CPSC 4240/5240: Parallel Programming Techniques

Lecture 18: Vectorization

Lecturer: Quanquan C. Liu

Spring 2026

Some slides courtesy of the instructors of MIT 6.172 and 6.106 Lecturers

Vectorization

- What is **vectorization**?
  - Converting algorithms on single data elements (**scalars**) into algorithms that operate on **vectors of data** (implemented as arrays, sequences or vectors in C++)
  - Example:
    - Replacing loops over elements by vector operations
    - C = A + B where A and B are both vectors
  - Vectorized operations can utilize **hardware-specific optimizations**

Example C++ Code for Adding Two Vectors

Scalar Addition

```
for (size_t i = 0; i < N; ++i) {
        C[i] = A[i] + B[i];
 }
```

g++ -O3 -mavx2 -std=c++11 vectorization-add.cpp -o vector

Vectorized using AVX Intrinsics

```
size_t i = 0;
for (; i + 15 < N; i += 16) {
        __m256 a1 = _mm256_load_ps(A + i);
        __m256 a2 = _mm256_load_ps(A + i + 8);
        __m256 b1 = _mm256_load_ps(B + i);
        __m256 b2 = _mm256_load_ps(B + i + 8);
        __m256 c1 = _mm256_add_ps(a1, b1);
        __m256 c2 = _mm256_add_ps(a2, b2);
        _mm256_store_ps(C + i, c1);
        _mm256_store_ps(C + i + 8, c2);
}

for (; i < N; i++) {
        C[i] = A[i] + B[i];
}
```

Example C++ Code for Adding Two Vectors

Scalar Addition

Vectorized using AVX Intrinsics

Runtimes:

Scalar addition took **0.938539 seconds**

Optimized vectorized addition took **0.877878 seconds**

```
g
v
```

```
for (; i < N; i++) {
        C[i] = A[i] + B[i];
}
```

Example C++ Code for Adding Two Vectors

Scalar Addition

```
for (size
        C[i
 }
```

Vectorized using AVX Intrinsics

```
size_t i = 0;
for (; i + 15 < N; i += 16) {
                                          A + i);
                                          A + i + 8);
                                          B + i);
                                          B + i + 8);
                                          1, b1);
                                          2, b2);

        _mm256_store_ps(C + i + 8, c2);
}

for (; i < N; i++) {
        C[i] = A[i] + B[i];
}
```

AVX is an Intel CPU feature; can only compile on Intel CPUs!

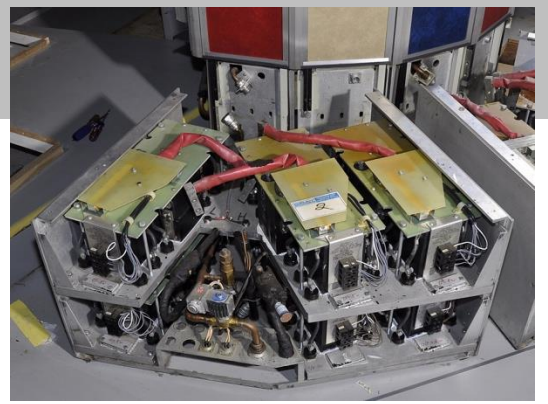
```
g++ -O3 -mavx2 -std=c++11
vectorization-add.cpp -o vector
```

AVX Intrinsics

What is it?

Vectorization History

- The First Supercomputers were developed in the 70's

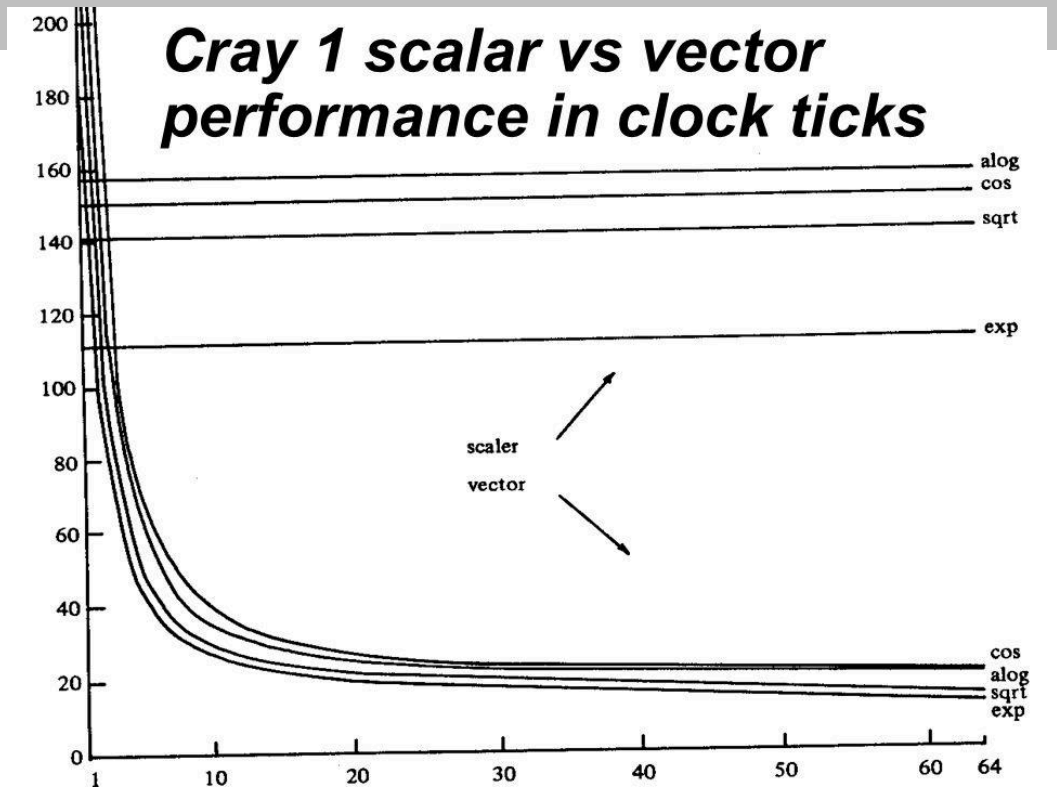


Seymour Cray

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

Vectorization History

- 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

Multimedia Instructions

- Multimedia ISA extensions came in the 90's
- Multimedia instructions are
  - **SIMD: Single Instruction Multiple**
  - Integrate
  - Short and
  - No startu
- Different r
  - The **mult**
  - Extension of the original floating-point registers
  - Wide loads direct to the multimedia registers

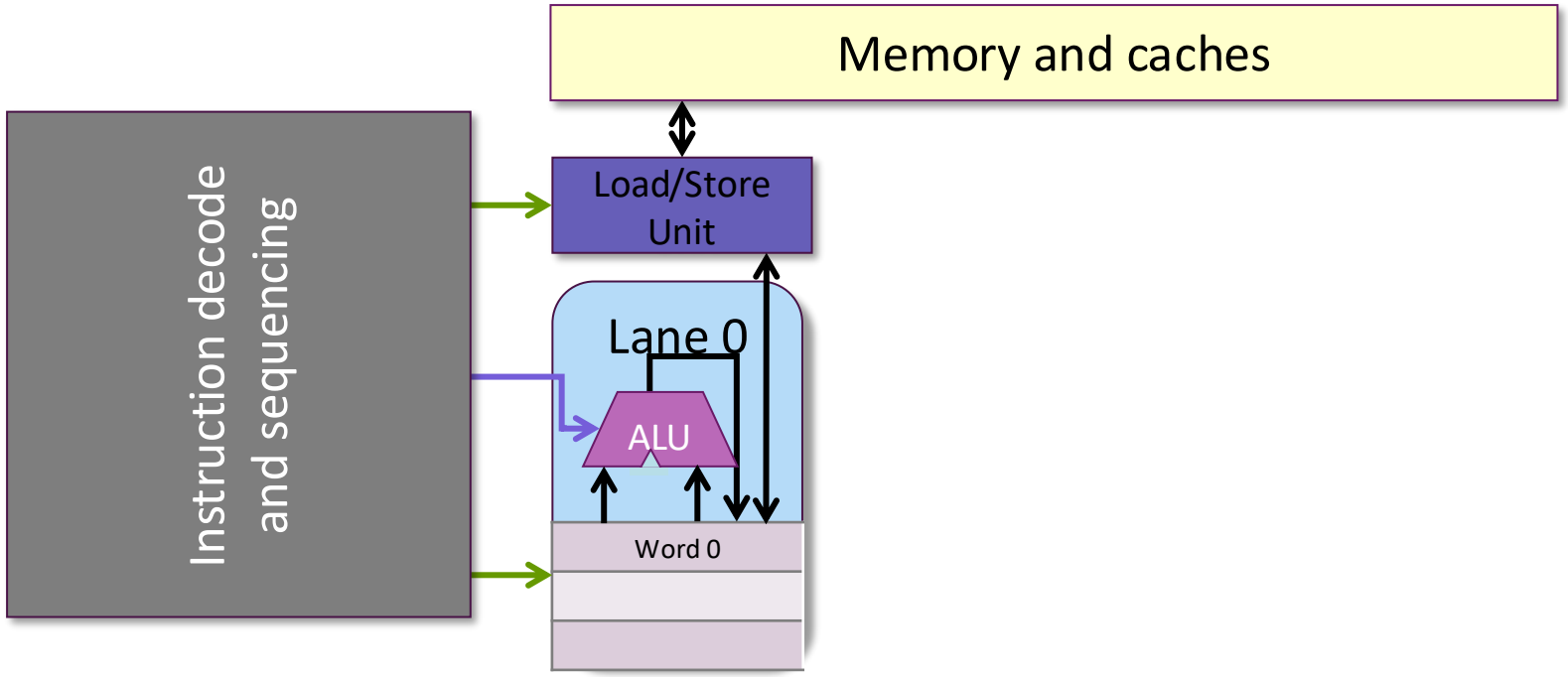
These Multimedia Instructions use Intrinsic Parallelization!

Vendor	ISA	Year	Bitwidth
		1996	64 bits
		1998	64 bits
		1999	128 bits
		2001	128 bits
		2006	128 bits
		2008	256 bits
		2011	128 bits
Intel	AVX2	1013	256 bits
Intel	AVX512	2015	512 bits
Intel	AVX512-VNNI	2021	512 bits

What is SIMD?

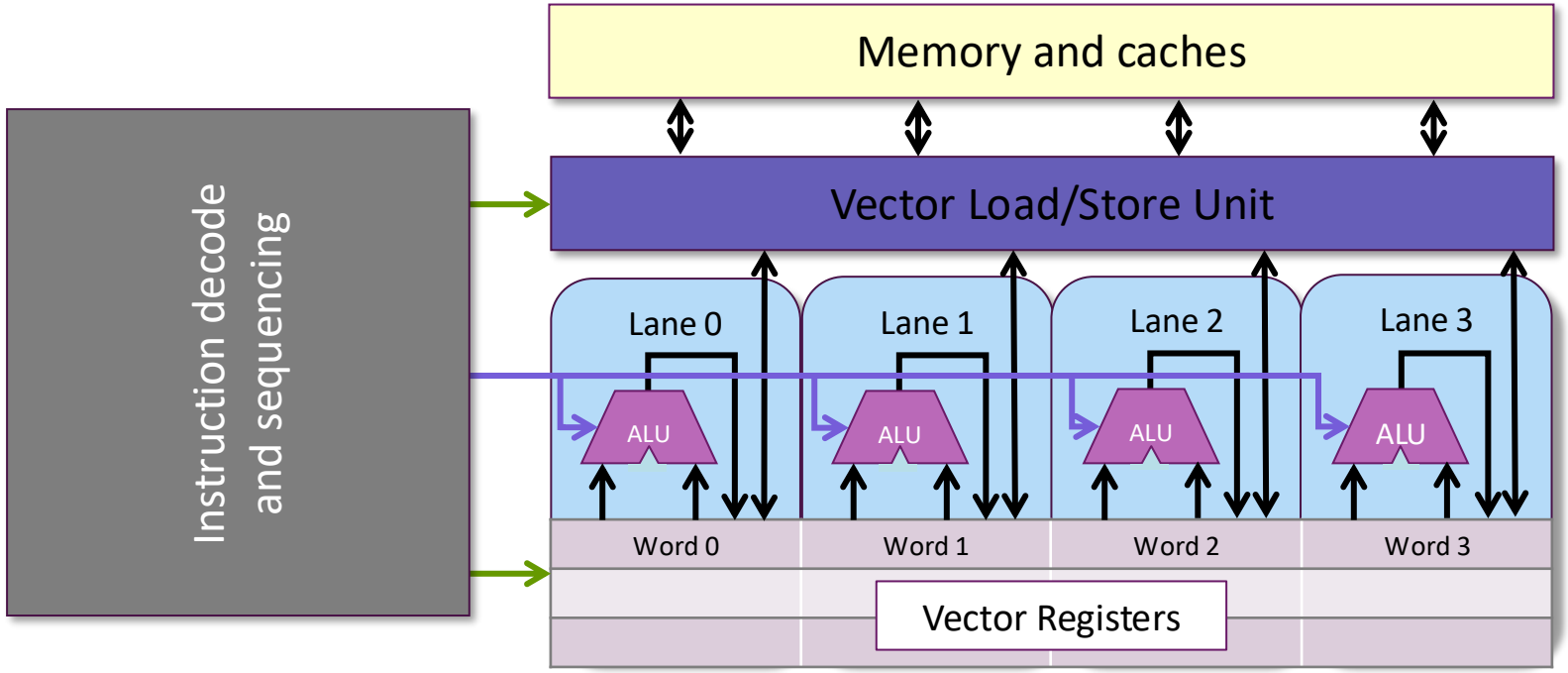
Scalar Hardware

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



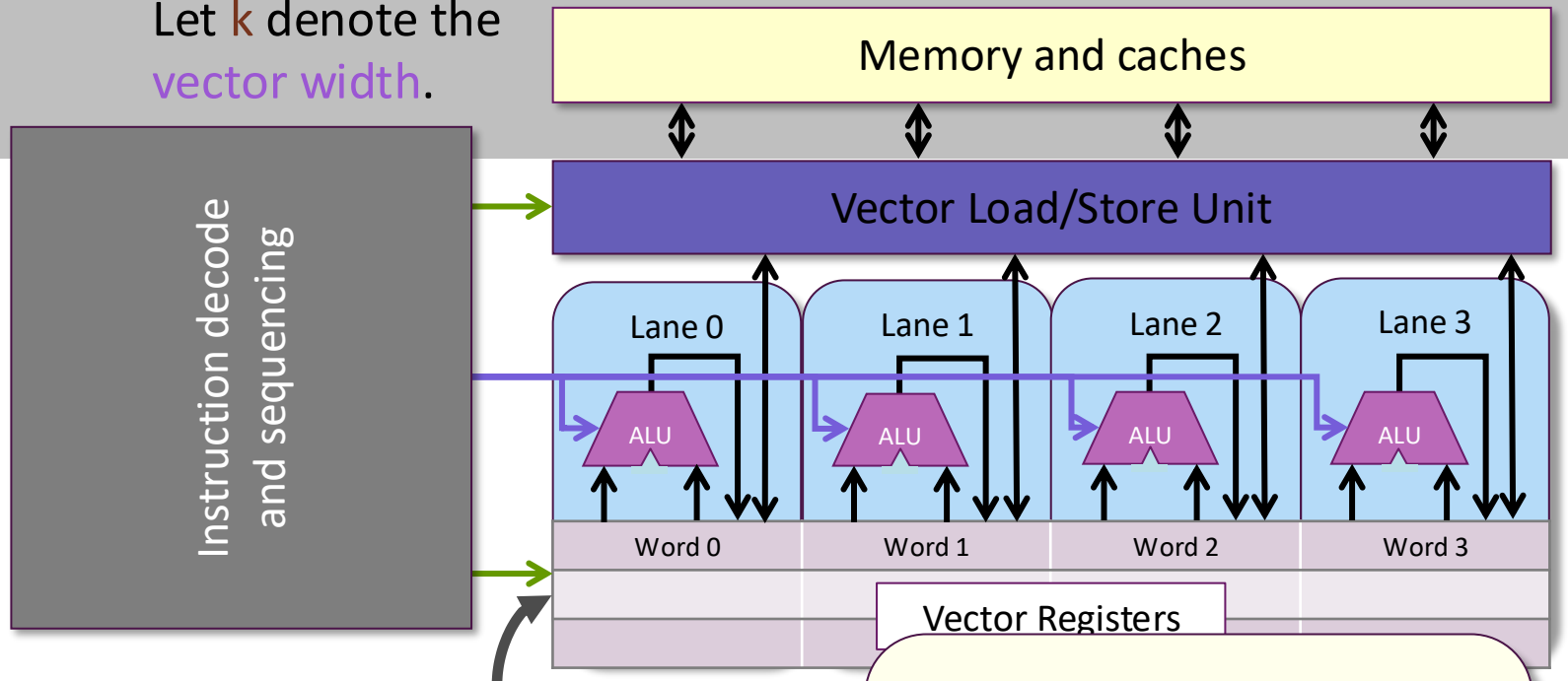
Vector Hardware

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



A k-Wide Vector Unit

Let k denote the vector width.



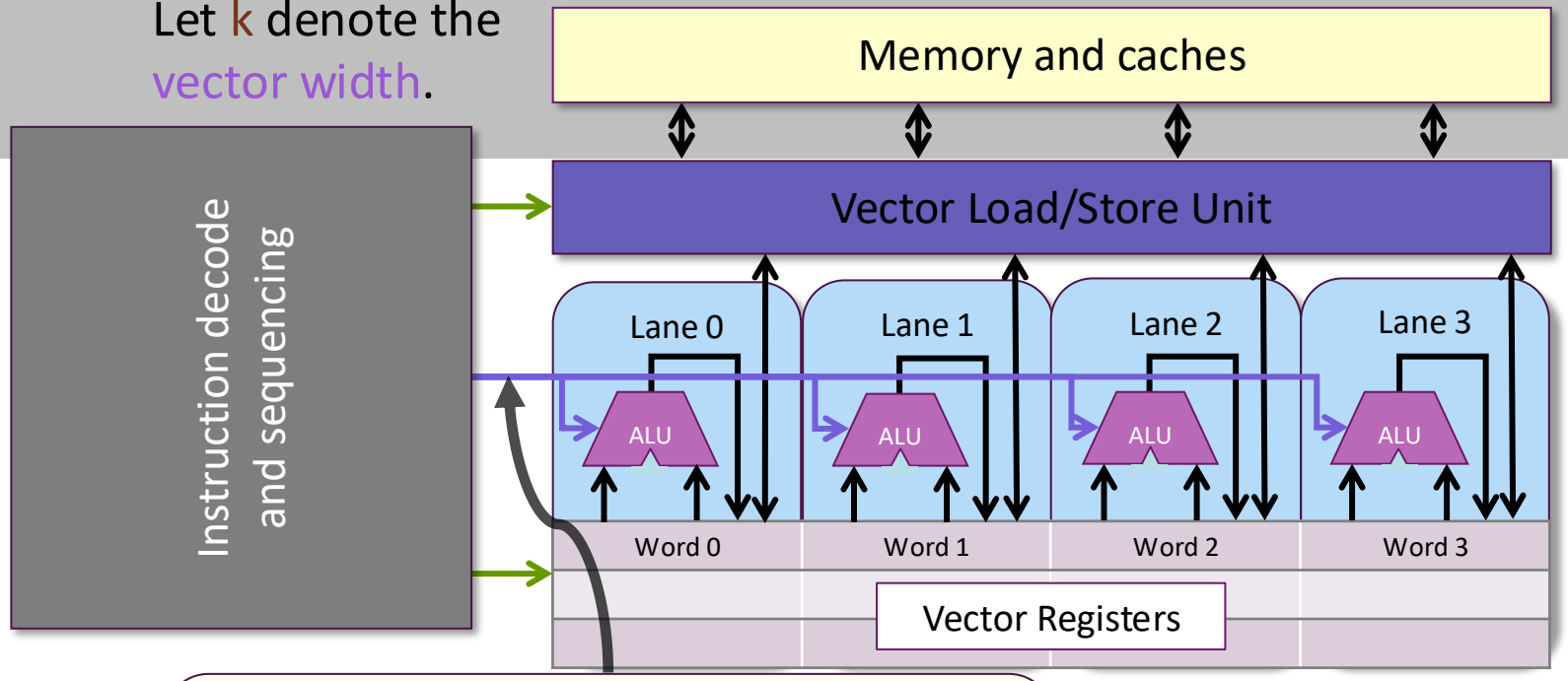
$k = 4$ here

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

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

A k-Wide Vector Unit

Let k denote the vector width.



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

SIMD Instructions

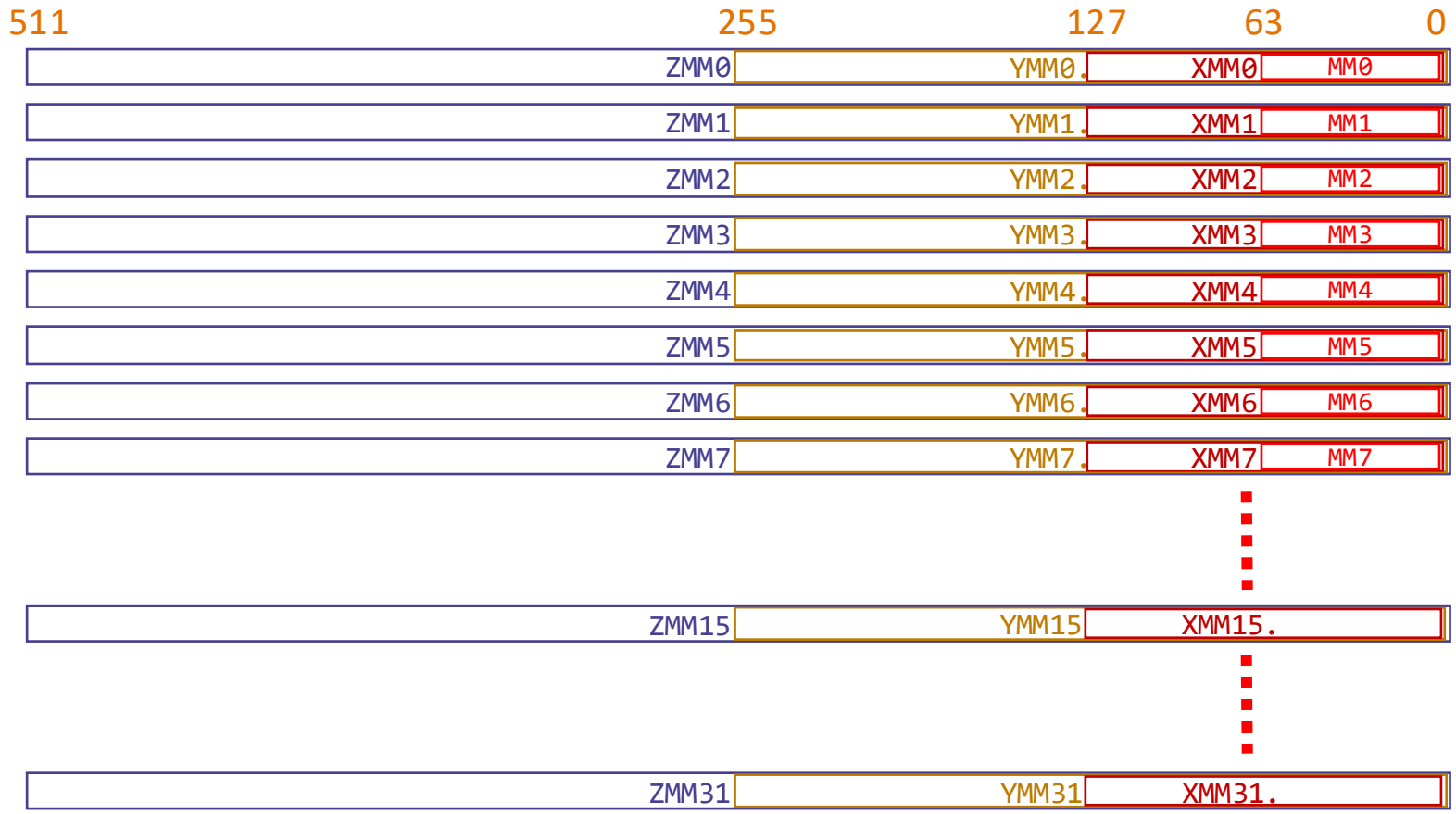
SIMD instructions typically operate in an *elementwise* fashion.

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

Vector Register Aliasing

Different names, but same registers

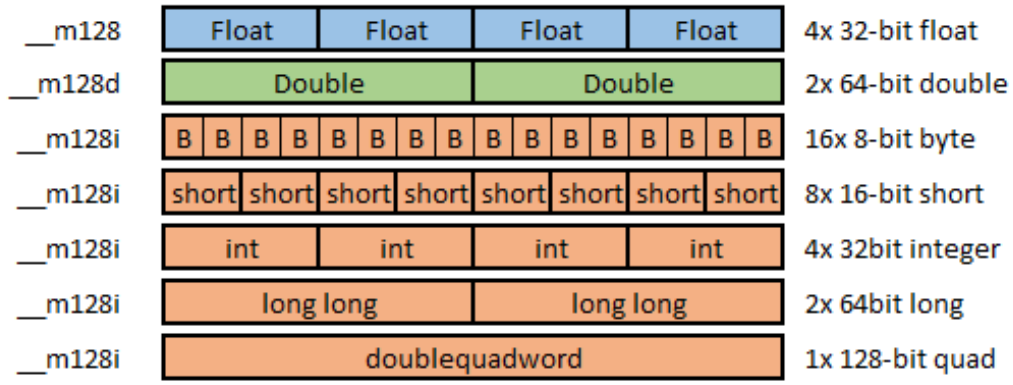
MMX SSE SSE2 AVX AVX512



Vector Register Aliasing

Each register can have different data types

SSE Data Types (16 XMM Registers)

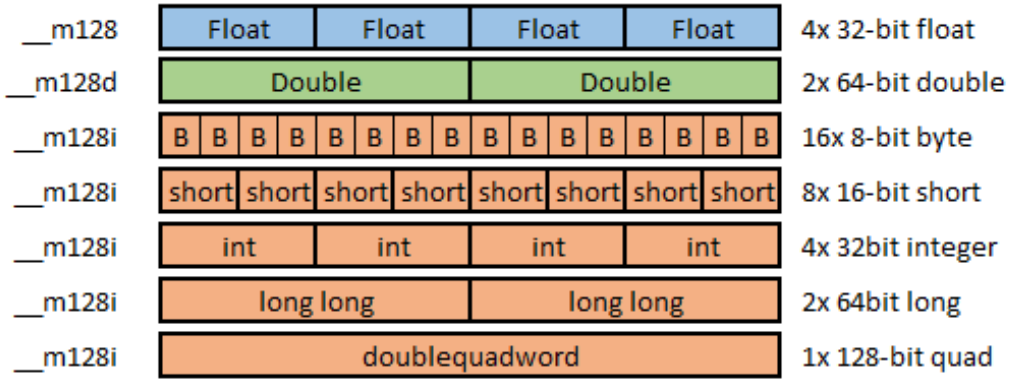


https://www.codingame.com/playgrounds/283/sse-avx-vectorization/what-is-sse-and-avx

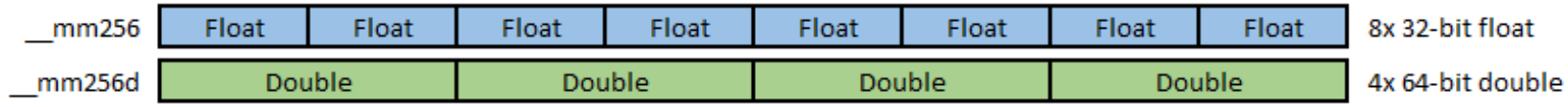
Vector Register Aliasing

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

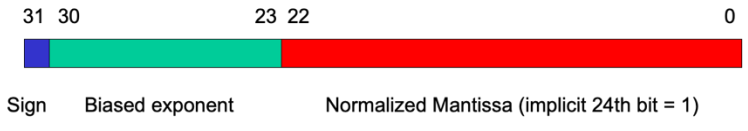
AVX-512 has ZMM registers of 512 bits…

Floating Point

Numbers, Numbers,...Numbers

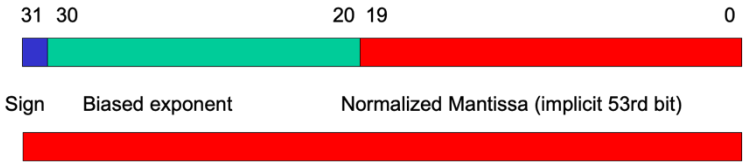
	bits	Smallest value	Biggest value
Short	16	±1	±32,768
Int	32	±1	±2.147...x 10^9
Long long	64	±1	±9.223.... x 10^{18}
IEEE Single Precision	**64**	±1.2 × 10^{-38}	±3.4 × 10^{+38}
IEEE Double Precision	128	±2.2 × 10^{-308}	±1.8 × 10^{+308}
Minfloats	**8**	±0.125	±15,360

- **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

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

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

- Compilers are conservative unless told otherwise

- Use the fast-math flag
- In gcc –ffast-math means disregarding a lot of rules

```
-fno-math-errno,            -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

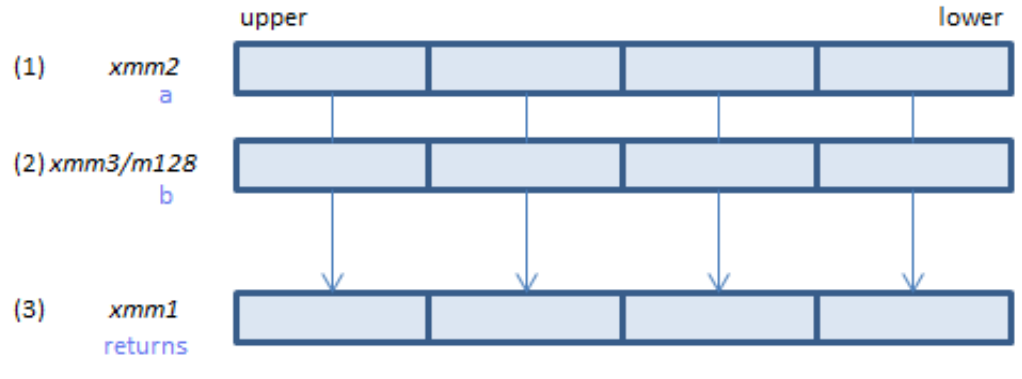
1. Arithmetic and Logic
2. Compare
3. Load/Store
4. Shuffle (Swizzle)
5. Masks
6. Non-SIMD

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
}
```

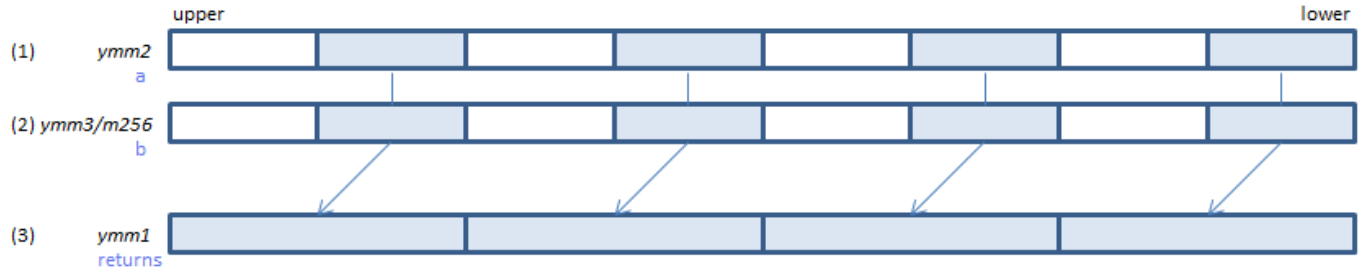


1. Arithmetic and Logic

- 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

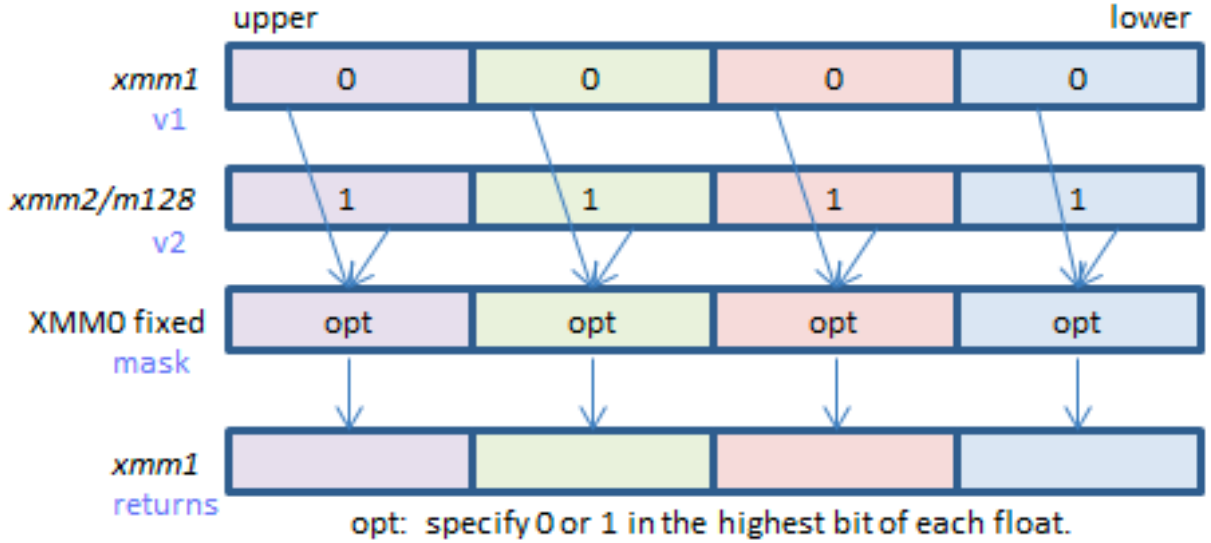
- Example

```
__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

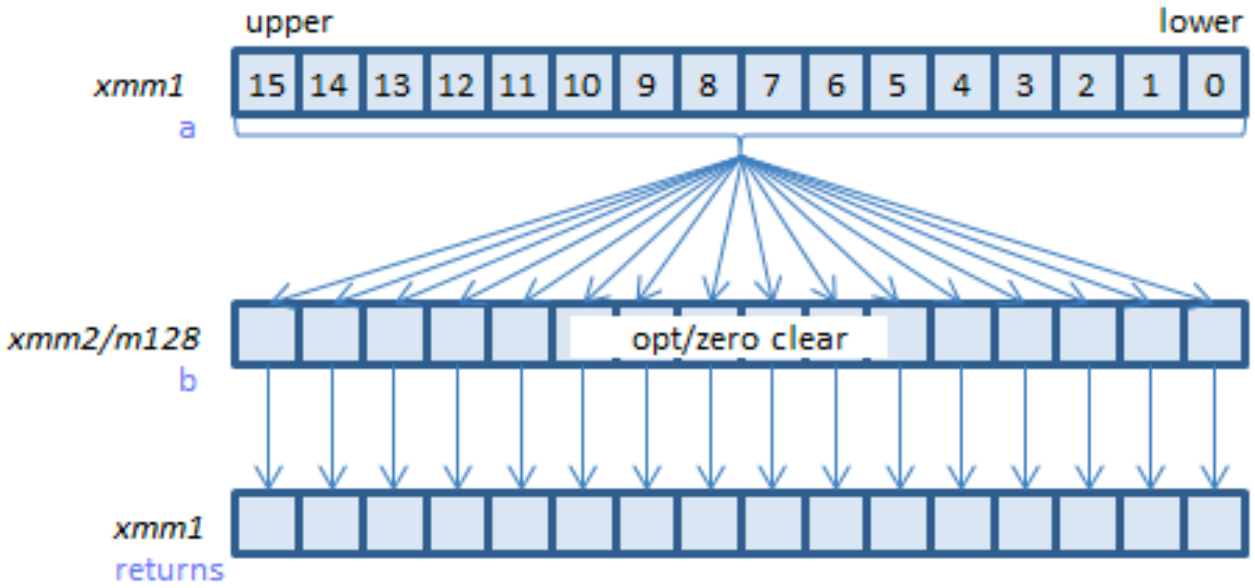
```
__m128 _mm_blendv_ps(__m128 v1, __m128 v2, __m128 mask)
```



4. Shuffle: all-to-all shuffle

- Example

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



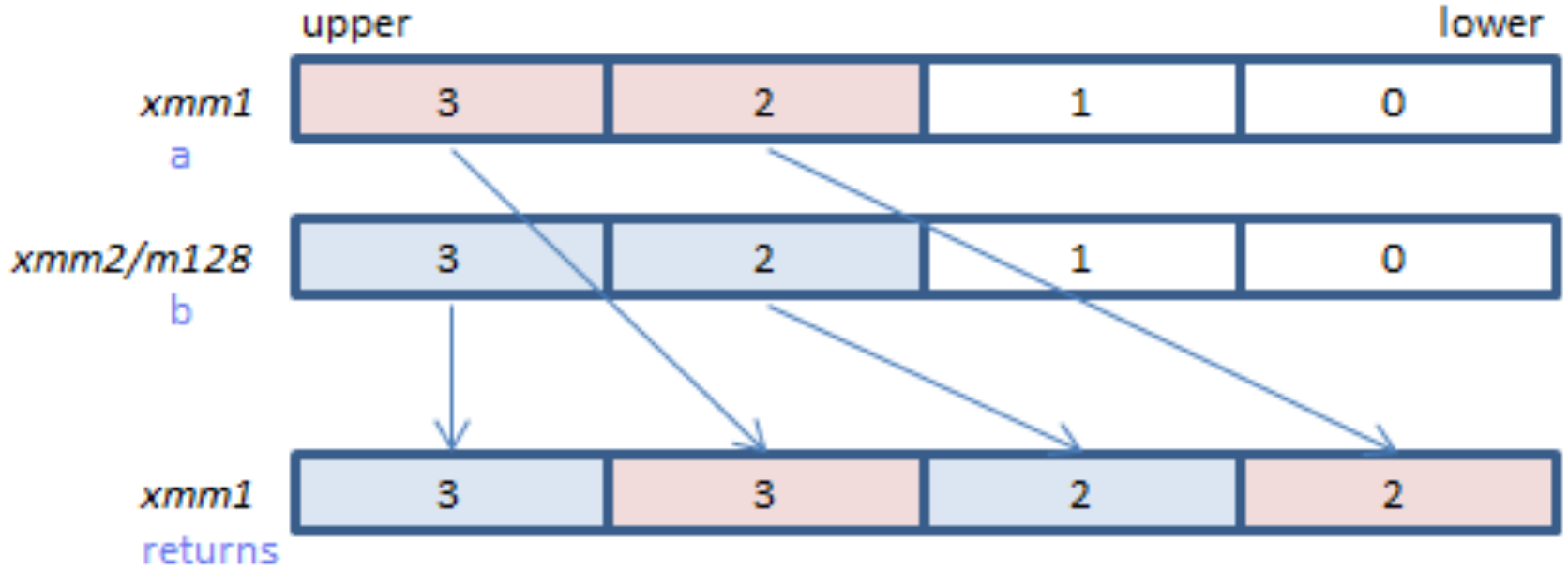
each byte of xmm2/m128:
bit 7 == 0 specifies copying, bit 3:0 specifies 0 to 15
bit 7 == 1 specifies zero clearing

4. Shuffle: unpack

- Unpackhi
- Unpacklo

- Example

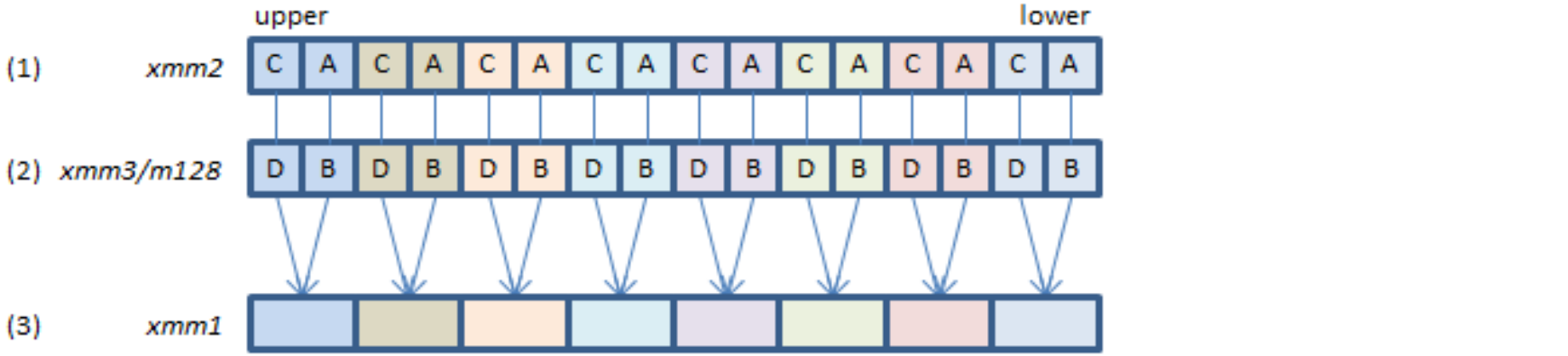
```
__m128 _mm_unpackhi_ps(__m128 a, __m128 b)
```



5. Non-SIMD: dot product instructions

- Example

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

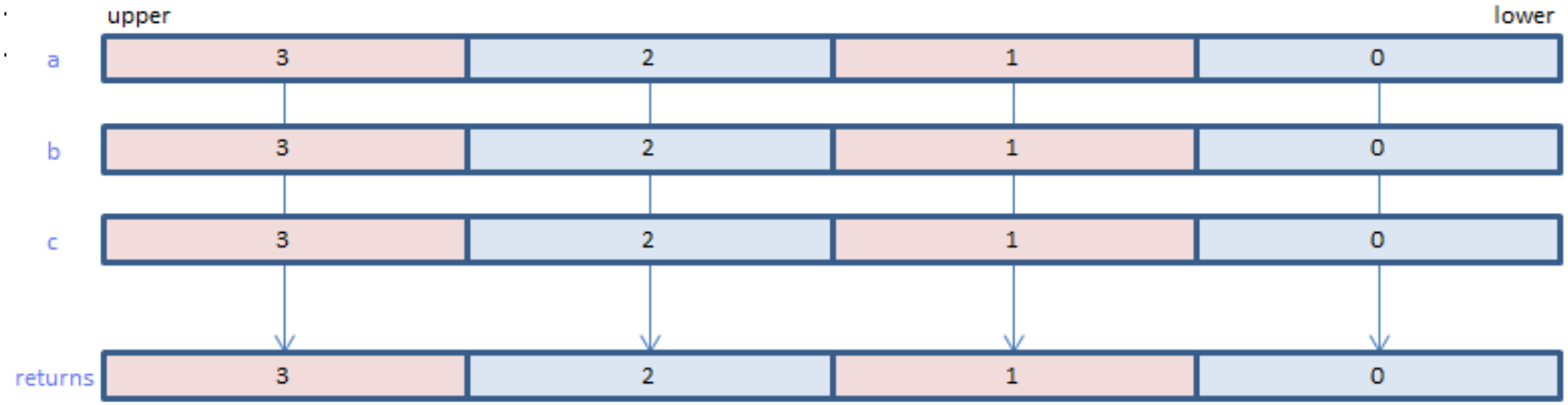


Calculate A*B+C*D and set to each WORD of (3). Signed saturation: set -32768 / 32767 on overflow.
Each BYTE of (1) is unsigned. Each BYTE of (2) is signed.

5. Non-SIMD: addsub

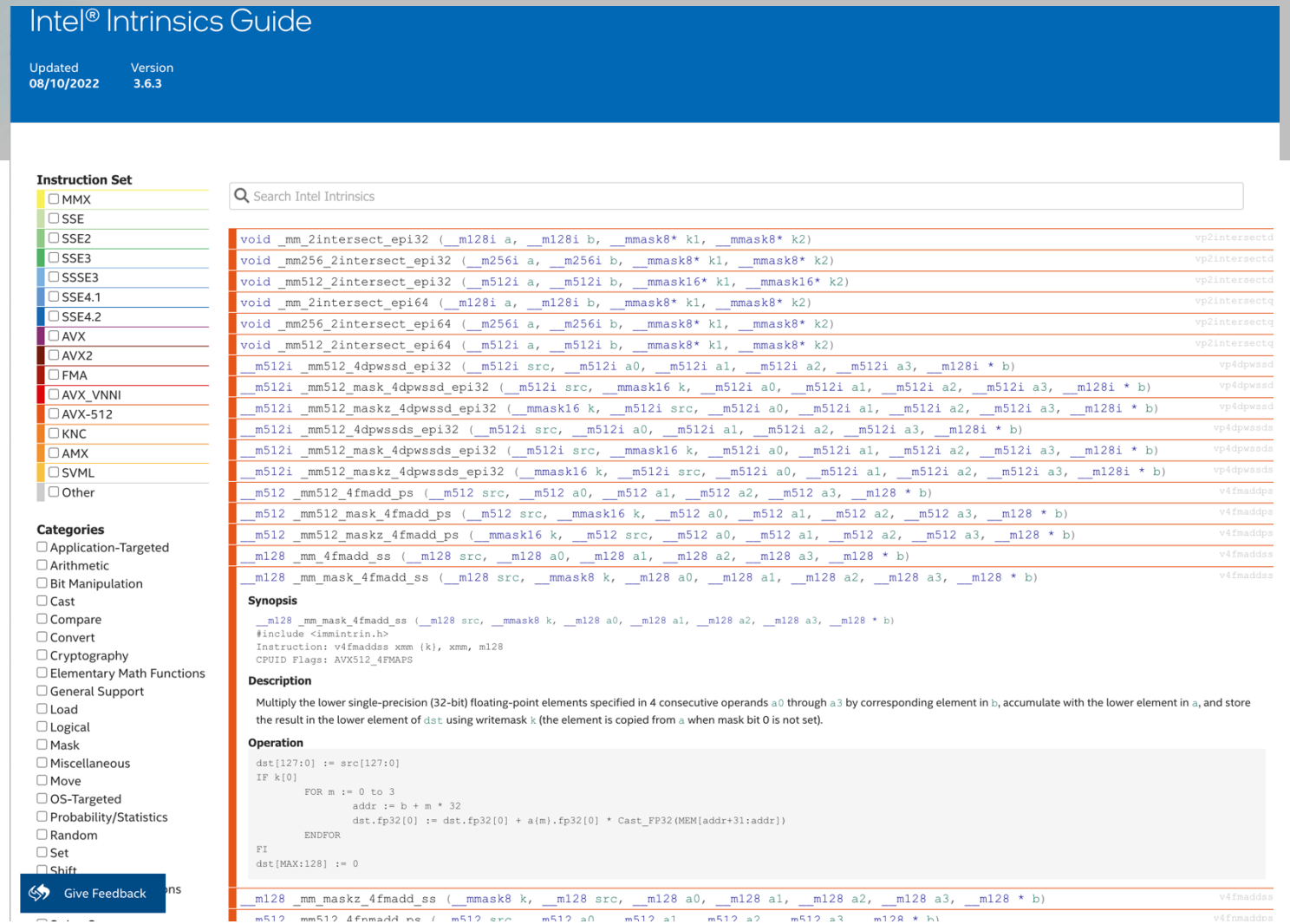
- Example

```
__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

GCC performs vectorization during its compile-time optimization phases—using -O2 and -O3 flags

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;
}
```

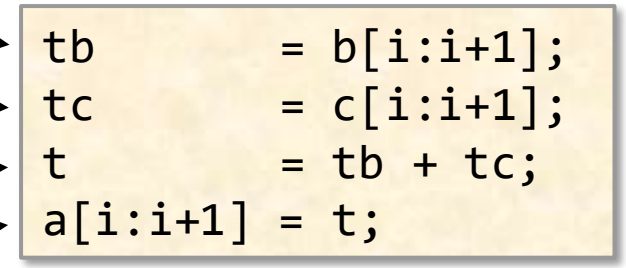
```
for (int i = 0; i < n; i += 2) {
  tb   = b[i];
  tb2  = b[i+1];
  tc   = c[i];
  tc2  = c[i+1];
  t    = tb + tc;
  t2   = tb2 + tc2;
  a[i] = t;
  a[i] = t2;
}
```

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

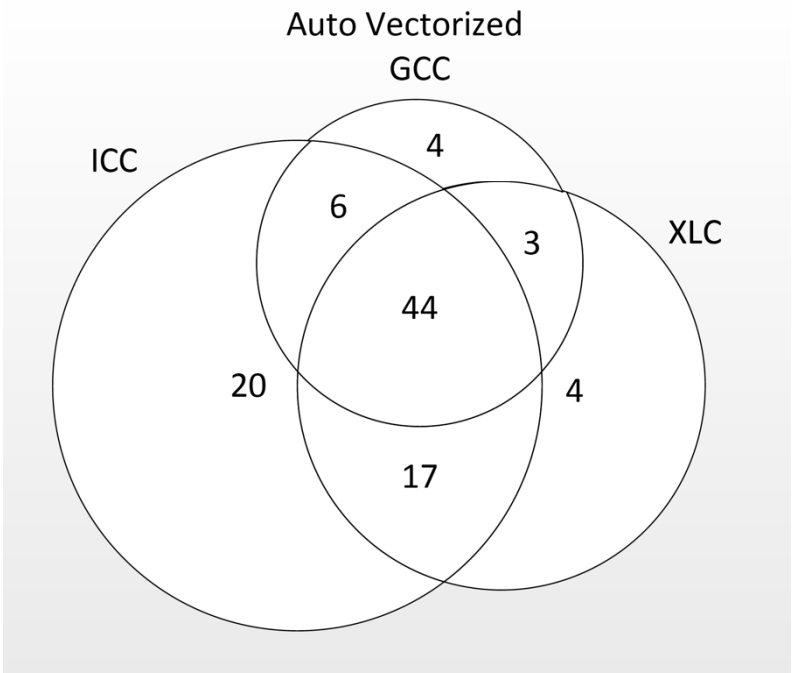
```
for (int i = 0; i < n; i++) {
  tb   = b[i];
  tc   = c[i];
  t    = tb + tc;
  a[i] = t;
}
```

```
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

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

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 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.

Vectorization in NLP

Example Application of Vectorization: NLP

- What Is Vectorization in NLP?
  - **Definition**: Converting words, sentences, or documents into numerical vectors
- Why do we need it?
  - Machine learning algorithms require numerical input
  - Enables measurement of similarity, semantic relatedness, etc.

Classic Approaches to Word Vectorization

- **One-Hot Encoding**
  - Large sparse vectors
  - High dimensionality (vocabulary size)
  - Lacks semantic relationships (no notion of distance except orthogonal vs. identical)

Vocabulary={"cat", "dog", "cow", "lion", "bird"}

Classic Approaches to Word Vectorization

- **One-Hot Encoding**

Vocabulary={"cat", "dog", "cow", "lion", "bird"}

- If we index them as:
  - **cat** → index 0
  - **dog** → index 1
  - **cow** → index 2
  - **lion** → index 3
  - **bird** → index 4

cat → [1, 0, 0, 0, 0]
dog → [0, 1, 0, 0, 0]
cow → [0, 0, 1, 0, 0]
lion → [0, 0, 0, 1, 0]
bird → [0, 0, 0, 0, 1]

Vectorization in NLP

- **Vectorization techniques**:
  - Bag of words: Extracts features from the text; calculate frequency of words in document
  - **TF-IDF: Information retrieval, keyword extraction**
  - Word2Vec: Semantic analysis task
  - GloVe: Word analogy, named entity recognition tasks
  - BERT: language translation, question answering system

TF-IDF Vectorization

Bag-of-Words (BoW)

- **Definition**:
  - A representation where each document (sentence, paragraph, or article) is described by the frequencies of each word (or token) it contains, irrespective of word order.
- Procedure:
  - Define a vocabulary of all unique words
  - For each document, build a numerical vector of word counts or frequencies
  - A **sparse vector** because most words do not appear in each document

Bag-of-Words (BoW)

- Procedure:
  - Define a vocabulary of all unique words
  - For each document, build a numerical vector of word counts or frequencies
  - A **sparse vector** because most words do not appear in each document
- **Limitations**:
  - Treats all words equally; doesn't differentiate between datasets
  - Doesn't capture word order or context

TF-IDF

- **Definition**: Term Frequency–Inverse Document Frequency
  - Weighting scheme that adjusts how much weight a word contributes based on **how frequently it appears in a specific document**
  - And **how rarely it appears** across all documents
- Why use this?
  - Words like *"the"*, *"a"*, *"and"* might be very frequent but carry little distinguishing power for a particular document
  - Uncommon or domain-specific words (e.g., *"hematology"*, *"tensor"*) are more indicative
  - Emphasizes **unique words**

TF-IDF Formula

- Term Frequency (TF)
  - The number of times a term t appears in a document d

$$tf(t,d) = \frac{count\ of\ t\ in\ d}{total\ number\ of\ terms\ in\ d}$$

- Inverse Document Frequency (IDF)
  - Downweight terms that appear in many documents, upweight rare terms

TF-IDF Formula

- Term Frequency (TF)

$$tf(t,d) = \frac{count\ of\ t\ in\ d}{total\ number\ of\ terms\ in\ d}$$

- Inverse Document Frequency (IDF)
  - Downweight terms that appear in many documents, upweight rare terms

$$idf(t,N) = \log\left(\frac{N}{1+df(t)}\right)$$

  - N := total number of documents
  - $df(t)$:= number of documents containing t

TF-IDF Formula

- TF-IDF Formula:

$$tf - idf(t, d, N) = tf(t, d) \cdot idf(t, N)$$

TF-IDF Formula

- TF-IDF Formula:

$$tf - idf(t, d, N) = tf(t, d) \cdot idf(t, N)$$

- A term will have a **high tf-idf** weight if:
  - It occurs many times in a single document; $tf(t, d)$ is high
  - It appears in relatively few documents; $idf(t, N)$ is high
- Conversely, words common to many documents (e.g., “the”, “and”) get a low tf-idf

TF-IDF Example

- Suppose you have 3 documents:
  - **D1**: “the cat sat on the mat”
  - **D2**: “the dog chased the cat”
  - **D3**: “the lion roared”

Term frequency for **D1**:

$$tf(\text{“}the\text{”}) \text{ ?}$$

$$\boldsymbol{tf(\text{“}the\text{”}) = \frac{2}{6}}$$

- Step 1: Vocabulary

{the, cat, sat, on, mat, dog, chased, lion, roared}

TF-IDF Example

- Suppose you have 3 documents:
  - **D1**: “the cat sat on the mat”
  - **D2**: “the dog chased the cat”
  - **D3**: “the lion roared”

IDF:

$$idf(\text{“}the\text{”}) \text{ ?}$$

- Step 1: Vocabulary

{the, cat, sat, on, mat, dog, chased, lion, roared}

TF-IDF Example

- Suppose you have 3 documents:
  - **D1**: “the cat sat on the mat”
  - **D2**: “the dog chased the cat”
  - **D3**: “the lion roared”

IDF:

$$idf(\text{“}the\text{”}) \; ?$$

$$\log\left(\frac{3}{4}\right) < 1$$

- Step 1: Vocabulary

{the, cat, sat, on, mat, dog, chased, lion, roared}

TF-IDF Example

- Suppose you have 3 documents:
  - **D1**: “the cat sat on the mat”
  - **D2**: “the dog chased the cat”
  - **D3**: “the lion roared”

Term frequency for **D3**:

$$tf(\text{“}lion\text{”}) \text{ ?}$$

- Step 1: Vocabulary

{the, cat, sat, on, mat, dog, chased, lion, roared}

TF-IDF Example

- Suppose you have 3 documents:
  - **D1**: “the cat sat on the mat”
  - **D2**: “the dog chased the cat”
  - **D3**: “the lion roared”

Term frequency for **D3**:

$$tf(\text{“}lion\text{”}) \text{ ?}$$

$$\boldsymbol{tf(\text{“}lion\text{”}) = \frac{1}{3}}$$

- Step 1: Vocabulary

{the, cat, sat, on, mat, dog, chased, lion, roared}

TF-IDF Example

- Suppose you have 3 documents:
  - **D1**: “the cat sat on the mat”
  - **D2**: “the dog chased the cat”
  - **D3**: “the lion roared”

IDF:

$$idf(\text{“}lion\text{”}) \; ?$$

$$\log\left(\frac{3}{2}\right) > 1$$

- Step 1: Vocabulary

{the, cat, sat, on, mat, dog, chased, lion, roared}

TF-IDF Example

- Suppose you have 3 documents:
  - **D1**: “the cat sat on the mat”
  - **D2**: “the dog chased the cat”
  - **D3**: “the lion roared”

TF-IDF:

$$tf - idf(\text{“}lion\text{”}) \text{ ?}$$

- Step 1: Vocabulary

{the, cat, sat, on, mat, dog, chased, lion, roared}

TF-IDF Example

- Suppose you have 3 documents:
  - **D1**: “the cat sat on the mat”
  - **D2**: “the dog chased the cat”
  - **D3**: “the lion roared”

TF-IDF:

$$tf - idf(\text{“}lion\text{”}) \text{ ?}$$

$$\frac{1}{3} \cdot \log\left(\frac{3}{2}\right)$$

- Step 1: Vocabulary

{the, cat, sat, on, mat, dog, chased, lion, roared}

TF-IDF Example

- Suppose you have 3 documents:
  - **D1**: “the cat sat on the mat”
  - **D2**: “the dog chased the cat”
  - **D3**: “the lion roared”

- Step 1: Vocabulary

  {the, cat, sat, on, mat, dog, chased, lion, roared}

- Final step:

TF-IDF Example

- Suppose you have 3 documents:
  - **D1**: “the cat sat on the mat”
  - **D2**: “the dog chased the cat”
  - **D3**: “the lion roared”

- Step 1: Vocabulary

  {the, cat, sat, on, mat, dog, chased, lion, roared}

- Final step:
  - Matrix of numbers: each row corresponds to a document

TF-IDF Example

- Suppose you have 3 documents:
  - **D1**: “the cat sat on the mat”
  - **D2**: “the dog chased the cat”
  - **D3**: “the lion roared”

- Step 1: Vocabulary

  {the, cat, sat, on, mat, dog, chased, lion, roared}

- Final step:
  - Matrix of numbers: each row corresponds to a document
  - Each column is a vocabulary term

TF-IDF Example

- Suppose you have 3 documents:
  - **D1**: “the cat sat on the mat”
  - **D2**: “the dog chased the cat”
  - **D3**: “the lion roared”

- Step 1: Vocabulary

  {the, cat, sat, on, mat, dog, chased, lion, roared}

- Final step:
  - Matrix of numbers: each row corresponds to a document
  - Each column is a vocabulary term
  - Each value is a TF-IDF number

TF-IDF Afterwards

- TF-IDF matrix: sparse but large
  - **Dimensionality reduction**: reduce the dimension of the matrix via methods like Singular Value Decomposition (SVD), PCA, or Truncated SVD
- Passed into ML Models
  - Classification (Logistic Regression, Naïve Bayes, SVM)
  - Clustering (kNN, k-Means, Hierarchical Clustering)
  - Similarity-Based Retrieval Systems (Recommendation systems)