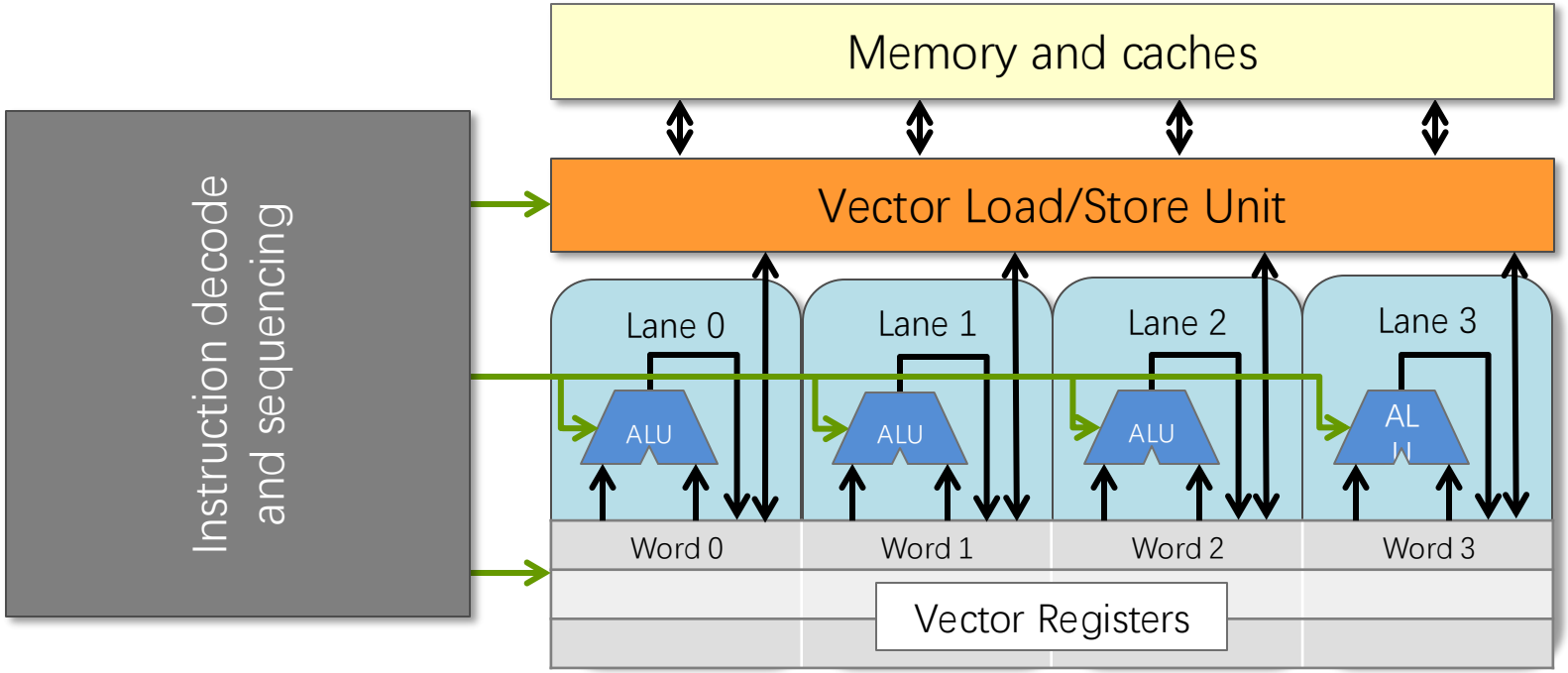
Scalar Hardware

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



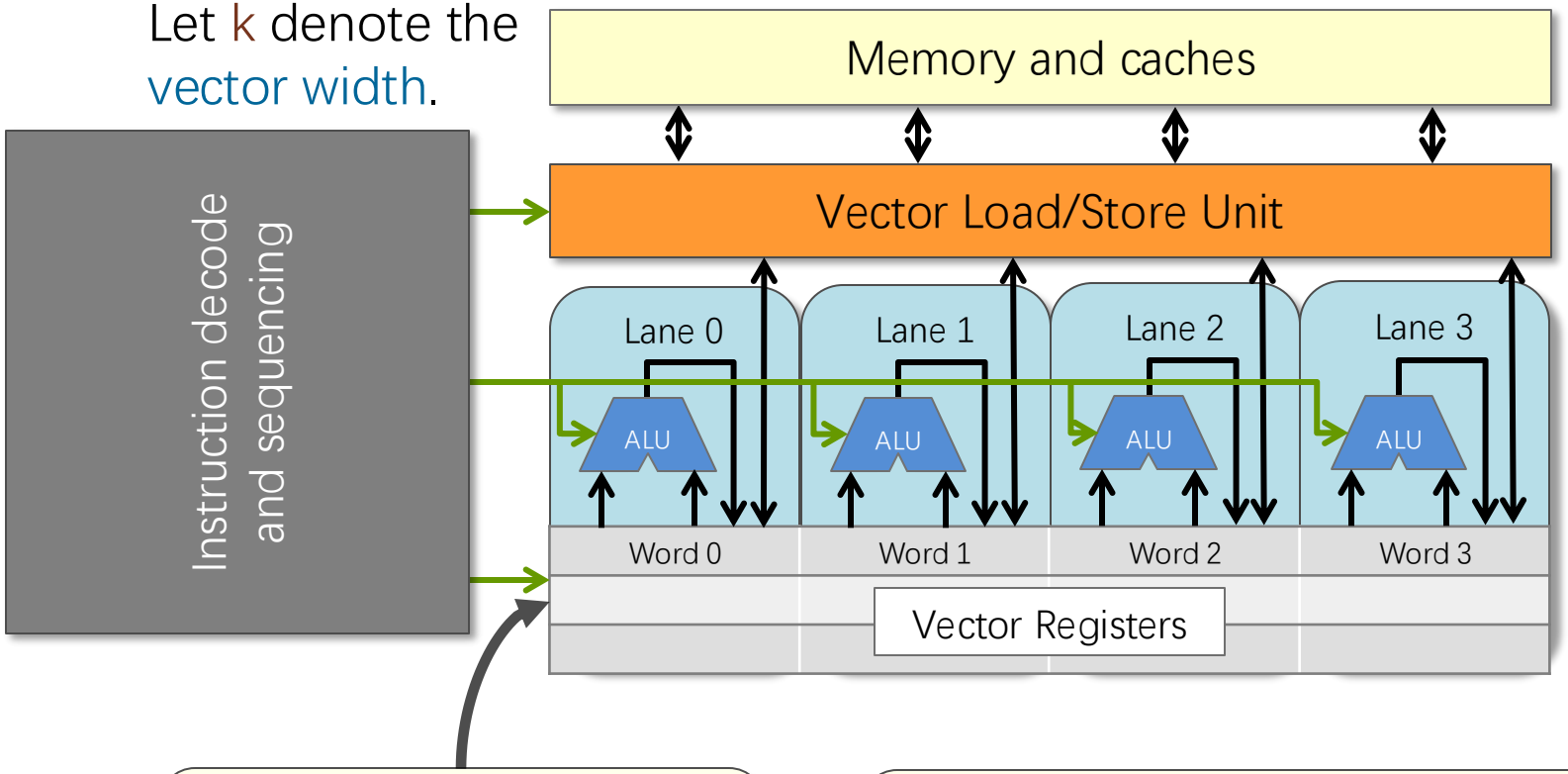
Vector Hardware

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



A k-Wide Vector Unit

Let k denote the vector width.

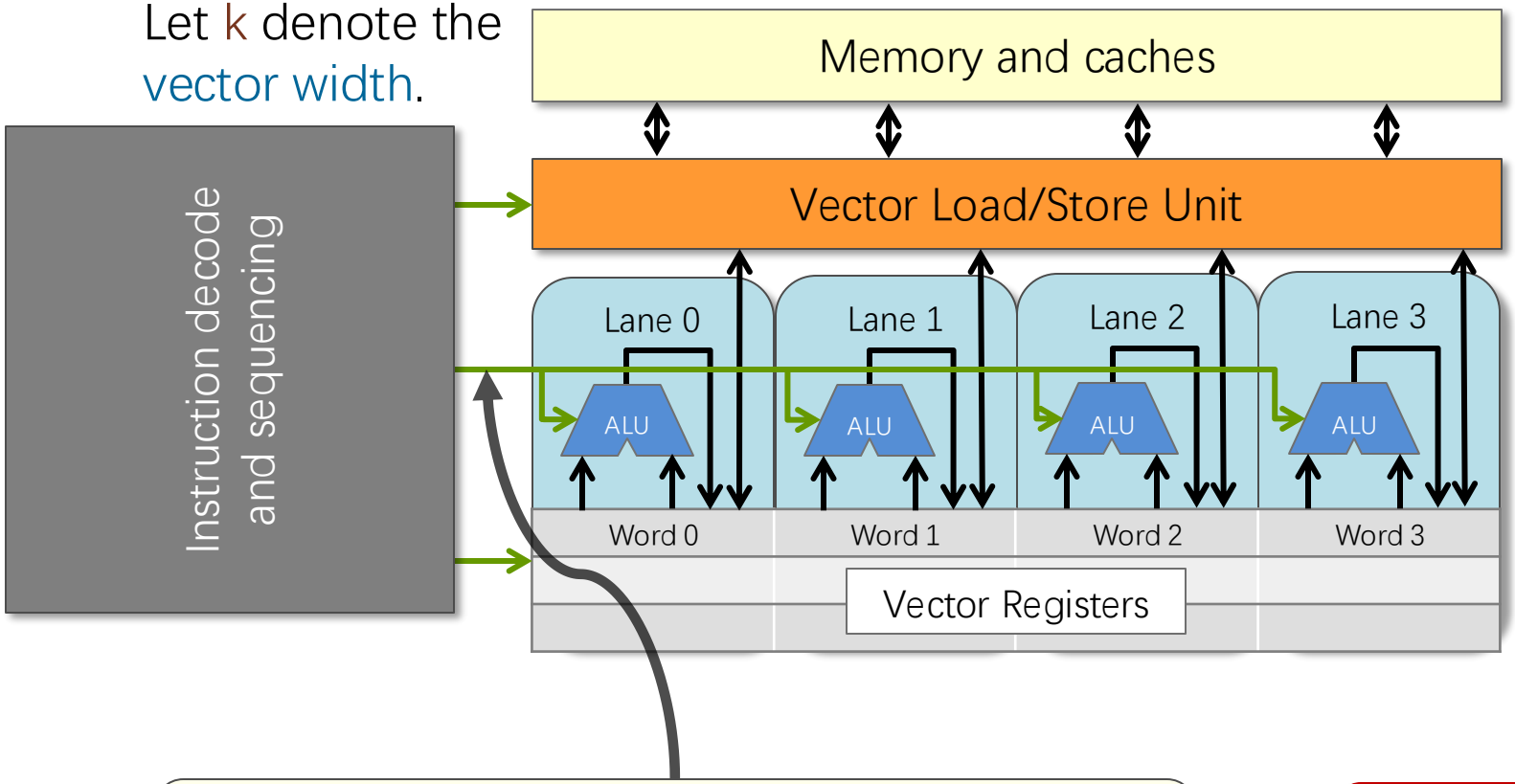


Each vector register holds k scalar integer or floating-point values

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.

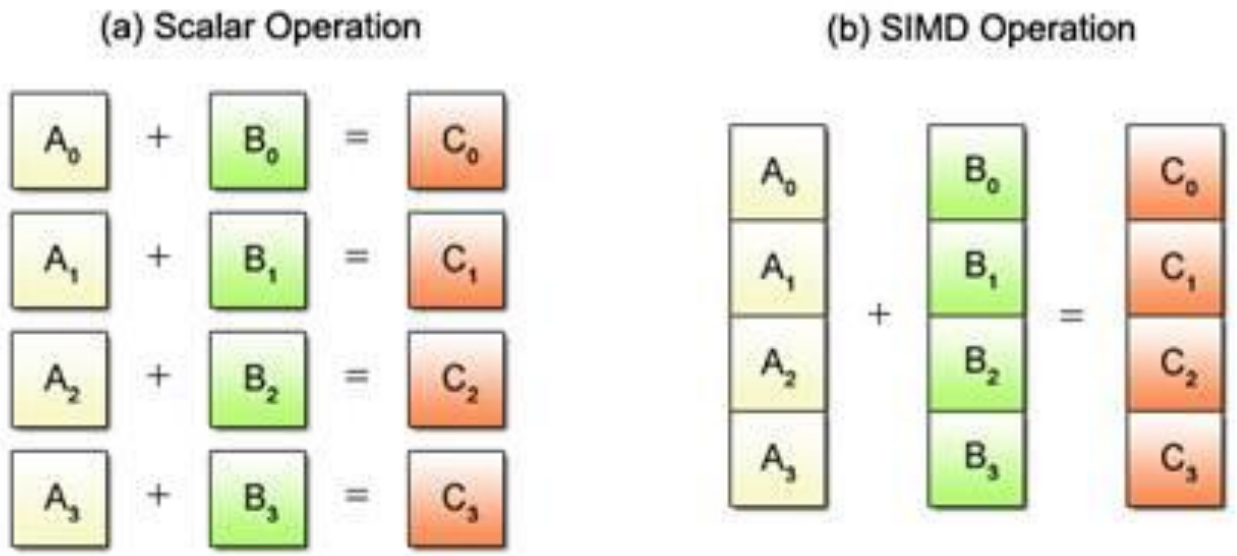


All vector lanes operate in **lock-step** and use the same instruction and control signals

Lock step!

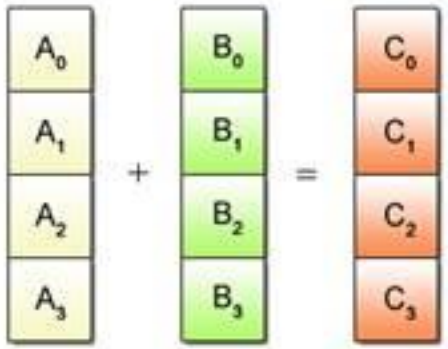
SIMD Instructions

- 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

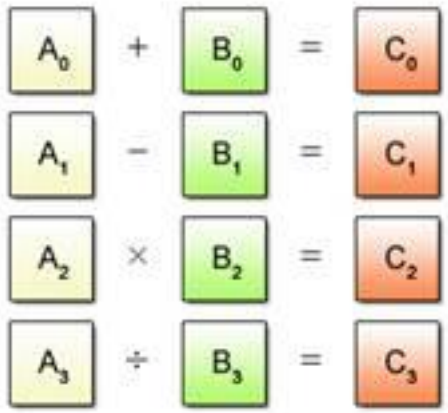


SIMD Instructions

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



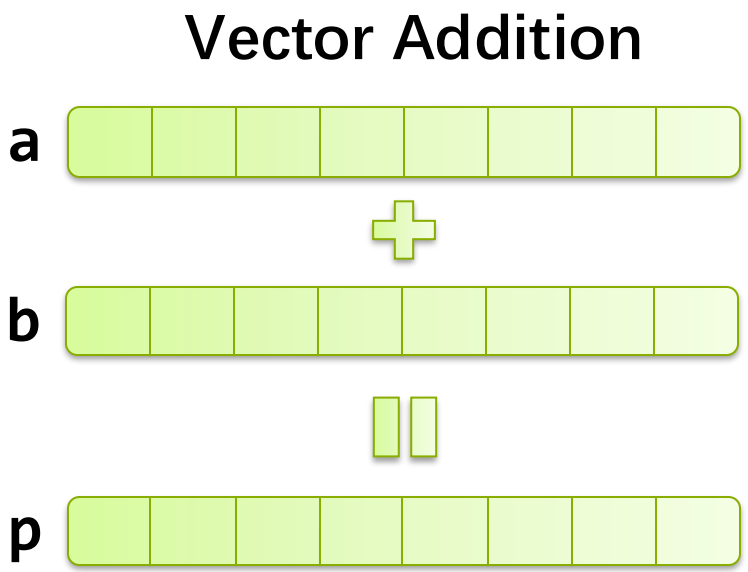
Example of SIMD Processable Patterns

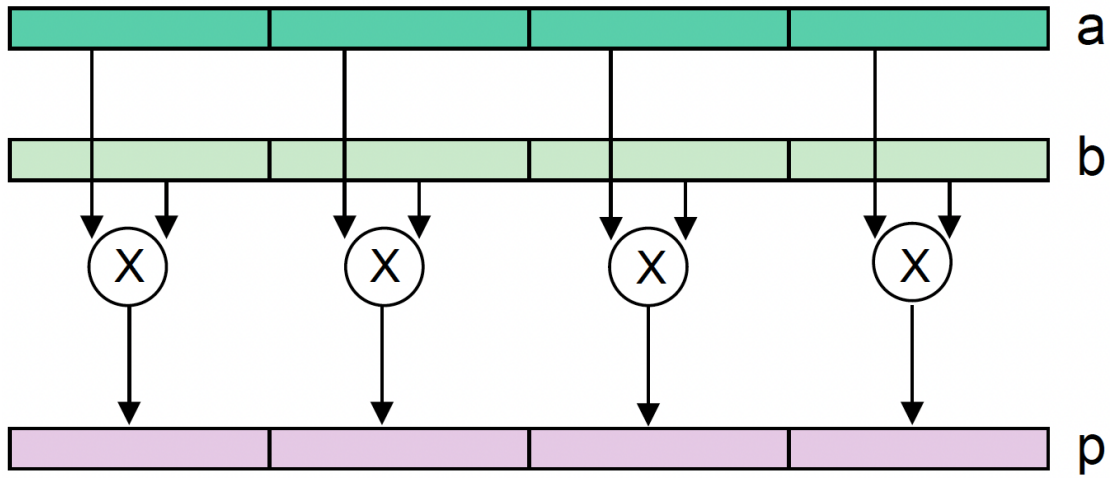


Example of SIMD Unprocesable Patterns

SIMD Instructions

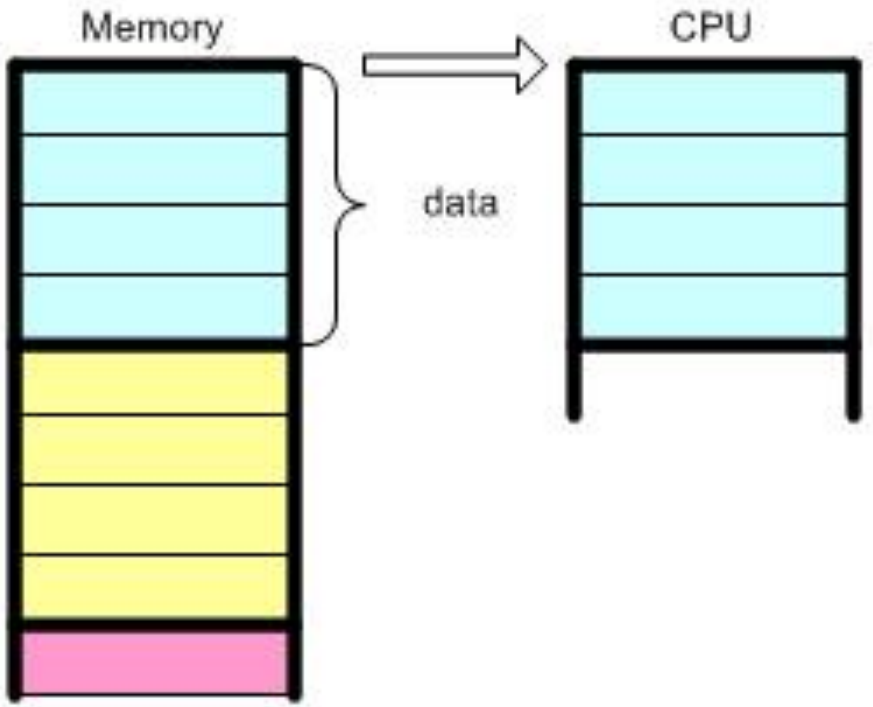
- SIMD instructions typically operate in an **elementwise** fashion
- All lanes perform exactly the same operation on each element



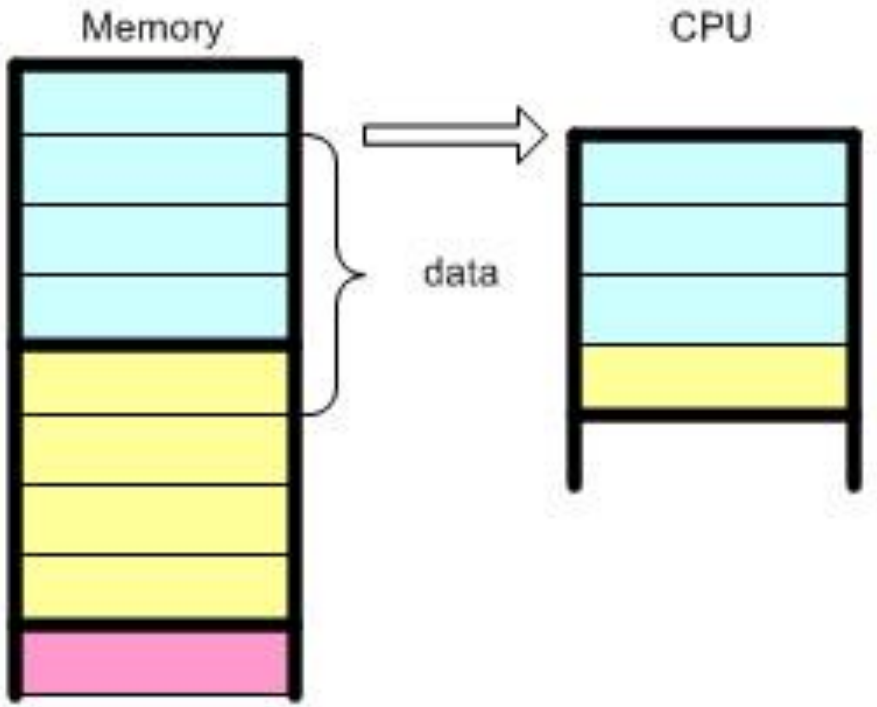


SIMD Instructions

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



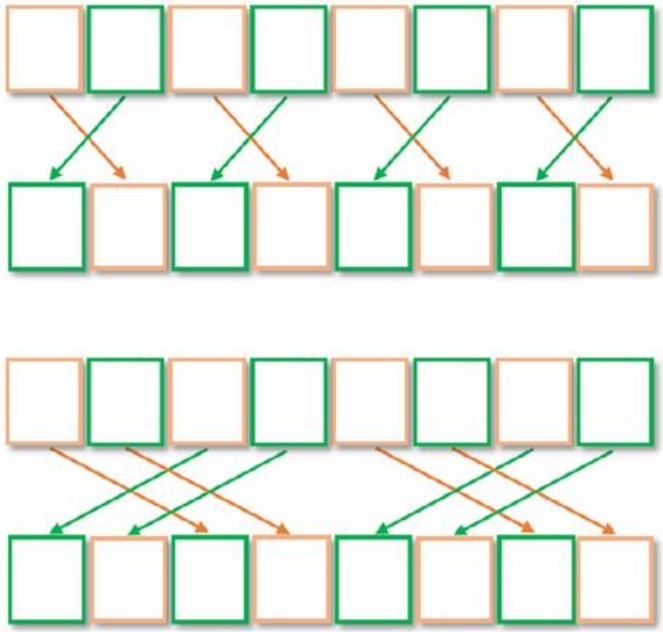
4-byte memory access for aligned data



4-byte memory access for misaligned data

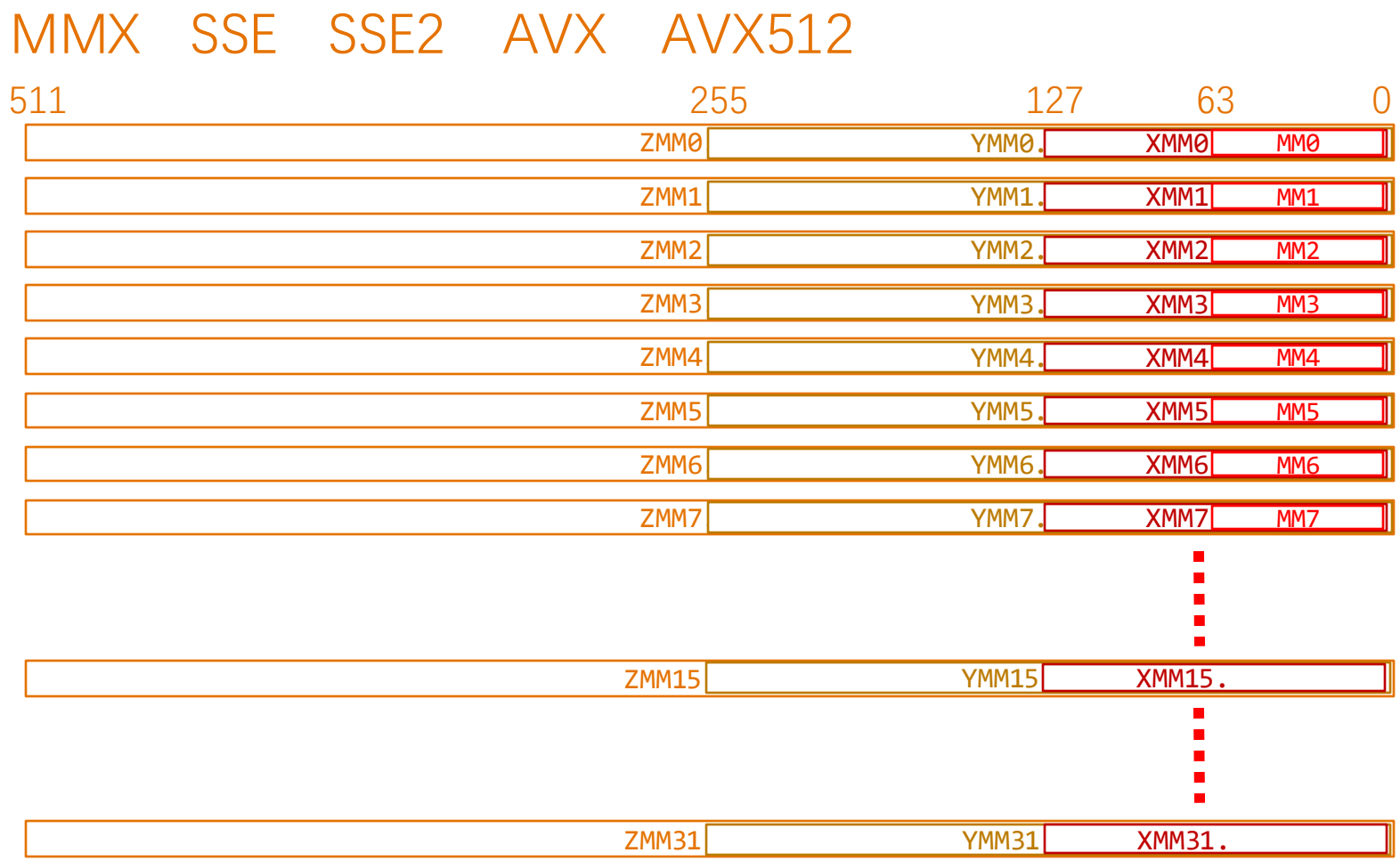
SIMD Instructions

- 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



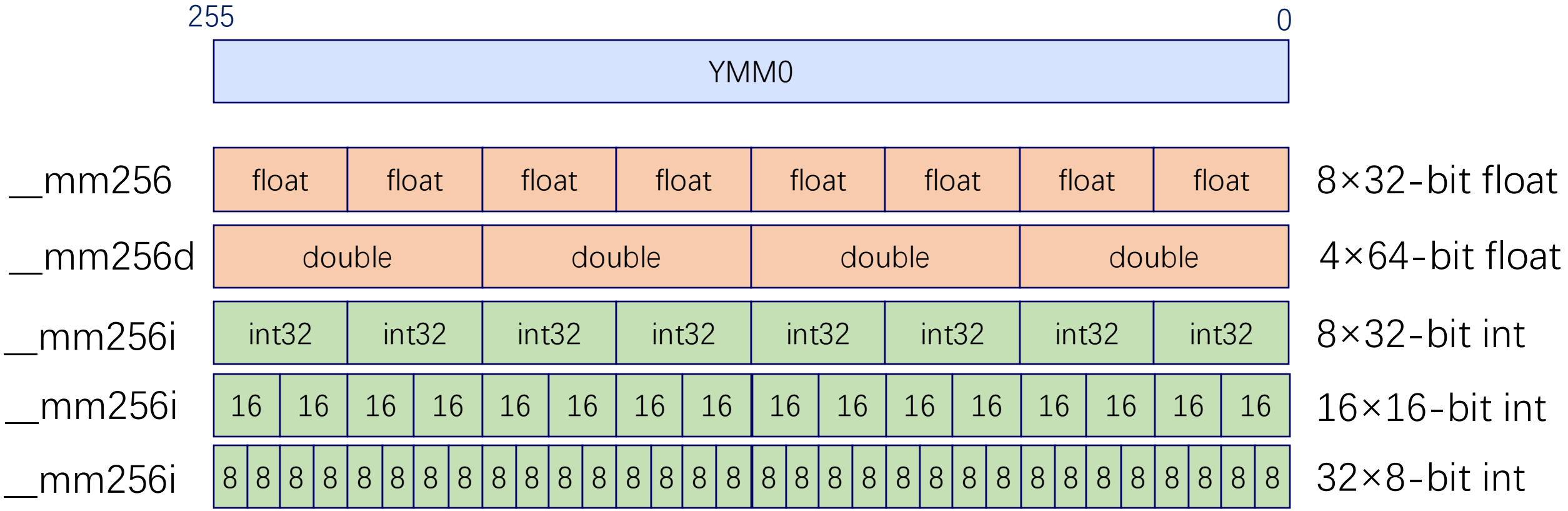
Vector Register Aliasing

- 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!



X86 SIMD INSTRUCTIONS

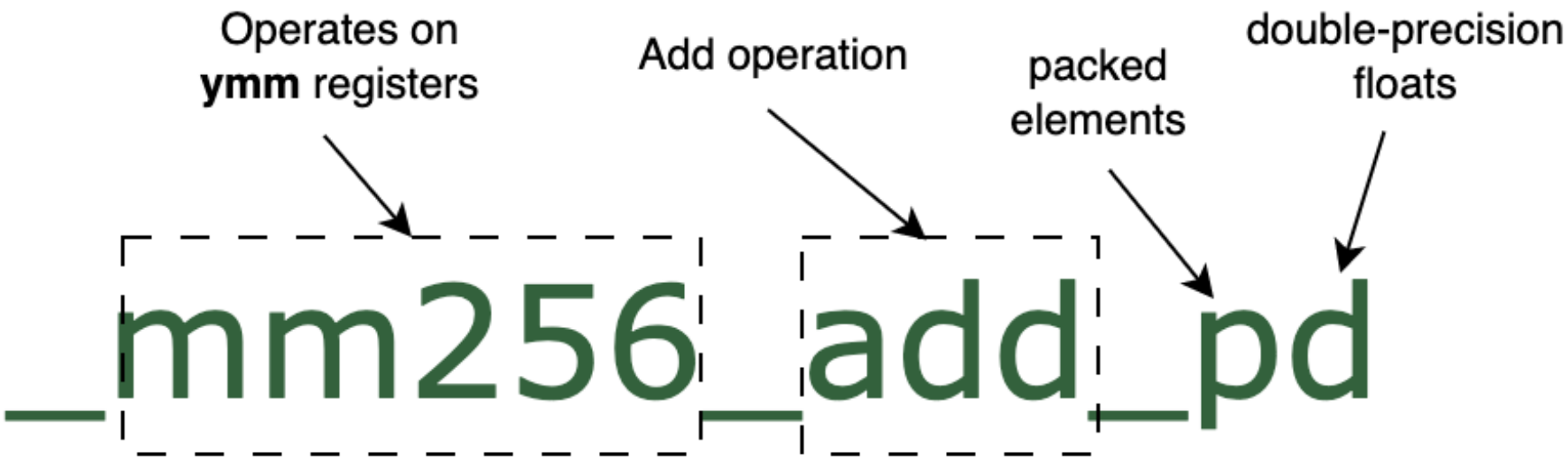
Instruction Naming Convention

- 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)

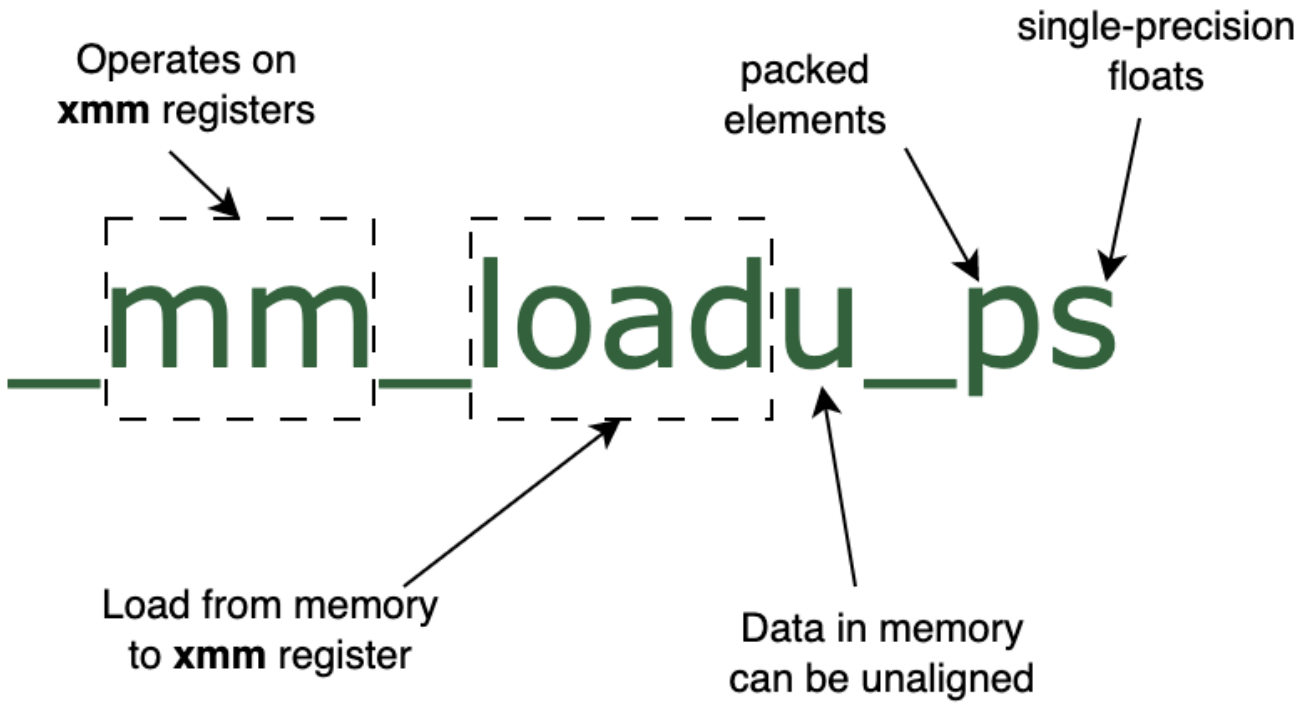
Instruction Naming Convention

- 256 bit 4-wide double precision float addition



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

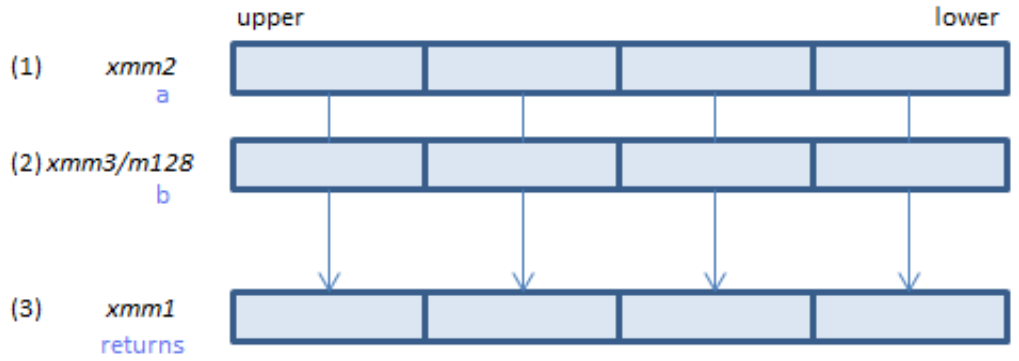
1. Arithmetic and Logic
2. Compare
3. Load/Store
4. Shuffle (Swizzle)
5. Masks
6. Non-SIMD

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
}
```

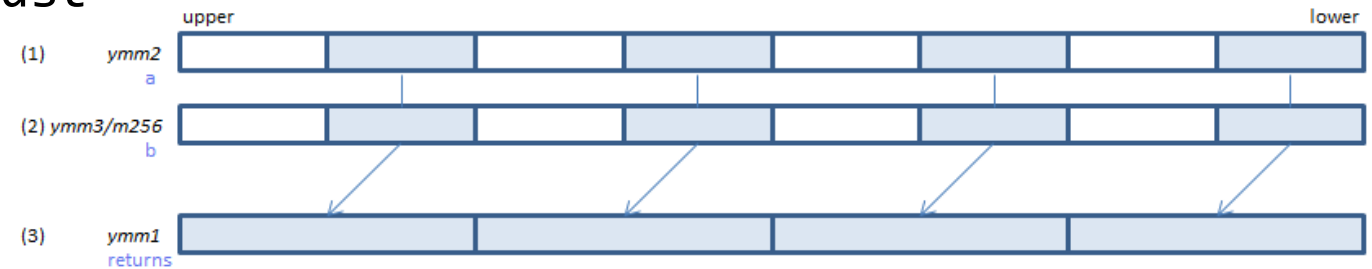


1. Arithmetic and Logic

- **Types**
  - `mul`

- **Example**

```
__m256i _mm_mul_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

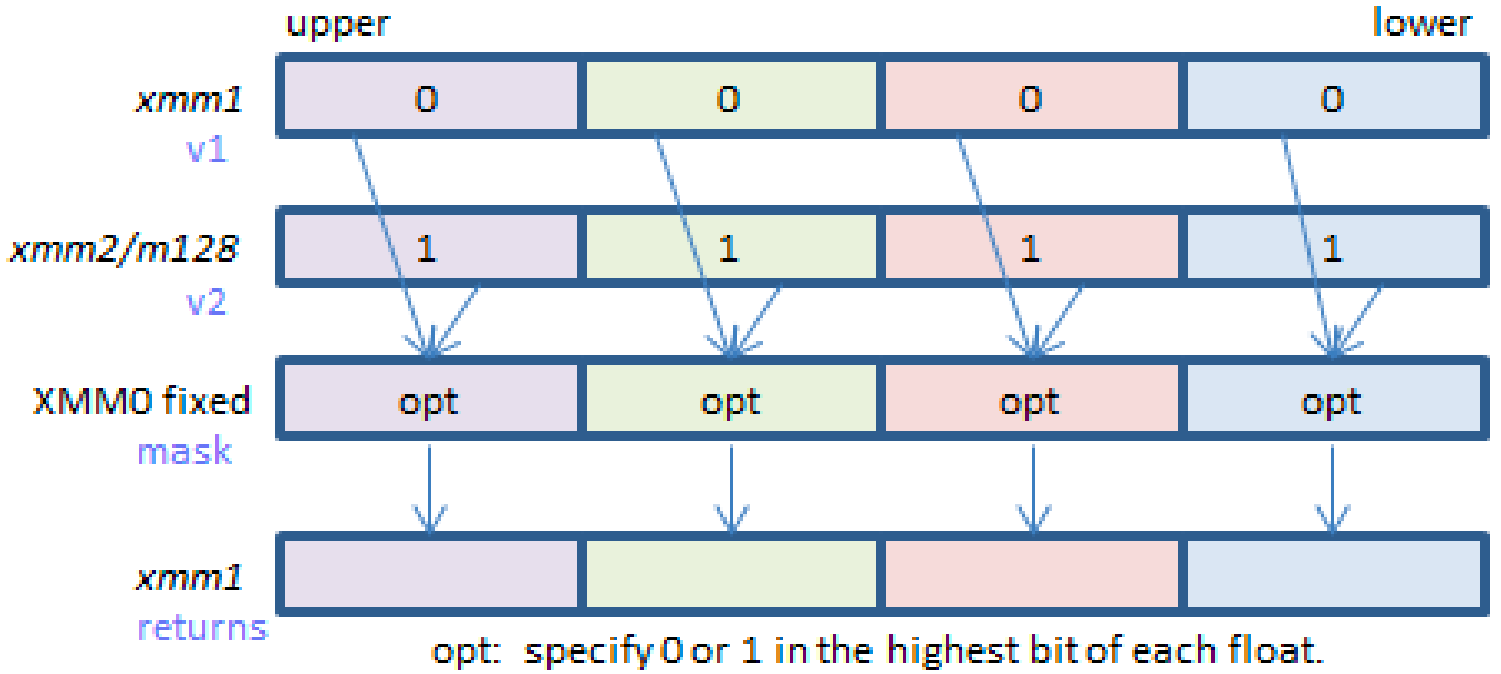
- **Example**

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

4. Shuffle: Blending

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

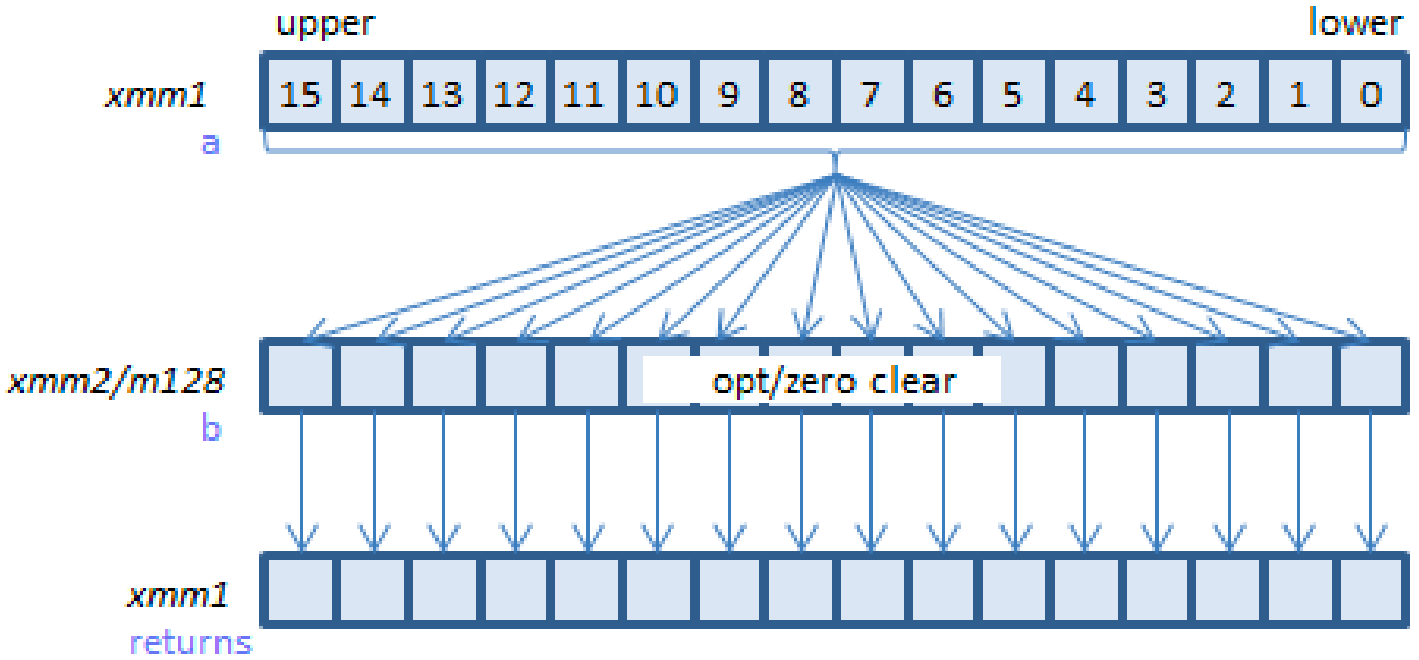
```
__m128 _mm_blendv_ps(__m128 v1, __m128 v2, __m128 mask)
```



4. Shuffle: all-to-all shuffle

- Shuffle operation: `shuffle`
- **Example**

```
__m128i _mm_shuffle_epi8(__m128i a, __m128i b)
```

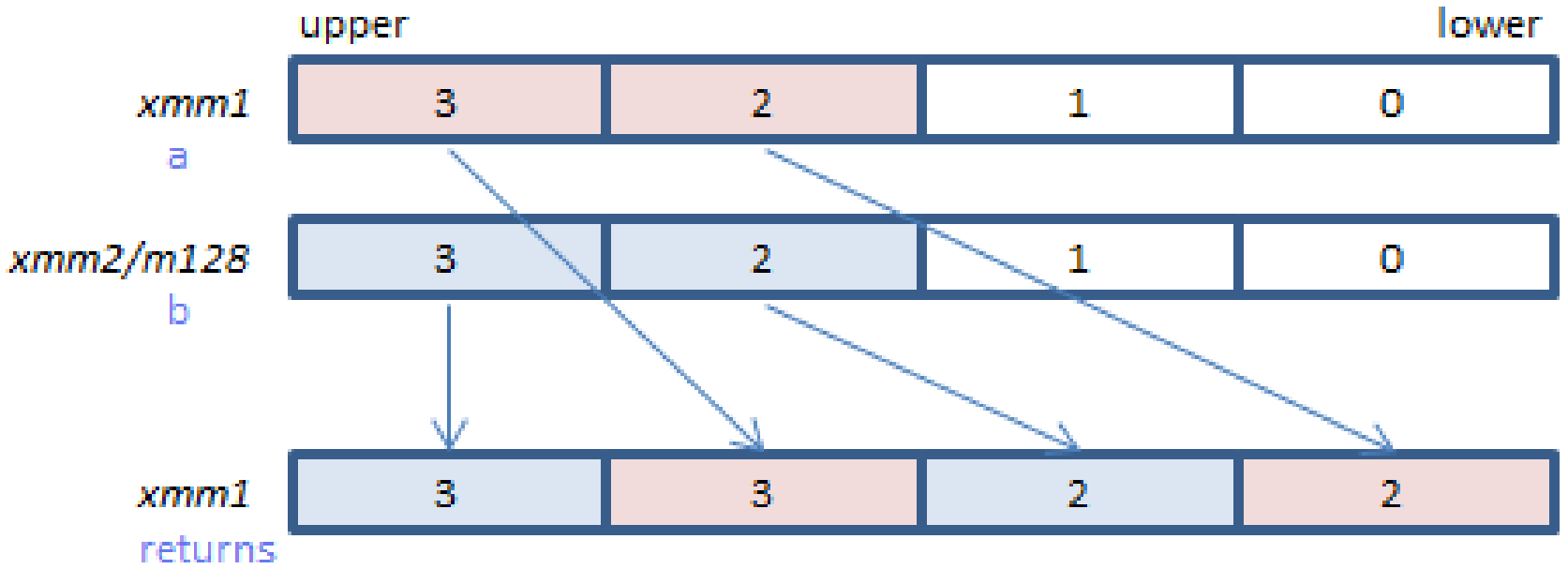


each byte of xmm2/m128:
bit 7 == 0 specifies copying, bit 3:0 specifies 0 to 15
bit 7 == 1 specifies zero clearing

4. Shuffle: unpack

- Shuffle operation: `unpackhi` & `unpacklo`
- **Example**

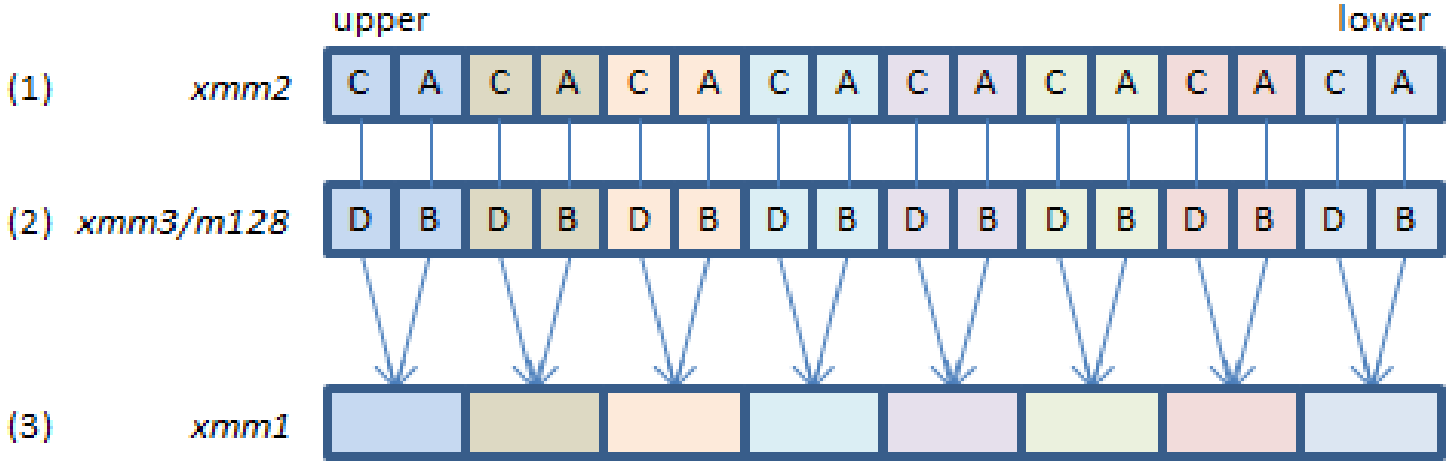
```
__m128 _mm_unpackhi_ps(__m128 a, __m128 b)
```



5. Non-SIMD: dot product instructions

- Non-SIMD operation: **madd**
- **Example**

```
__m128i _mm_maddubs_epi16(__m128i a, __m128i b)
```



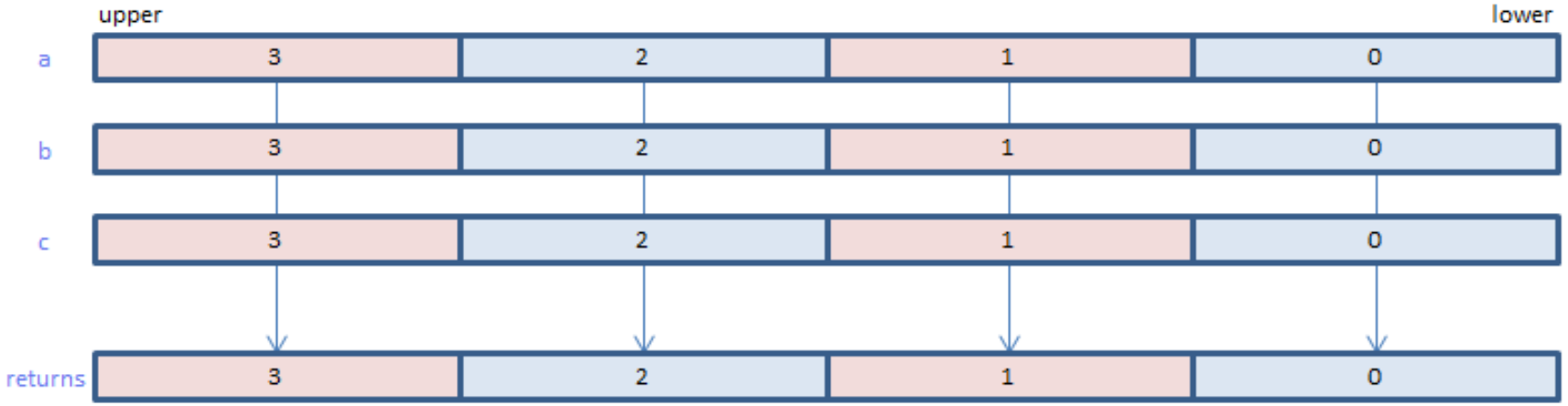
Calculate A*B+C*D and set to each WORD of (3). Signed saturation: set -32768 / 32767 on overflow.
Each BYTE of (1) is unsigned. Each BYTE of (2) is signed.

5. Non-SIMD: addsub

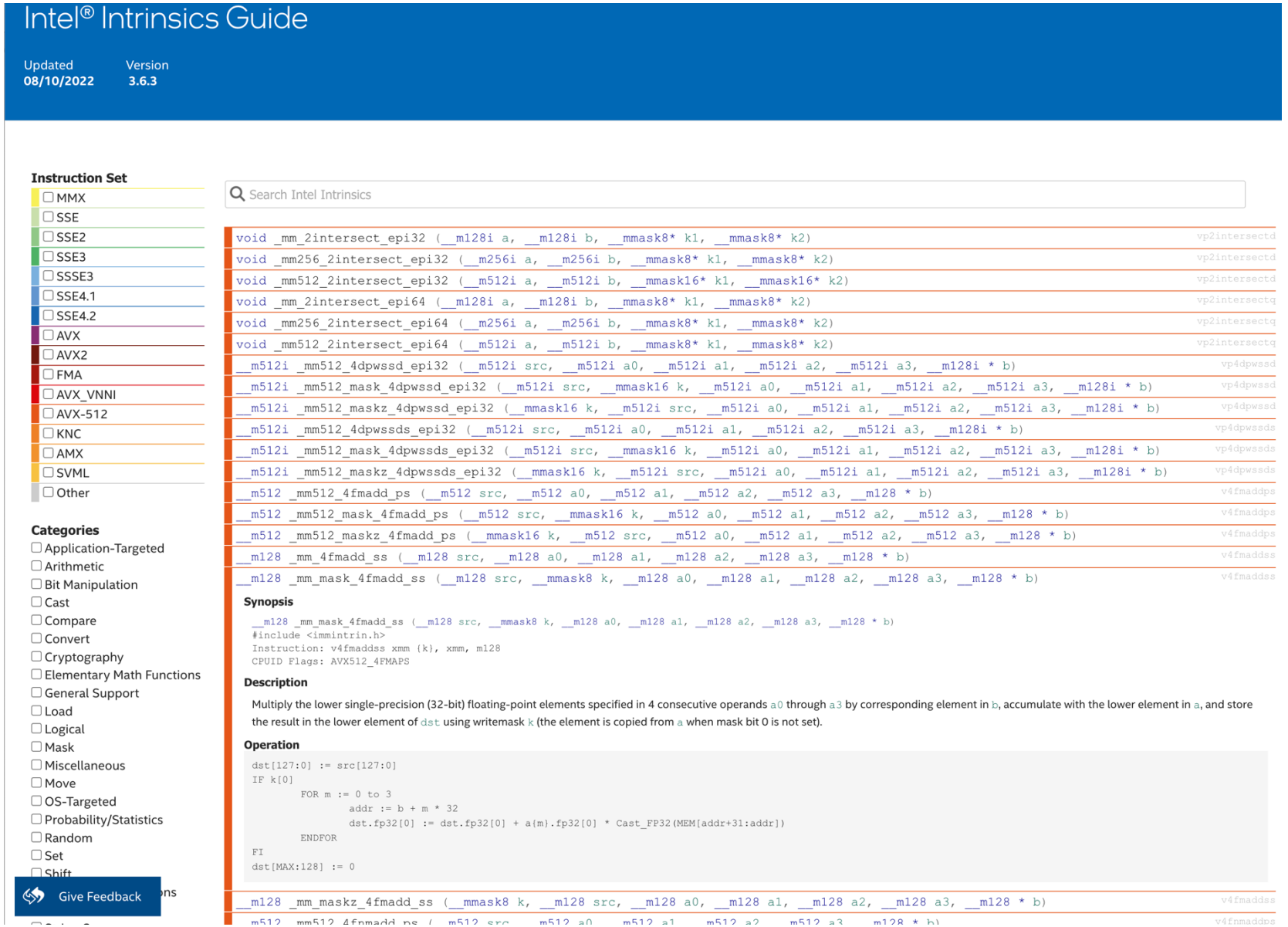
$$(a+bi)(c+di) = (ac - bd) + (ad + bc)i$$

- 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..



> 7000 Intrinsics!

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;
}
```

* 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];
  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;
}
```

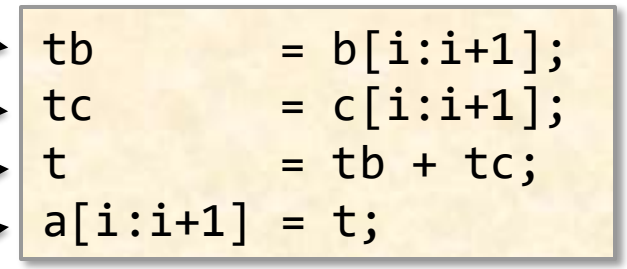
```
for (int i = 0; i < n; i += 2) {
  tb   = b[i];
  tb2  = b[i+1];
  tc   = c[i];
  tc2  = c[i+1];
  t    = tb + tc;
  t2   = tb2 + tc2;
  a[i] = t;
  a[i] = t2;
}
```

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] = t2;
}
```



Finally, pack into vector instructions

SLP Vectorization

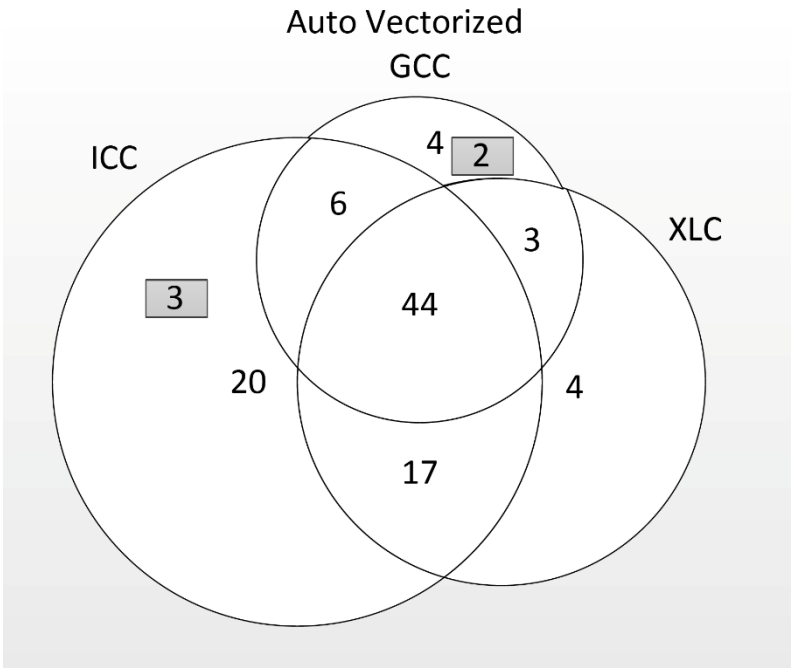
```
for (int i = 0; i < n; i++) {
  tb   = b[i];
  tc   = c[i];
  t    = tb + tc;
  a[i] = t;
}
```

```
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

How good are the Vectorizers: A Case Study

- *A matrix-vector multiplication kernel from TVM*

```
void
dot_16x1x16_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]
vpackuswb xmm4, xmm10, xmm4
vpackuswb xmm1, xmm1, xmm3
vpand xmm10, xmm0, xmm2
movzx r8d, [rdi+1]
vpand xmm3, xmm0, xmm6
vpsrlw xmm2, xmm2, 8
vmovd xmm8, r9d
vpackuswb xmm3, xmm10, xmm3
vpsrlw xmm6, xmm6, 8
vpand xmm10, xmm0, xmm4
vpackuswb xmm2, xmm2, xmm6
vpsrlw xmm4, xmm4, 8
vpand xmm6, xmm0, xmm3
vpsrlw xmm3, xmm3, 8
vpackuswb xmm10, xmm10, xmm6
vpbroadcastw xmm8, xmm8
vpackuswb xmm6, xmm4, xmm3
vpand xmm3, xmm0, xmm1
vpand xmm0, xmm0, xmm2
vpackuswb xmm3, xmm3, xmm0
vpsrldq xmm0, xmm10, 8
vpmovsxbw xmm4, xmm10
vmovd xmm7, r8d
vpmullw xmm4, xmm4, xmm8
vpmovsxbw xmm0, xmm0
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
vpackuswb xmm1, xmm1, xmm2
vmovd xmm5, eax
vpmullw xmm3, xmm3, xmm7
vpmovsxbw xmm7, xmm6
vpbroadcastw xmm5, xmm5
vpmullw xmm7, xmm7, xmm9
vpsrldq xmm2, xmm6, 8
```

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
```

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 dependences
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://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

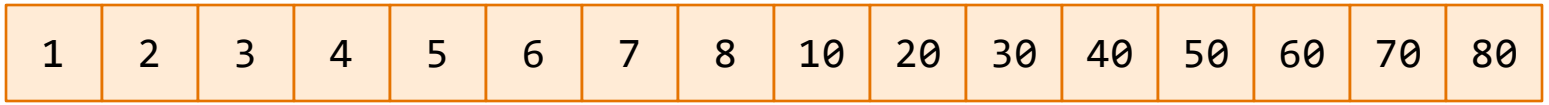
```
bar:                                    # @bar
        cmpl    %edx, %esi
        jge     .LBB0_8
        movslq  %esi, %rsi
        movslq  %edx, %rax
        movq    %rax, %rcx
        subq    %rsi, %rcx
        cmpq    $8, %rcx
        jae     .LBB0_3
        movq    %rsi, %r8
        jmp     .LBB0_14
.LBB0_3:
        cmpq    $16, %rcx
        jae     .LBB0_9
        xorl    %edx, %edx
        jmp     .LBB0_5
.LBB0_9:
        movq    %rcx, %rdx
        andq    $-16, %rdx
        vbroadcastss    %xmm0, %zmm1
        leaq    (%rdi,%rsi,4), %r8
        xorl    %r9d, %r9d
.LBB0_10:                               # =>This Inner Loop Header:
        vmovups %zmm1, (%r8,%r9,4)
        addq    $16, %r9
        cmpq    %r9, %rdx
        jne     .LBB0_10
        cmpq    %rdx, %rcx
        je      .LBB0_8
        testb   $8, %cl
        jne     .LBB0_5
        addq    %rsi, %rdx
        movq    %rdx, %r8
        jmp     .LBB0_14
.LBB0_5:
        movq    %rcx, %r9
        andq    $-8, %r9
        leaq    (%r9,%rsi), %r8
        vbroadcastss    %xmm0, %ymm1
        leaq    (%rdi,%rsi,4), %rsi
.LBB0_6:                                # =>This Inner Loop Header:
        vmovups %ymm1, (%rsi,%rdx,4)
        addq    $8, %rdx
        cmpq    %rdx, %r9
        jne     .LBB0_6
        cmpq    %r9, %rcx
        je      .LBB0_8
.LBB0_14:                               # =>This Inner Loop Header:
        vmovss  %xmm0, (%rdi,%r8,4)
        incq    %r8
        cmpq    %r8, %rax
        jne     .LBB0_14
.LBB0_8:
        vzeroupper
        retq
```

```
bar:                                    # @bar
        testl   %esi, %esi
        jle     .LBB0_13
        movl    %esi, %eax
        cmpl    $8, %esi
        jae     .LBB0_3
        xorl    %ecx, %ecx
        jmp     .LBB0_12
.LBB0_3:
        cmpl    $16, %esi
        jae     .LBB0_5
        xorl    %ecx, %ecx
        jmp     .LBB0_9
.LBB0_5:
        movl    %eax, %ecx
        andl    $-16, %ecx
        vbroadcastss    %xmm0, %zmm1
        leaq    (,%rax,4), %rdx
        andq    $-64, %rdx
        xorl    %esi, %esi
.LBB0_6:                                # =>This Inner Loop Header: Depth=1
        vmovups %zmm1, (%rdi,%rsi)
        addq    $64, %rsi
        cmpq    %rsi, %rdx
        jne     .LBB0_6
        cmpq    %rax, %rcx
        je      .LBB0_13
        testb   $8, %al
        je      .LBB0_12
.LBB0_9:
        movq    %rcx, %rdx
        movl    %eax, %ecx
        andl    $-8, %ecx
        vbroadcastss    %xmm0, %ymm1
.LBB0_10:                               # =>This Inner Loop Header: Depth=1
        vmovups %ymm1, (%rdi,%rdx,4)
        addq    $8, %rdx
        cmpq    %rdx, %rcx
        jne     .LBB0_10
        cmpq    %rax, %rcx
        je      .LBB0_13
.LBB0_12:                               # =>This Inner Loop Header: Depth=1
        vmovss  %xmm0, (%rdi,%rcx,4)
        incq    %rcx
        cmpq    %rcx, %rax
        jne     .LBB0_12
.LBB0_13:
        vzeroupper
        retq
```

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

2. Cannot Prove Independence

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



2. Cannot Prove Independence

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

Serial execution: in scalar

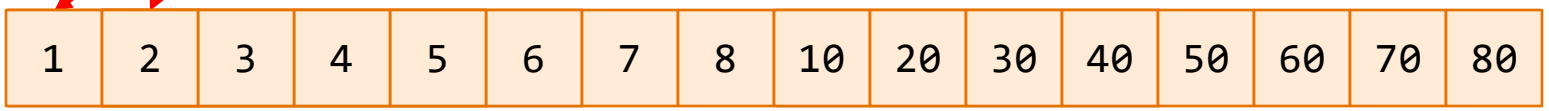
2. Cannot Prove Independence

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

Parallel execution: in vector of width 2

2. Cannot Prove Independence

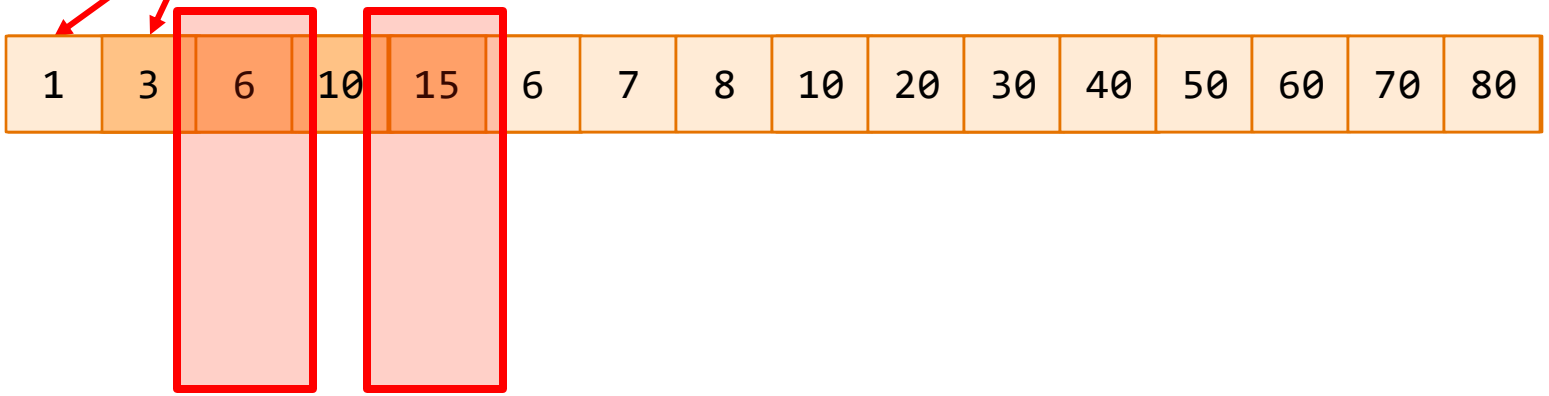
```
void bar(float * A, float * B) {
  for (int i = 0; i < 4; ++i)
    A[i] += B[i];
}
```



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 ≠ Scalar Results
- Memory overlap → 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];
}
```

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

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

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

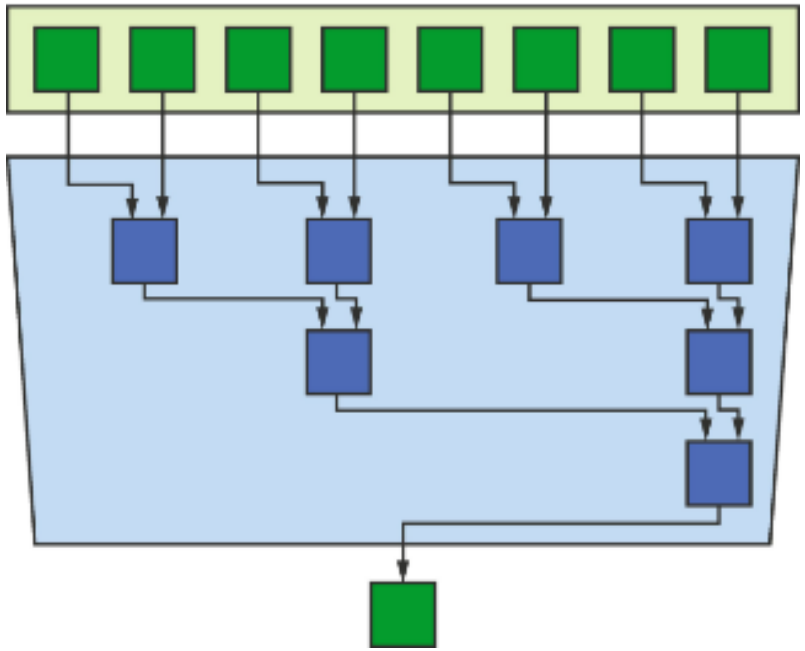
A[i] += A[i-100];

```
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:                                    # @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
```

Compute mask A[i] > 0 ?

Update value only if mask set

add w/o condition

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:                                    # @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:                                # =>This Inner Loop Header: Depth=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
```

K

A[i]

A[i] + K

A[i] * K

Using masks to update values

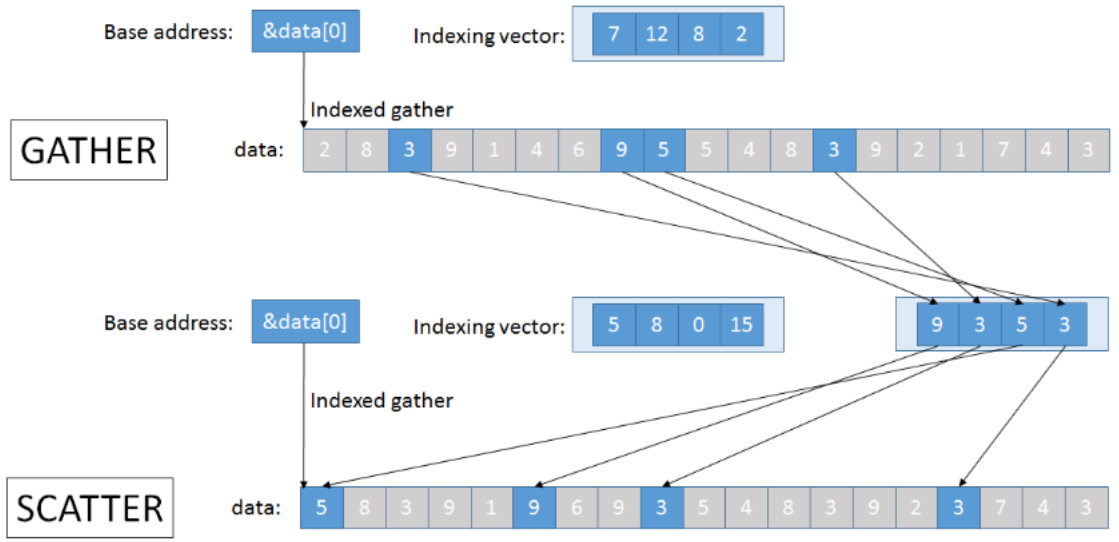
6. Non-contiguous data

```
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 = 0x40
        jne     .LBB0_1
        vzeroupper
        retq
```

gather from memory

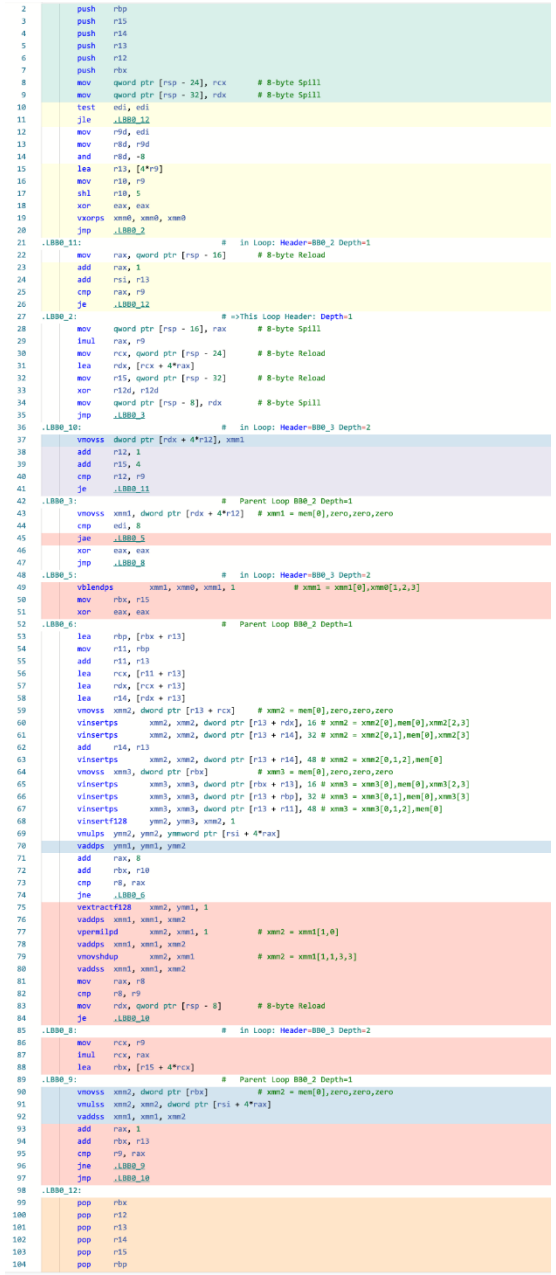
7. Cost Model Failures

- 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][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];
}
```

7. Cost Model Failures

```
matmul:                                 # @matmul
        push    r15
        push    r14
        push    rbx
        test    edi, edi
        jle     .LBB0_7
        mov     eax, edi
        lea     r15, [4*rax]
        xor     r8d, r8d
.LBB0_2:                                # =>This Loop Header: Depth=1
        mov     r9, r8
        imul    r9, rax
        mov     r10, rdx
        xor     r11d, r11d
.LBB0_3:                                #   Parent Loop BB0_2 Depth=1
        lea     r14, [r9 + r11]
        vmovss  xmm0, dword ptr [rcx + 4*r14]   # xmm0 = mem[0],zero,zero,zero
        mov     rbx, r10
        xor     edi, edi
.LBB0_4:                                #   Parent Loop BB0_2 Depth=1
        vmovss  xmm1, dword ptr [rbx]           # xmm1 = mem[0],zero,zero,zero
        vmulss  xmm1, xmm1, dword ptr [rsi + 4*rdi]
        vaddss  xmm0, xmm0, xmm1
        add     rdi, 1
        add     rbx, r15
        cmp     rax, rdi
        jne     .LBB0_4
        vmovss  dword ptr [rcx + 4*r14], xmm0
        add     r11, 1
        add     r10, 4
        cmp     r11, rax
        jne     .LBB0_3
        add     r8, 1
        add     rsi, r15
        cmp     r8, rax
        jne     .LBB0_2
.LBB0_7:
        pop     rbx
        pop     r14
        pop     r15
        ret
```



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

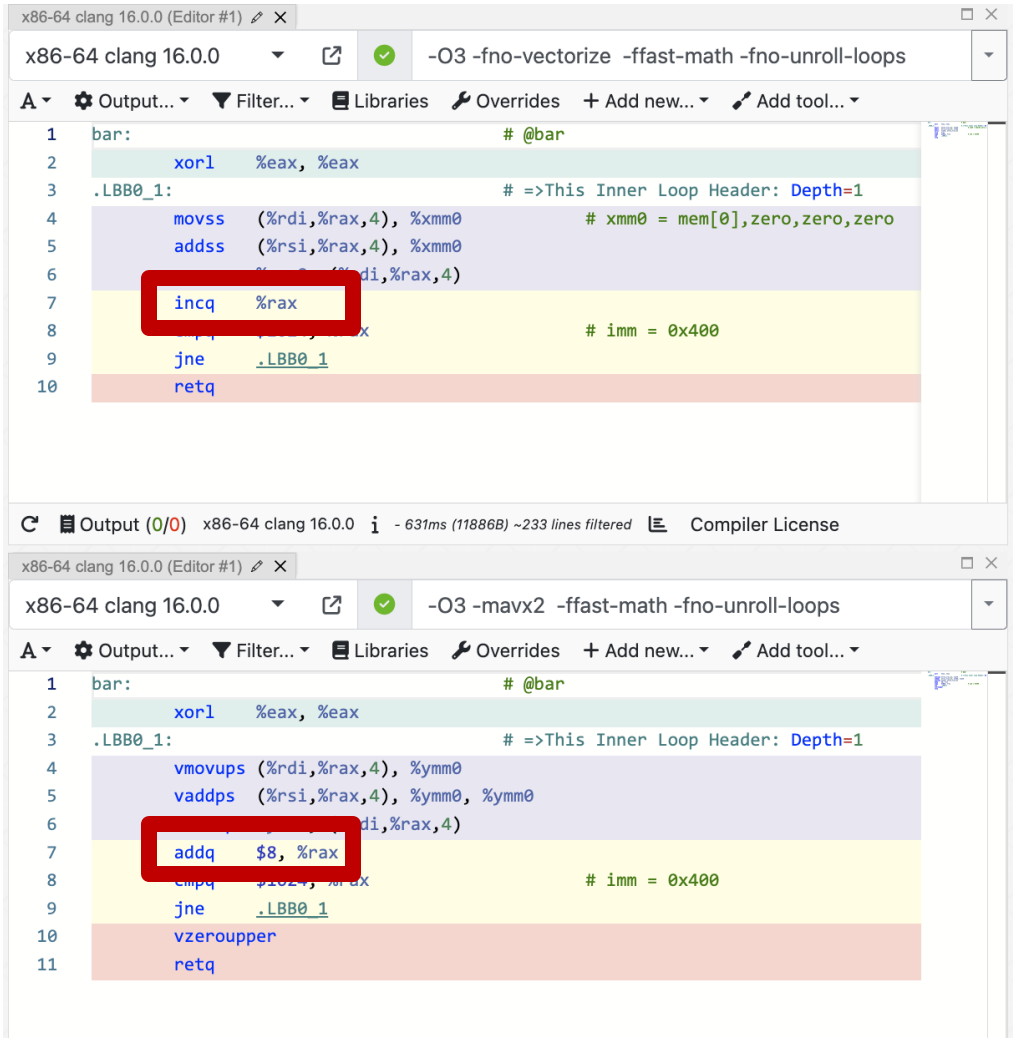
```
bar:                                    # @bar
        pushq   %r15
        pushq   %r14
        pushq   %rbx
        movq    %rsi, %rbx
        movq    %rdi, %r14
        xorl    %r15d, %r15d
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
        vmovups (%r14,%r15,4), %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
```

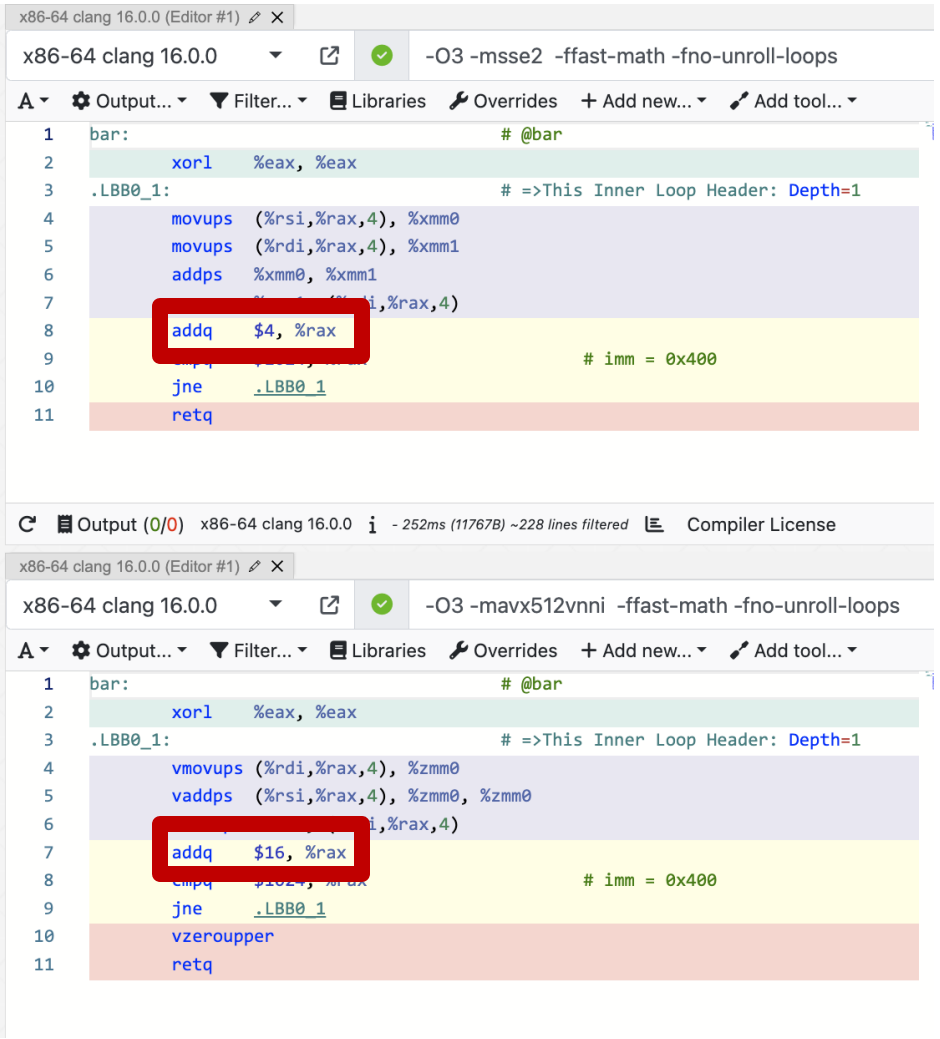
9. Different Hardware Versions

- 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





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

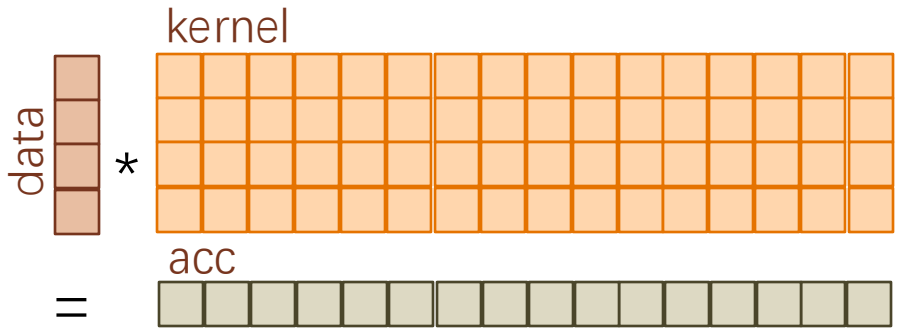
```
void foo(float *restrict x, float *restrict y) {
   for (int i = 0; i < 8; i++) {
       if (x[i] < y[i])
         x[i] += y[i];
       else
         x[i] -= y[i];
   }
}
```

```
__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

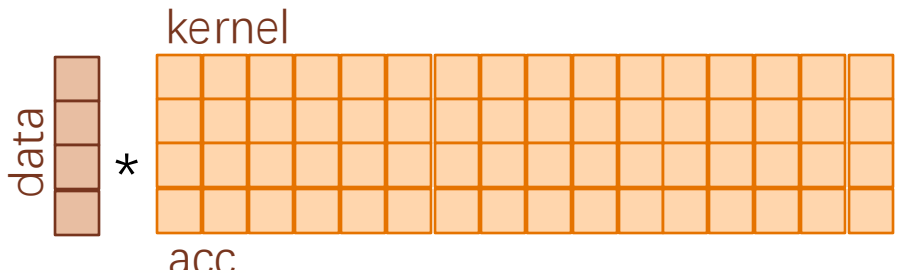


2.2x faster than scalar

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



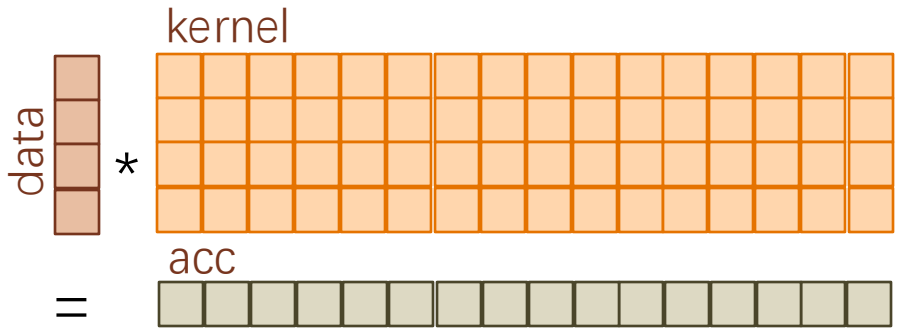

```
dot:                                    # @dot
        vpmovzxbd       (%rdi), %xmm0           # xmm0 = mem[0],zero,zero,zero,mem[1],zero,zero,zero,mem[2],zero,zero,zero,mem[3],zero,zero,zero
        xorl    %eax, %eax
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
        vmovd   (%rdx,%rax,4), %xmm1            # xmm1 = mem[0],zero,zero,zero
        vpmovsxbd       (%rsi,%rax,4), %xmm2
        vpmaddwd        %xmm0, %xmm2, %xmm2
        vpaddd  %xmm1, %xmm2, %xmm1
        vpshufd $238, %xmm1, %xmm2              # xmm2 = xmm1[2,3,2,3]
        vpaddd  %xmm2, %xmm1, %xmm1
        vpshufd $85, %xmm1, %xmm2               # xmm2 = xmm1[1,1,1,1]
        vpaddd  %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?**

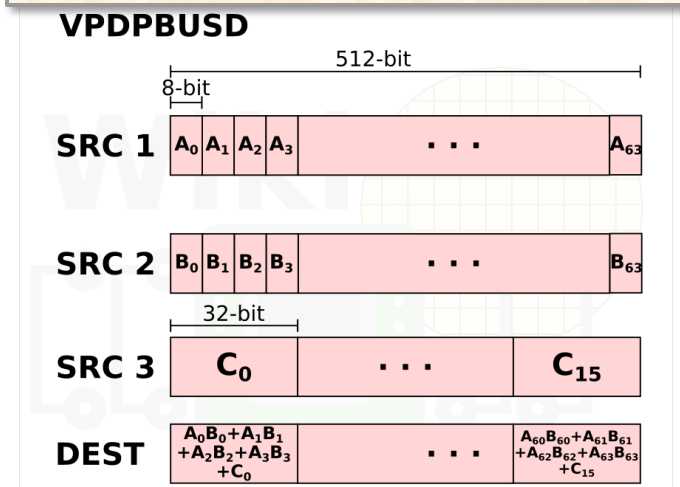


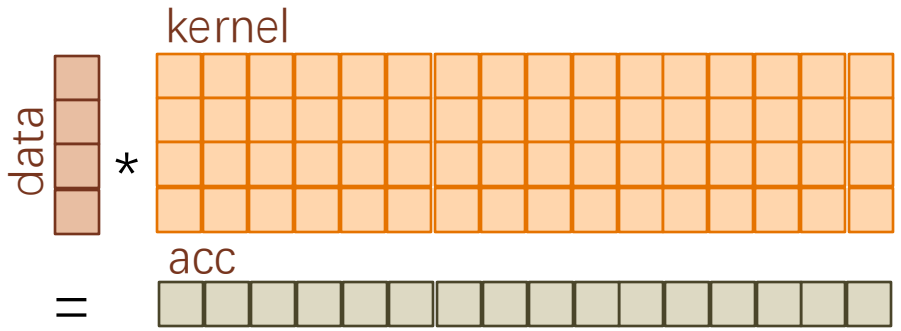
2.2x faster than scalar

```
__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];
}
```





2.2x faster than scalar

```
__m512i acc_vec = _mm512_loadu_epi32(acc);
// {data0, data1, data2, data3, …, data0, d
__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