Performance Engineering of Software Systems

LECTURE 6

Vectorization

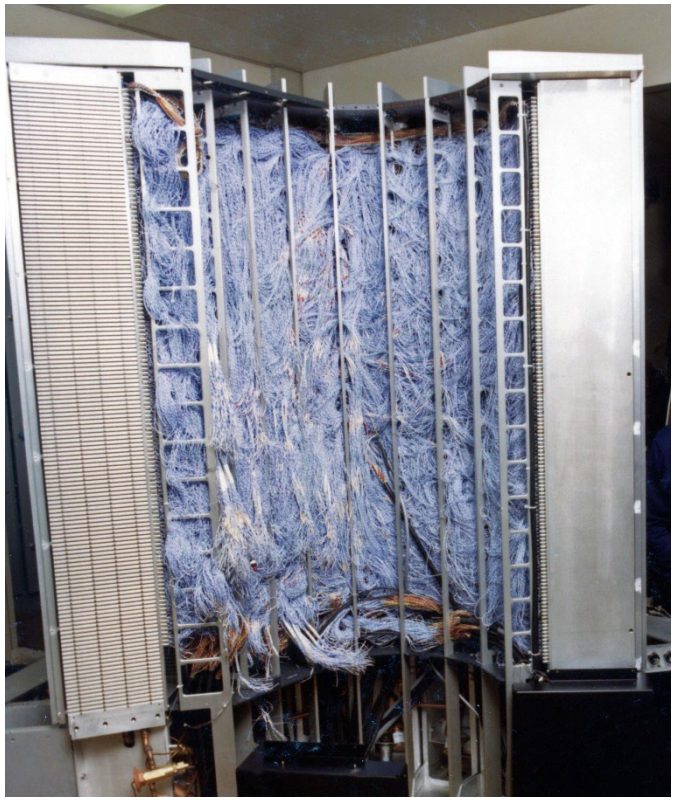
Saman Amarasinghe*

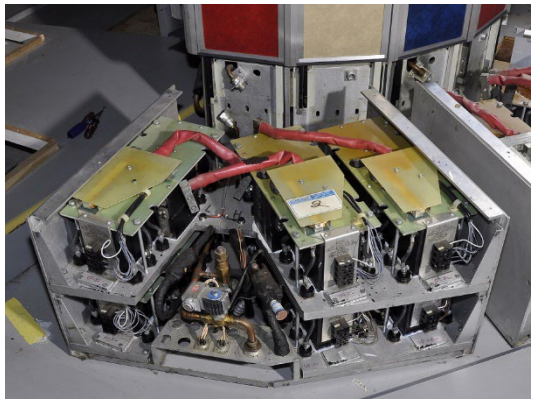
September 27, 2022

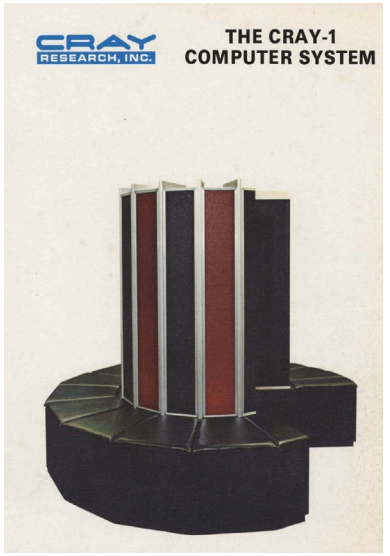
*With a lot of help from Yishen "Tom" Chen

Vectorization

The First Supercomputers were developed in the 70's





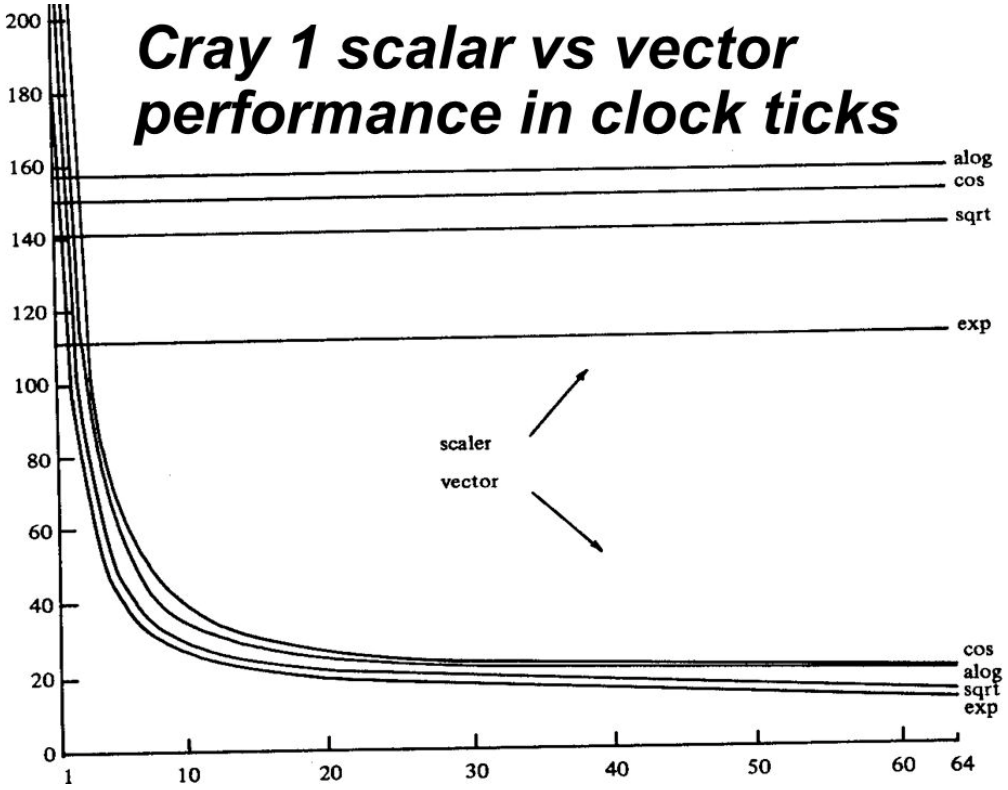


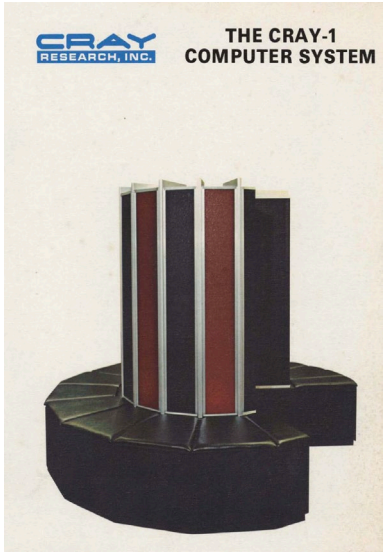
Seymour Cray

200,000 gates, 80 MHz, $10M

Vectorization

The First Supercomputers were developed in the 70's





Seymour Cray

Required long vectors for performance

Multimedia Instructions

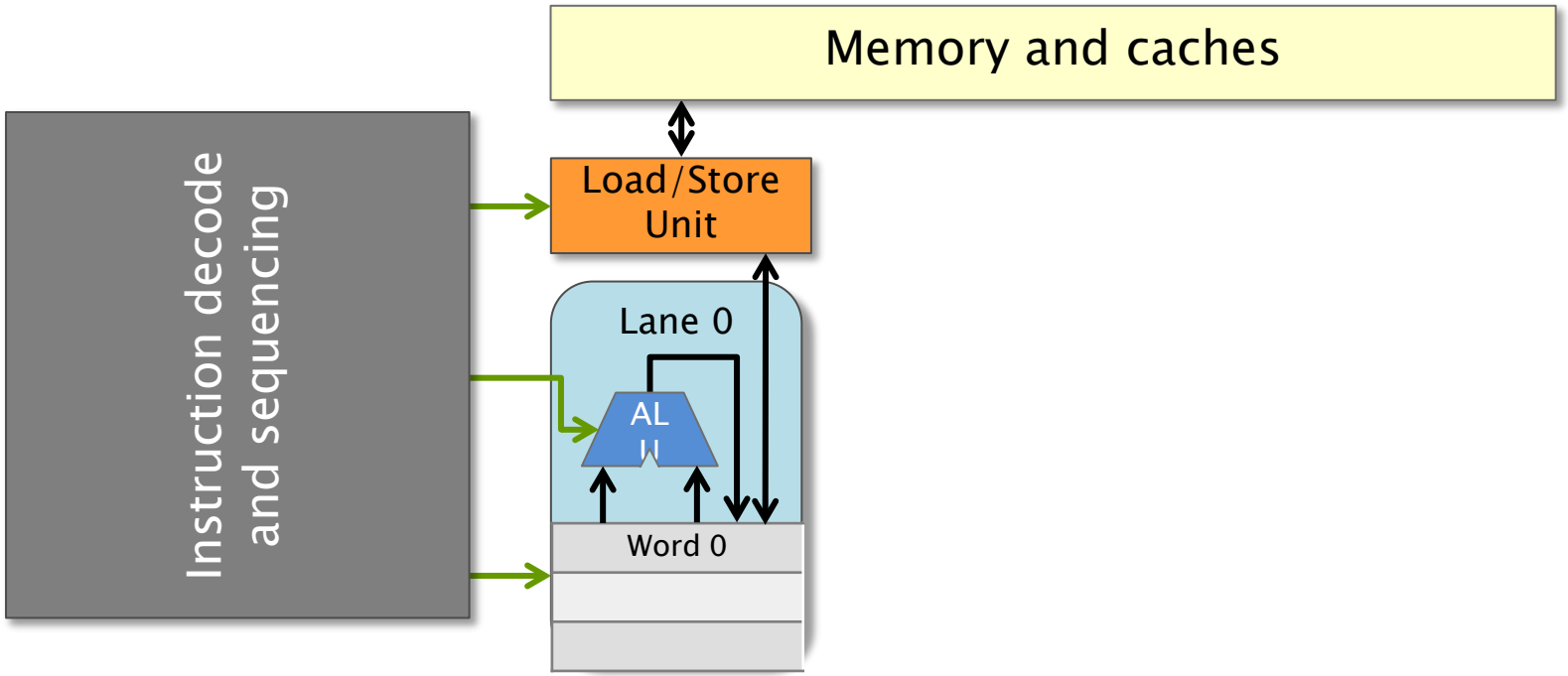
- Multimedia ISA extensions came in the 90's
- Multimedia instructions are
  - SIMD: Single Instruction Multiple Data
  - Integrated into the processor
  - Short and fast
  - No startup overhead
- Different register set
  - The multi-media register set
  - Extension of the original floating-point registers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

Vendor	ISA	Year	Bitwidth
Intel	MMX	1996	64 bits
AMD	3DNow!	1998	64 bits
Intel	SSE	1999	128 bits
Intel	SSE2	2001	128 bits
Intel	SSE3	2006	128 bits
Intel	AVX	2008	256 bits
ARM	Neon	2011	128 bits
Intel	AVX2	1013	256 bits
Intel	AVX512	2015	512 bits
Intel	AVX512-VNNI	2021	512 bits

What is SIMD?

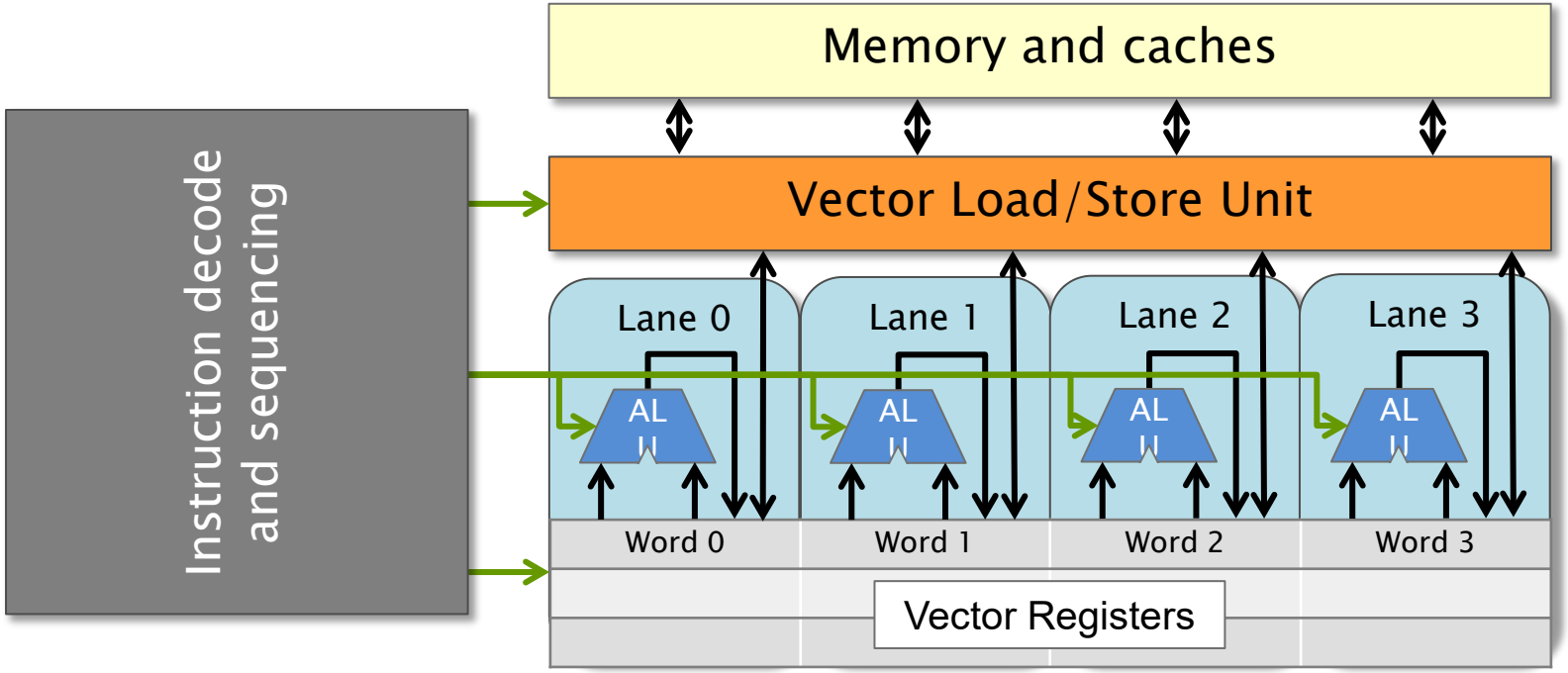
Scalar Hardware

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



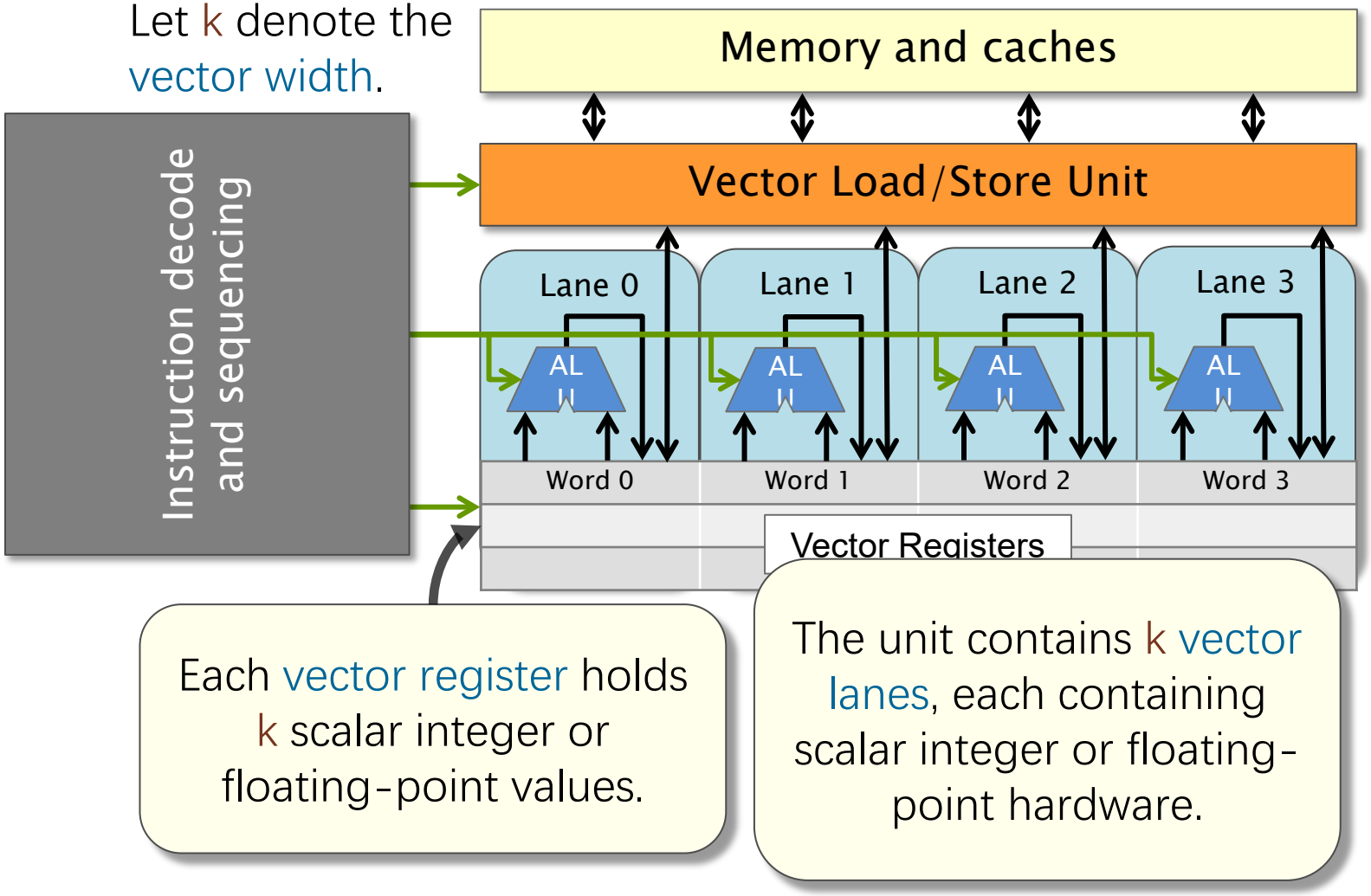
Vector Hardware

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



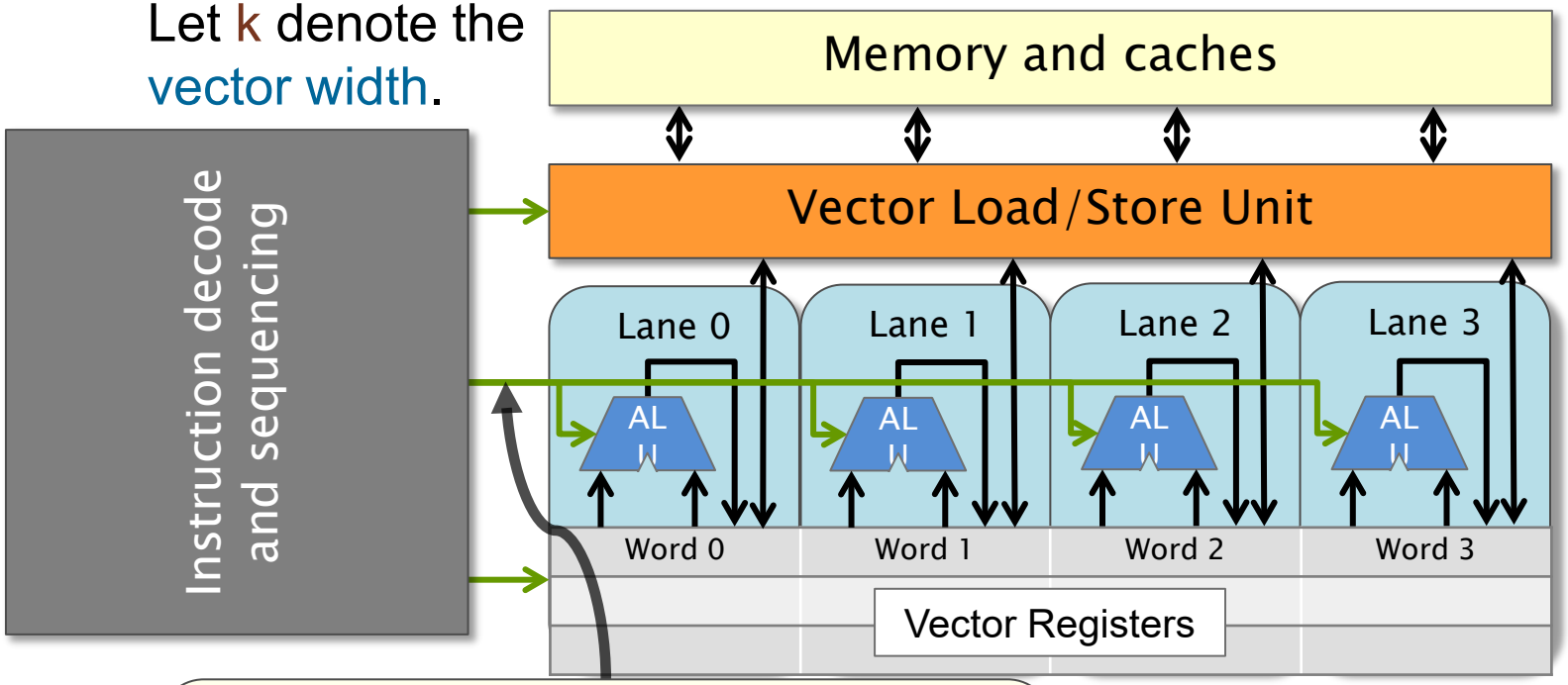
A k-Wide Vector Unit

Let k denote the vector width.



A k-Wide Vector Unit

Let k denote the vector width.



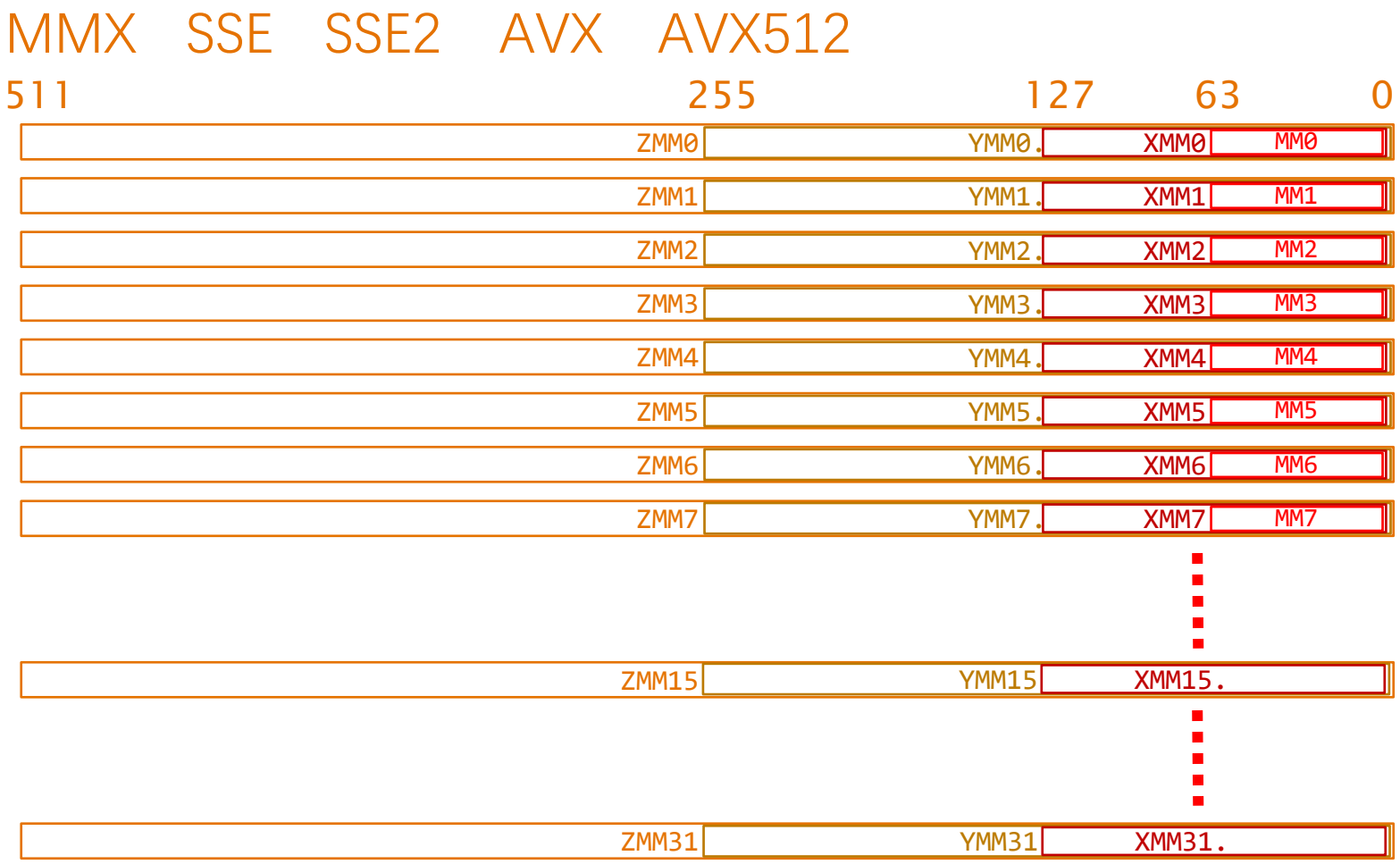
All vector lanes operate in lock-step and use the same instruction and control signals.

SIMD Instructions

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

Vector Register Aliasing

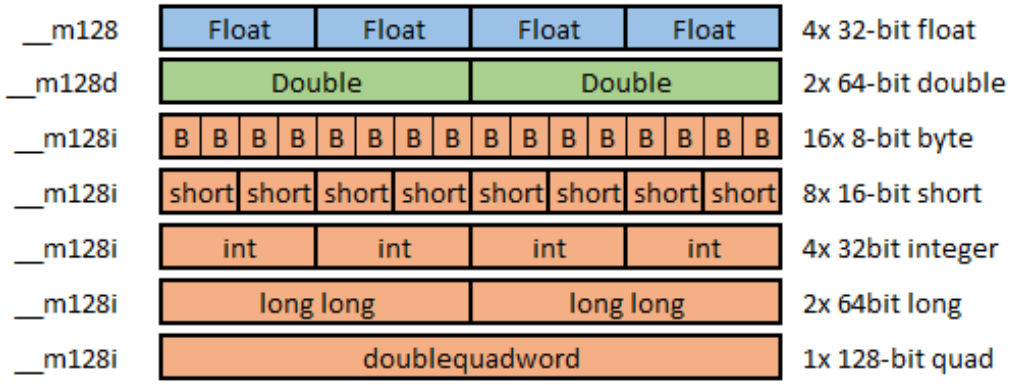
Different names, but same registers



Vector Register Aliasing

Each register can have different data types

SSE Data Types (16 XMM Registers)

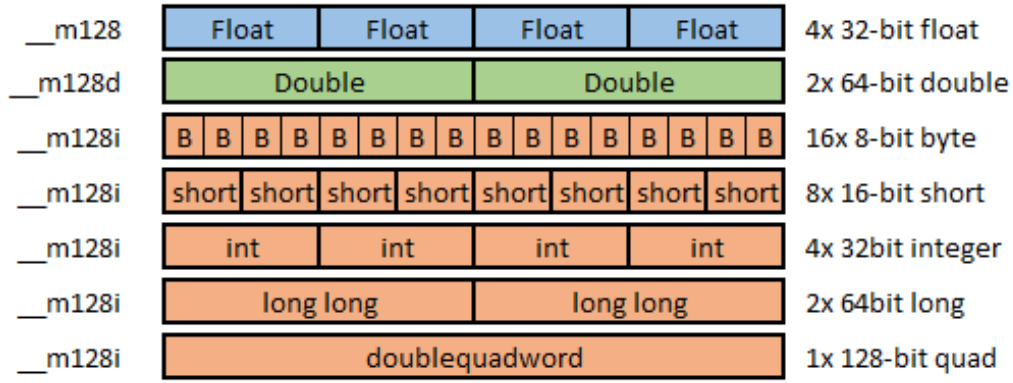


https://www.codingame.com/playgrounds/283/sse-avx-vectorization/what-is-sse-and-avx

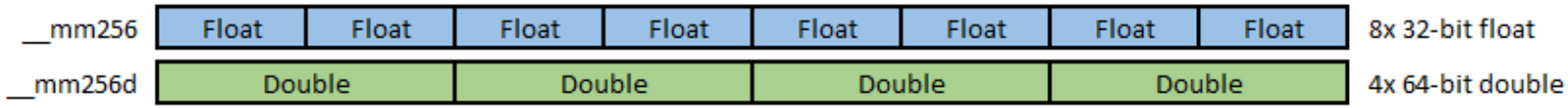
Vector Register Aliasing

Each register can have different data types

SSE Data Types (16 XMM Registers)



AVX Data Types (16 YMM Registers)



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

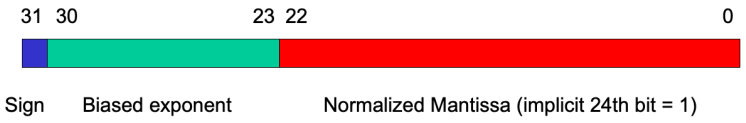
AVX-512 has ZMM registers of 512 bits…

Floating Point

Numbers, Numbers,...Numbers

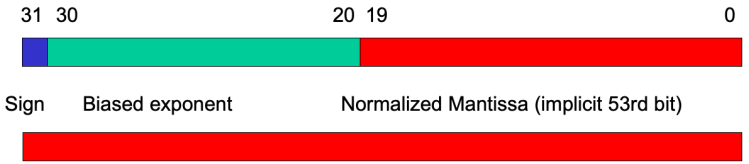
	bits	Smallest value	Biggest value
Short	16	±1	±32,768
Int	32	±1	±2.147...x 10^{9}
Long long	64	±1	±9.223.... x 10^{18}
IEEE Single Precision	**64**	±1.2 × 10^{-38}	±3.4 × 10^{+38}
IEEE Double Precision	128	±2.2 × 10^{-308}	±1.8 × 10^{+308}
Minfloats	**8**	±0.125	±15,360

- **IEEE 754 single precision**



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

- **IEEE 754 double precision**



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

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

Issues with Floating Point

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

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

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

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

- Compilers are conservative unless told otherwise
- Use the fast-math flag
- In gcc `-ffast-math` means disregarding a lot of rules

```
-fno-math-errno,           -funsafe-math-optimizations,
-ffinite-math-only,        -fno-rounding-math,
-fno-signaling-nans,       -fcx-limited-range,
-fexcess-precision=fast, -fno-signed-zeros,
-fno-trapping-math,       -fassociative-math,
-freciprocal-math
```

X86 Multi-Media Instructions

Instruction Naming Convention

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)
- Examples
  - 8-wide float addition: `_mm256_add_ps`
  - 16-wide absolute value of 16-bit ints: `_mm256_abs_epi16`
  - 4-wide float subtraction: `_mm_sub_ps` (128 is omitted)

Categories of Vector Instructions

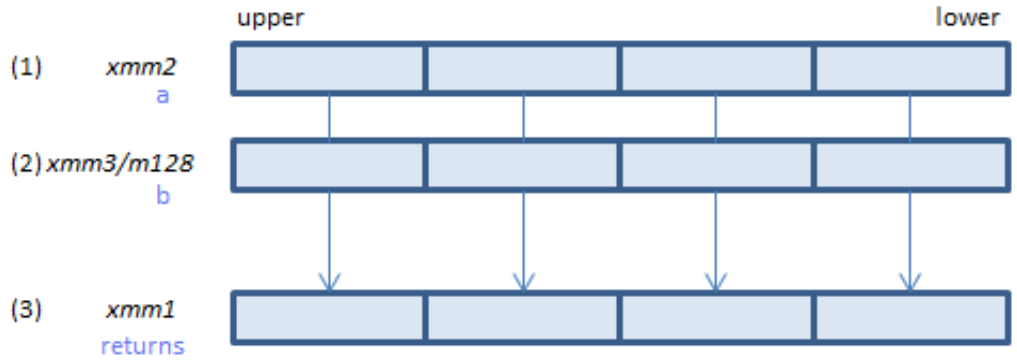
1. Arithmetic and Logic
2. Compare
3. Load/Store
4. Shuffle (Swizzle)
5. Masks
6. Non-SIMD

1. Arithmetic and Logic

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

★ Example

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

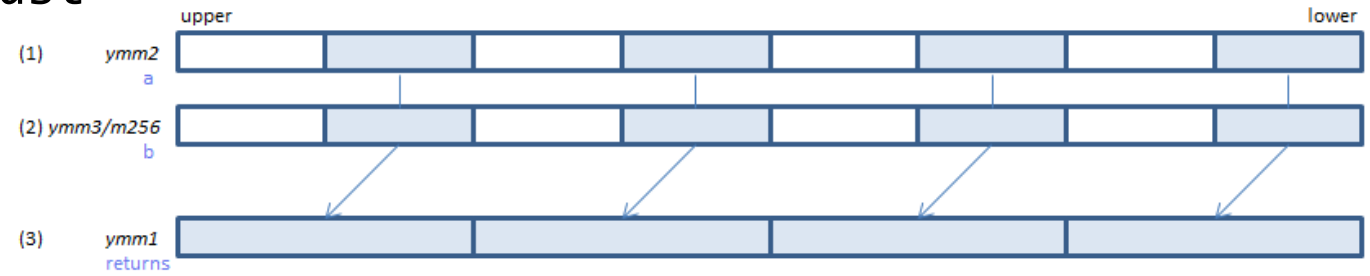


1. Arithmetic and Logic

- Types
  - mul

★ **Example**

```
__m256i _mm_mul_epi32 (__m256i a, __m256i b) {
    for j := 0 to 3 {
        i := j*64
        dst[i+63:i] := a[i+31:i] * b[i+31:i]
    }
    dst[MAX:256] := 0
    return dst
}
```



2. Compare

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

★ Example

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

3. Load/Store

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

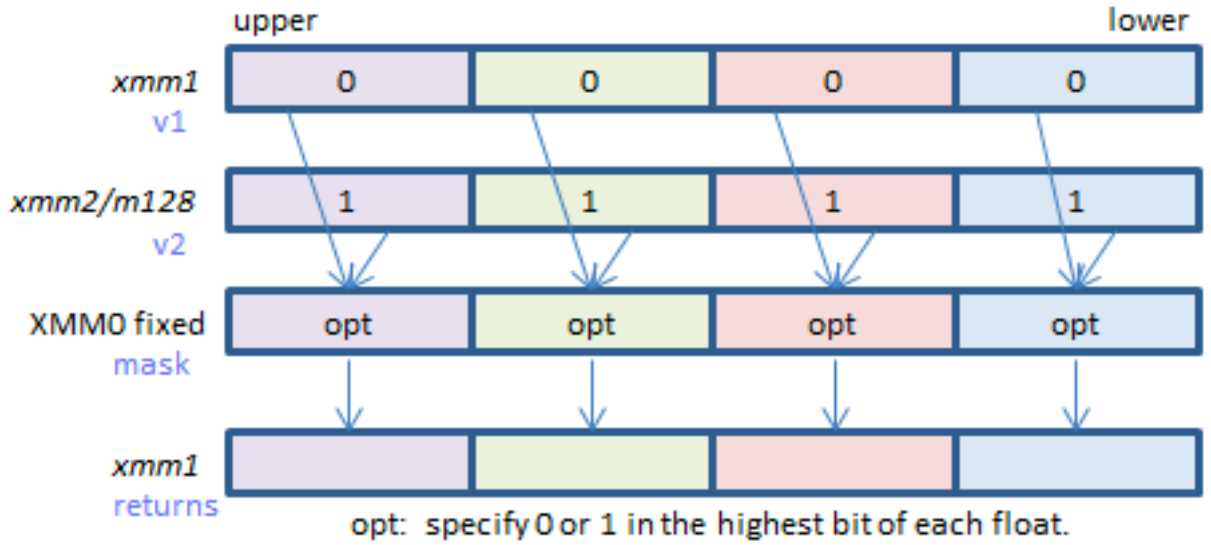
★ Example

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

4. Shuffle: Blending

★ Example

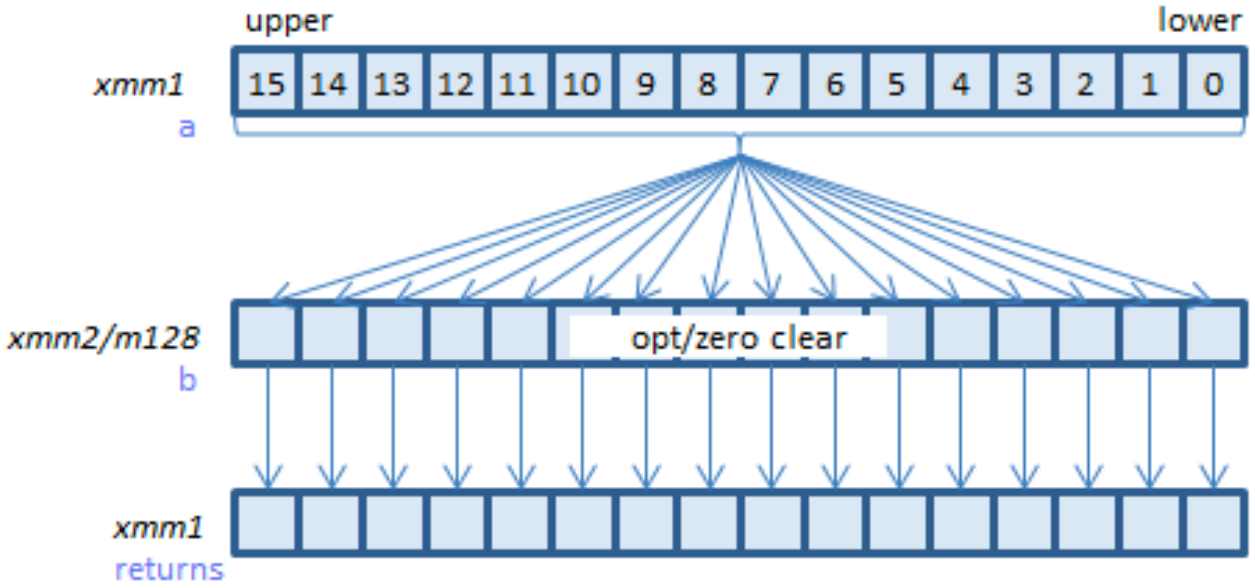
```
__m128 _mm_blendv_ps(__m128 v1, __m128 v2, __m128 mask)
```



4. Shuffle: all-to-all shuffle

★ Example

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

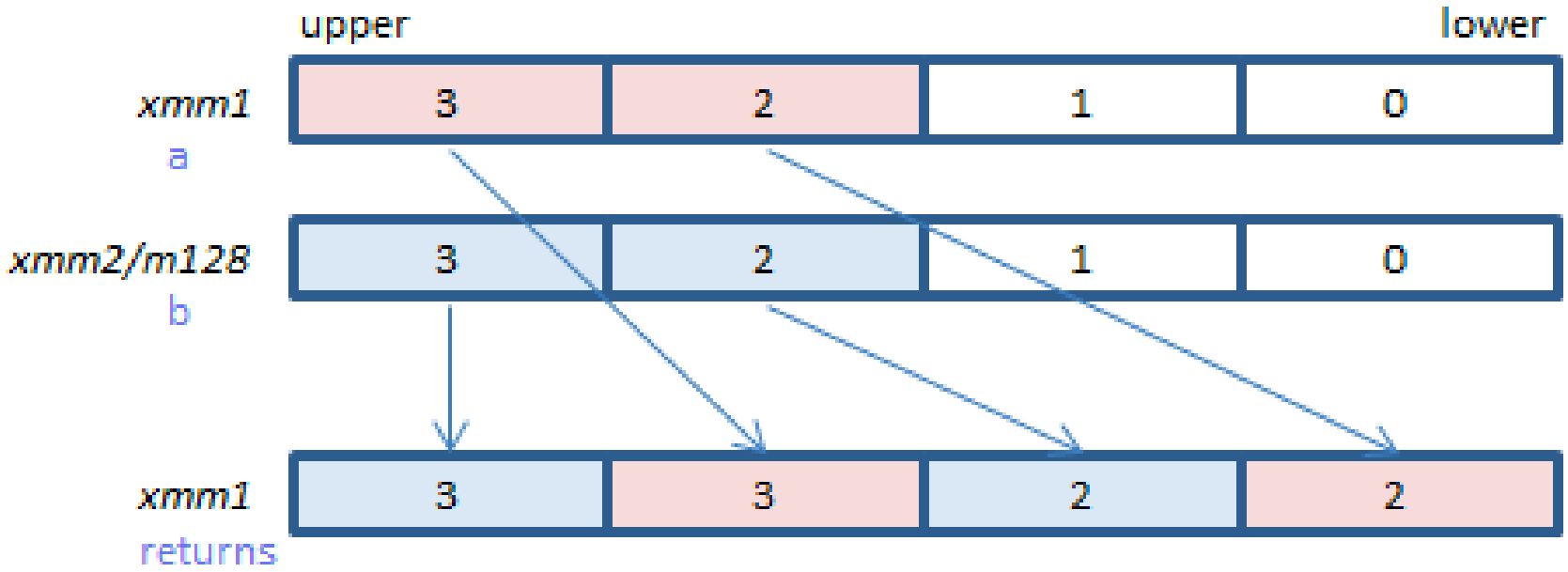


4. Shuffle: unpack

- Unpackhi
- Unpacklo

★ Example

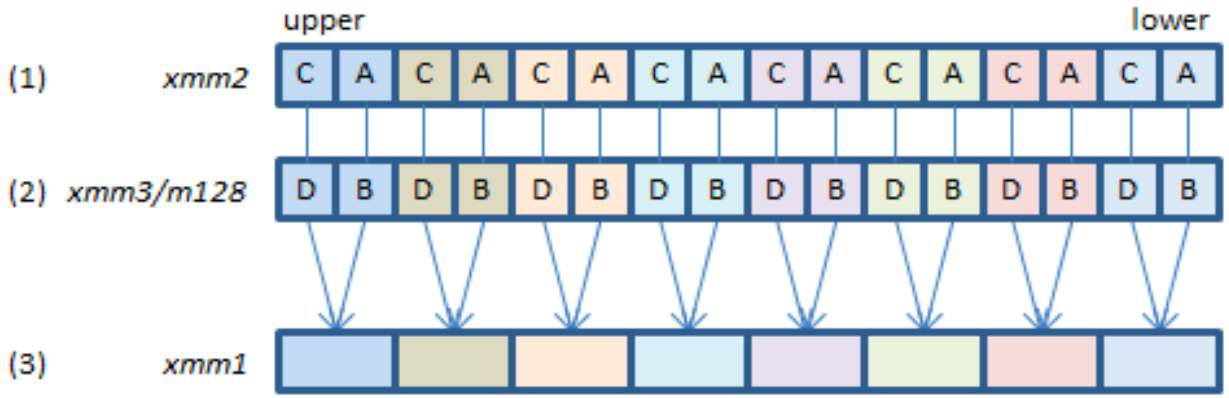
```
__m128 _mm_unpackhi_ps(__m128 a, __m128 b)
```



5. Non-SIMD: dot product instructions

★ Example

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

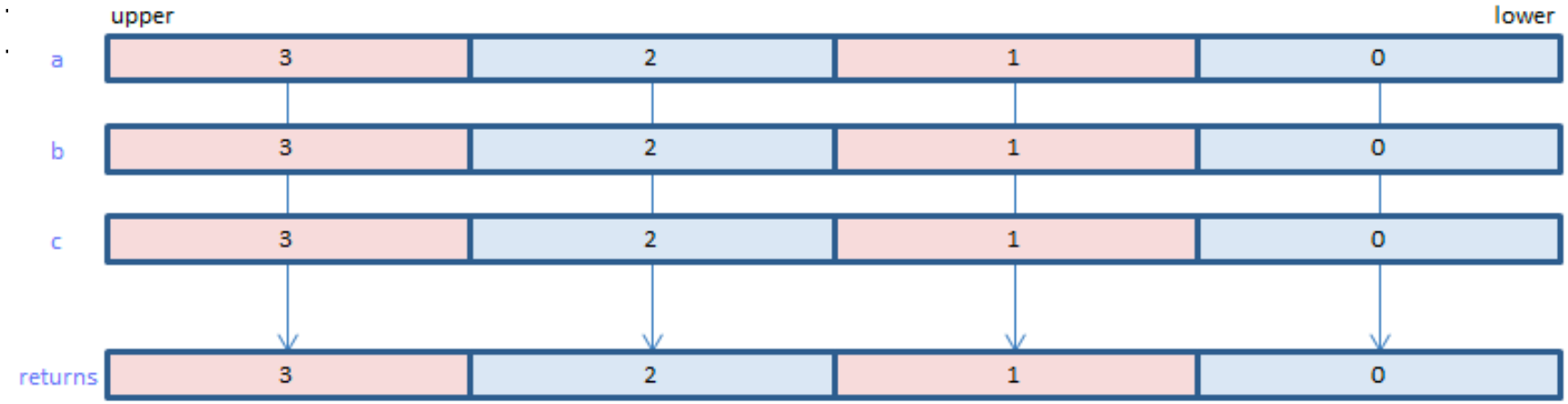


Calculate A*B+C*D and set to each WORD of (3). Signed saturation: set -32768 / 32767 on overflow.
Each BYTE of (1) is unsigned. Each BYTE of (2) is signed.

5. Non-SIMD: addsub

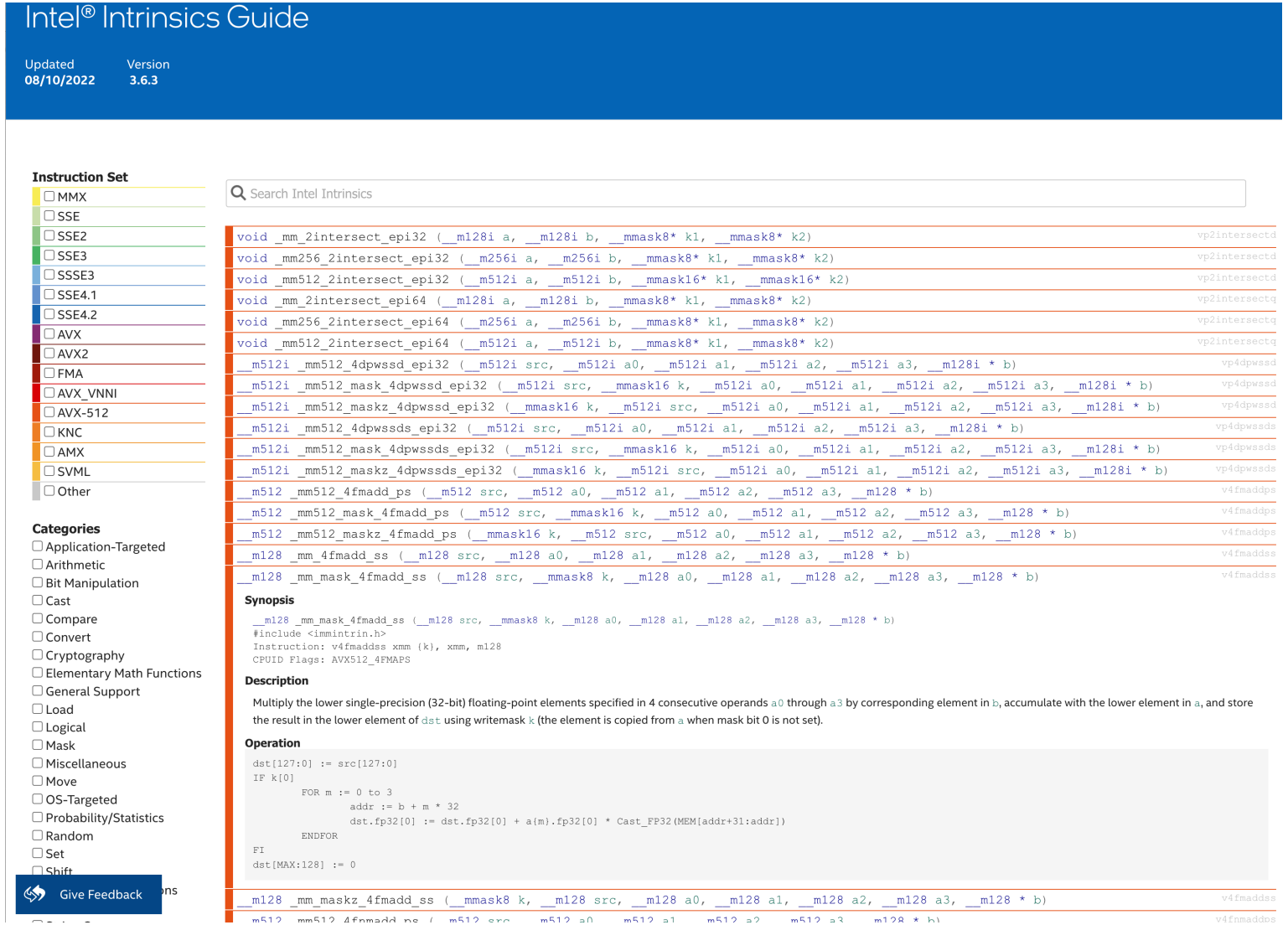
★ Example

```
__m256d _mm256_addsub_pd(__m256d a, __m256d b) {
  for j := 0 to 3 {
    i := j*64
    if ((j & 1) == 0)
      dst[i+63:i] := a[i+63:i] - b[i+63:i]
    else
      dst[i+63:i] := a[i+63:i] + b[i+63:i]
  }
```



And a lot more instructions..

Google "Intel Intrinsics Guide"



Over 7000 Intrinsics!

Let It Be Vectorized (by the Compiler)

Vectorizing Compiler

- Two types of Vectorizers

1. Loop Vectorizer
2. Superword Level Parallelism(SLP) Vectorizer

Loop Vectorization

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

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

Loop Vectorization

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

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

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

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;
}
```

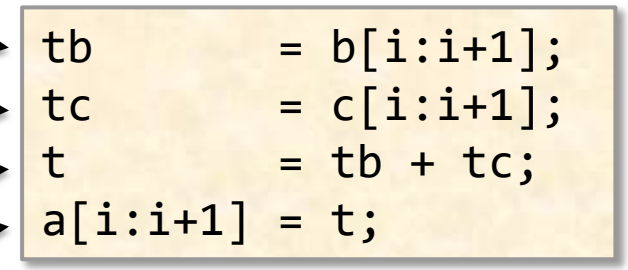
```
for (int i = 0; i < n; i += 2) {
  tb   = b[i];
  tb2  = b[i+1];
  tc   = c[i];
  tc2  = c[i+1];
  t    = tb + tc;
  t2   = tb2 + tc2;
  a[i] = t;
  a[i] = t2;
}
```

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

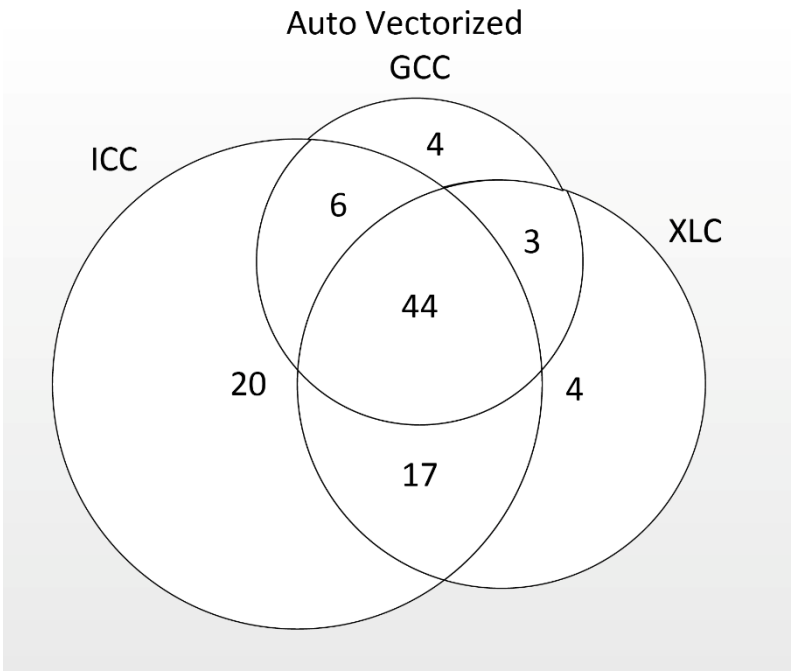
```
for (int i = 0; i < n; i++) {
  tb   = b[i];
  tc   = c[i];
  t    = tb + tc;
  a[i] = t;
}
```

```
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

What makes it hard for the compiler to vectorize

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

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

What makes it hard for the compiler to vectorize

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

1	Loop with unknown trip count	https://www.godbolt.org/z/oh15M6baK
2	Cannot prove independence	https://www.godbolt.org/z/eYYadjraP
3	Loop carried dependences	https://www.godbolt.org/z/6WEcrMaM4
4	Reductions	https://www.godbolt.org/z/q7sxv8cEa
5	Control-flow	https://www.godbolt.org/z/5v66a8Mq6
6	Non-contiguous data	https://www.godbolt.org/z/bGc1GvThM https://www.godbolt.org/z/31bzd4PxY
7	Cost model failures	https://godbolt.org/z/b3cYo6388
8	Vectorization of function calls	https://www.godbolt.org/z/ra8Esjjx9
9	Different hardware versions	https://www.godbolt.org/z/dd53r1Mf7

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

1. Loop with unknown trip count

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

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

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

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

1. Loop with unknown trip count

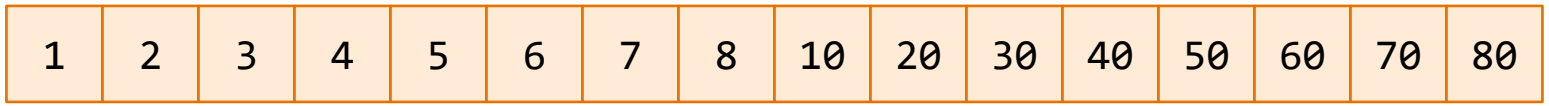
```
bar:                                    # @bar
        cmp     esi, edx
        jge     .LBB0_8
        movsxd  r8, esi
        movsxd  r10, edx
        mov     r9, r10
        sub     r9, r8
        cmp     r9, 8
        jae     .LBB0_3
        mov     rsi, r8
        jmp     .LBB0_14
.LBB0_3:
        cmp     r9, 16
        jae     .LBB0_9
        xor     edx, edx
        jmp     .LBB0_5
.LBB0_9:
        mov     rdx, r9
        and     rdx, -16
        vbroadcastss    zmm1, xmm0
        lea     rcx, [rdi + 4*r8]
        xor     esi, esi
.LBB0_10:                               # =>This Inner Loop He
        vmovups zmmword ptr [rcx + 4*rsi], zmm1
        add     rsi, 16
        cmp     rdx, rsi
        jne     .LBB0_10
        cmp     r9, rdx
        je      .LBB0_8
        test    r9b, 8
        jne     .LBB0_5
        add     rdx, r8
        mov     rsi, rdx
        jmp     .LBB0_14
.LBB0_5:
        mov     rcx, r9
        and     rcx, -8
        lea     rsi, [rcx + r8]
        vbroadcastss    ymm1, xmm0
        lea     rax, [rdi + 4*r8]
.LBB0_6:                                # =>This Inner Loop He
        vmovups ymmword ptr [rax + 4*rdx], ymm1
        add     rdx, 8
        cmp     rcx, rdx
        jne     .LBB0_6
        cmp     r9, rcx
        je      .LBB0_8
.LBB0_14:                               # =>This Inner Loop He
        vmovss  dword ptr [rdi + 4*rsi], xmm0
        inc     rsi
        cmp     r10, rsi
        jne     .LBB0_14
.LBB0_8:
        vzeroupper
        ret
```

```
bar:                                    # @bar
        test    edx, edx
        jle     .LBB0_13
        mov     eax, edx
        cmp     edx, 8
        jae     .LBB0_3
        xor     ecx, ecx
        jmp     .LBB0_12
.LBB0_3:
        cmp     edx, 16
        jae     .LBB0_5
        xor     ecx, ecx
        jmp     .LBB0_9
.LBB0_5:
        mov     ecx, eax
        and     ecx, -16
        vbroadcastss    zmm1, xmm0
        lea     rdx, [4*rax]
        and     rdx, -64
        xor     esi, esi
.LBB0_6:                                # =>This Inner Loop Header: Depth=1
        vmovups zmmword ptr [rdi + rsi], zmm1
        add     rsi, 64
        cmp     rdx, rsi
        jne     .LBB0_6
        cmp     rcx, rax
        je      .LBB0_13
        test    al, 8
        je      .LBB0_12
.LBB0_9:
        mov     rdx, rcx
        mov     ecx, eax
        and     ecx, -8
        vbroadcastss    ymm1, xmm0
.LBB0_10:                               # =>This Inner Loop Header: Depth=1
        vmovups ymmword ptr [rdi + 4*rdx], ymm1
        add     rdx, 8
        cmp     rcx, rdx
        jne     .LBB0_10
        cmp     rcx, rax
        je      .LBB0_13
.LBB0_12:                               # =>This Inner Loop Header: Depth=1
        vmovss  dword ptr [rdi + 4*rcx], xmm0
        inc     rcx
        cmp     rax, rcx
        jne     .LBB0_12
.LBB0_13:
        vzeroupper
        ret
```

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

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

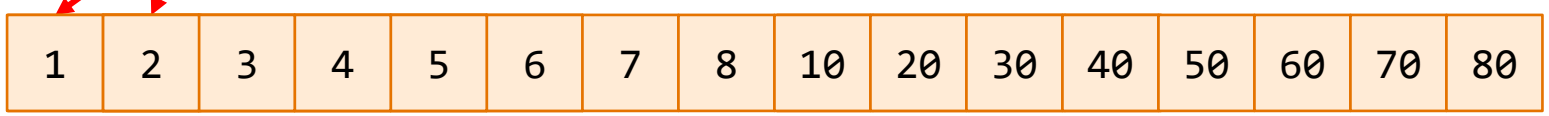
```
void bar(float * A, float * B) {
  for (int i = 0; i < 4; ++i)
    A[i] += B[i];
}
```

2. Cannot Prove Independence

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

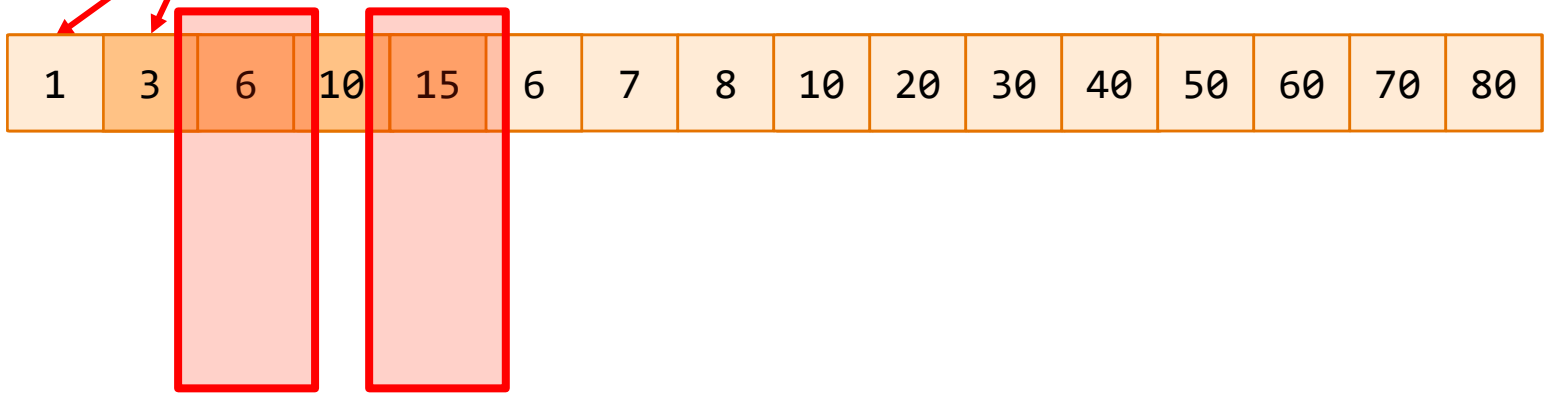
2. Cannot Prove Independence

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



2. Cannot Prove Independence

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



- Vector Results ≠ Scalar Results
- Dependence violated

2. Cannot Prove Independence

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

LLVM can still vectorize!

- How?

By checking if the address ranges overlap

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

However, Can save all these hassles by….

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

https://www.godbolt.org/z/eYYadjraP

2. Cannot Prove Independence

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

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

https://www.godbolt.org/z/eYYadjraP

3. Loop Carried Dependence

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

1024

A[i-2];

A[i-100];

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

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

3. Loop Carried Dependence

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

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

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

4. Reductions

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

https://www.godbolt.org/z/q7sxv8cEa

4. Reductions

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

5. Control Flow

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

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

5. Control Flow

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

6. Non-contiguous data

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

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

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

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

6. Non-contiguous data

```
.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    zmm0, xmm0
        xor     eax, eax
        vmovaps zmm1, zmmword ptr [rip + .LCPI0_0] # zmm1 = [0,16,1,17,2,18,3,19,4,20,5,21,6,22,7,23]
        vmovaps zmm2, zmmword ptr [rip + .LCPI0_1] # 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 zmm3, zmmword ptr [rdi + 4*rax]
        vaddps  zmm4, zmm3, zmm0
        vmulps  zmm3, zmm3, zmm0
        vmovaps zmm5, zmm4
        vpermt2ps       zmm5, zmm1, zmm3
        vpermt2ps       zmm4, zmm2, zmm3
        vmovups zmmword ptr [rsi + 8*rax + 64], zmm4
        vmovups zmmword ptr [rsi + 8*rax], zmm5
        add     rax, 16
        cmp     rax, 1024
        jne     .LBB0_1
        vzeroupper
        ret
```

```
bar:                                    # @bar
        xor     eax, eax
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
        vmovups zmm0, zmmword ptr [rdx + 4*rax]
        kxnorw  k1, k0, k0
        vxorps  xmm1, xmm1, xmm1
        vgatherdps      zmm1 {k1}, zmmword ptr [rdi + 4*zmm0]
        vmovups zmmword ptr [rsi + 4*rax], zmm1
        add     rax, 16
        cmp     rax, 1024
        jne     .LBB0_1
        vzeroupper
        ret
```

7. Cost Model Failures

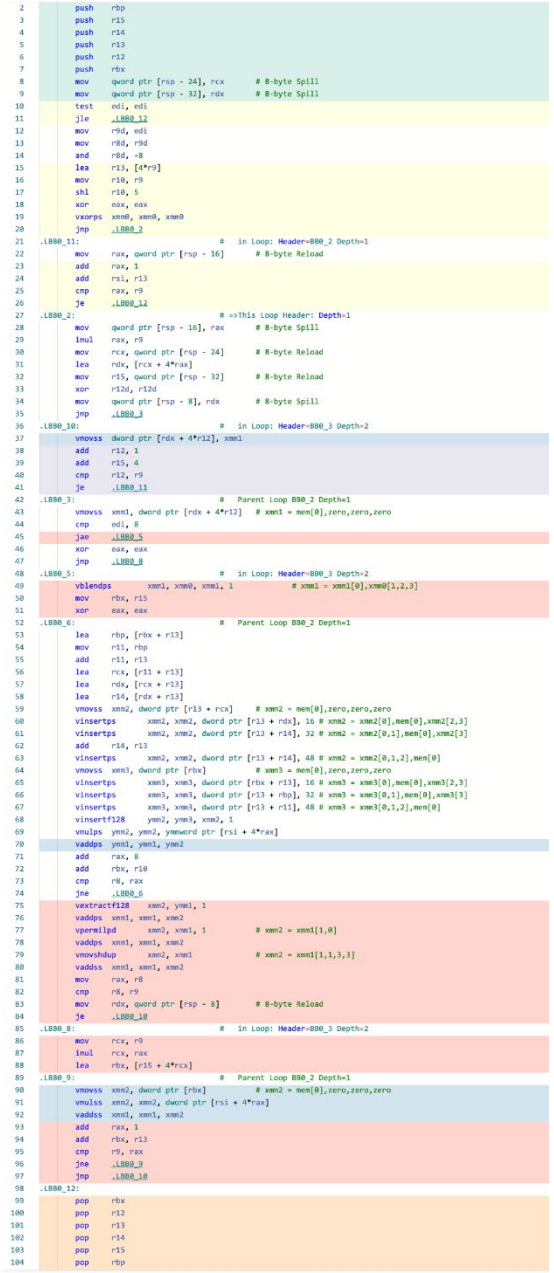
```
void matmul(float a[restrict N][N], float b[restrict N][N],
            float c[restrict N][N], int N) {
   for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
         for (int k = 0; k < N; k++)
            c[i][j] += a[i][k] * b[k][j];
}
```

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

https://godbolt.org/z/b3cYo6388

7. Cost Model Failures

```
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

```
#include <math.h>

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

- Works for a few built-in functions

- Note, need to add the exact library to the command line

https://www.godbolt.org/z/ra8Esjjx9

8. Vectorization of Function Calls

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

9. Different Hardware Versions

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

- Different Intel chips have different vector hardware

MMX, SSE, SSE2, SSE3, AVX, AVX2, AVX512, AVX512VNNI etc.

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

9. Different Hardware Versions

x86-64 clang 15.0.0 | -O3 -fno-vectorize -ffast-math -fno-unroll-loops

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

Output (0/0) x86-64 clang 15.0.0 - 974ms (11885B) ~234 lines filtered Compiler License

x86-64 clang 15.0.0 | -O3 -msse2 -ffast-math -fno-unroll-loops

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

Output (0/0) x86-64 clang 15.0.0 - cached (11778B) ~229 lines filtered Compiler License

x86-64 clang 15.0.0 (C, Editor #1, Compiler #3)

x86-64 clang 15.0.0 | -O3 -mavx2 -ffast-math -fno-unroll-loops

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

x86-64 clang 15.0.0 (C, Editor #1, Compiler #4)

x86-64 clang 15.0.0 | -O3 -mavx512vnni -ffast-math -fno-unroll-loops

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

Blood Sweat and Tears: Hand Vectorization using Intrinsics

Example: SAXPY

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

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

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

Example: SAXPY

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

Compiler can do this automatically too.

Running time (n=1024)

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

Vectorizing Reductions

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

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

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

Vectorizing Reductions (cont.)

```
float reduce_vector(__m256 v) {
  __m128 v_0_4 = _mm256_extracti128_si256(v, 0);
  __m128 v_4_8 = _mm256_extracti128_si256(v, 1);
  // t  = {v3+v7, v2+v6,       v1+v5,   v0+v4}
  __m128 t = _mm_add_ps(v_0_4, v_4_8);
  // t1 = {v2+v6, v3+v7,       v0+v4,   v1+v5}
  __m128 t1 = _mm_shuffle_ps(t, t, _MM_SHUFFLE(2, 3, 0, 1));
  // t2 = {_,     v2+v6+v3+v7, _, v0+v4+v1+v5}
  __m128 t2 = _mm_add_ps(t, t1);
  // t3 = {_,     _,           _, v2+v6+v3+v7}
  __m128 t3 = _mm_shuffle_ps(t2, t2, _MM_SHUFFLE(0, 0, 0, 2));
  // t4 = {_,     _,           _, v0+v4+v1+v5+v2+v6+v3+v7}
  __m128 t4 = _mm_add_ps(t2, t3);
  // Extract the first lane
  return _mm_cvtss_f32(t4);
}
```

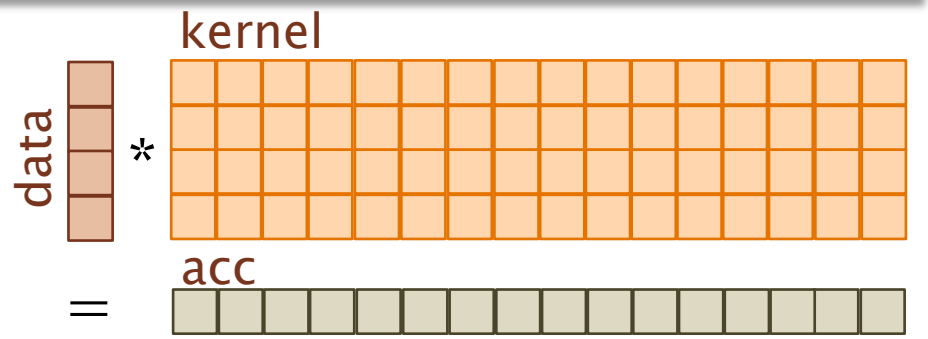
Vectorizing Control Flow

```
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];
}
```



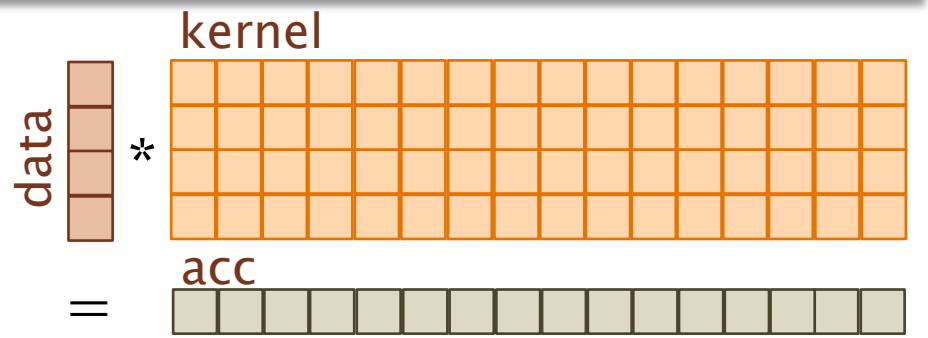
2.2x faster than scalar.

- Compiler vectorizes the inner loop
  - Similar to a reduction

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

Vectorizing the Dot Product

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

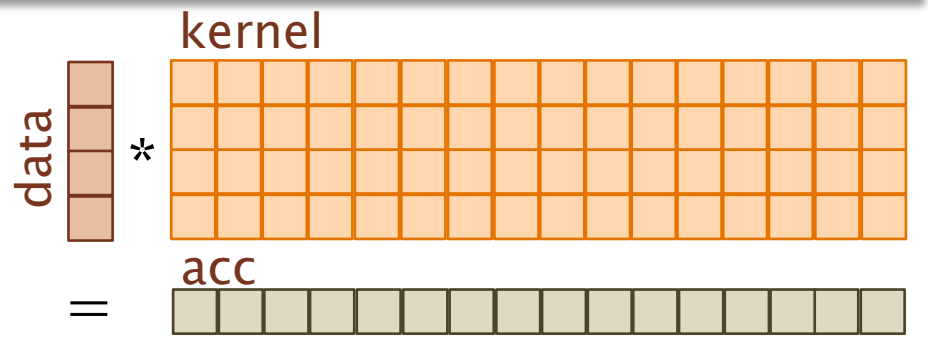


2.2x faster than scalar.

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

Vectorizing the Dot Product

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



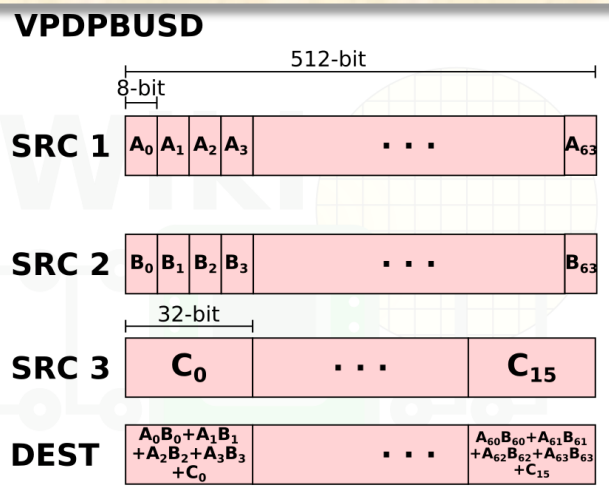
2.2x faster than scalar.

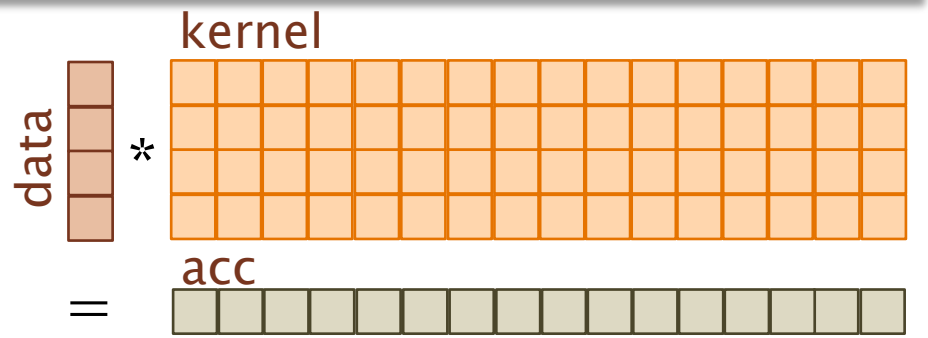
- Compiler vectorizes the inner loop
  - Similar to a reduction
- Can we do better?

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

Vectorizing the Dot Product

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





2.2x faster than scalar.

11x faster than scalar.

```
__m512i acc_vec = _mm512_loadu_epi32(acc);
// {data0, data1, data2, data3, …, data0, data1, data2, data3}
__m512i data_vec = _mm512_set1_epi32(*(int32_t)data);
__m512i kernel_vec = _mm512_loadu_epi32(&kernel);
_mm512_dpbusd_epi32(acc_vec, data_vec, kernel_vec);
```

Conclusion

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