

# Single Instruction Multiple Data (SIMD) processing



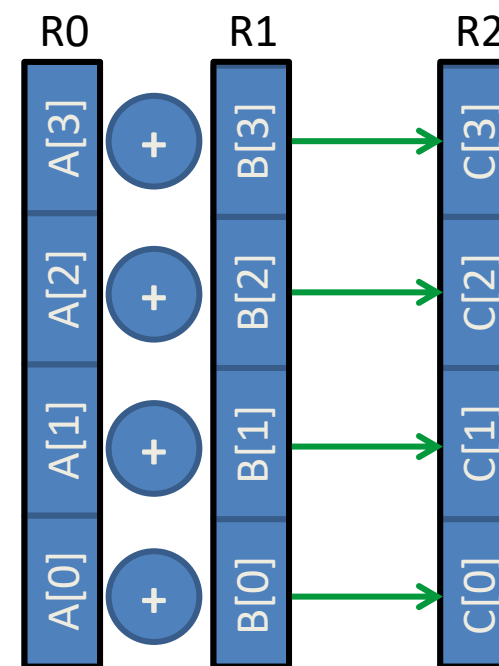
# SIMD terminology

## A word on terminology

- SIMD == “one instruction → several operations”
- “SIMD width” == number of operands that fit into a register
- No statement about parallelism among those operations
- Original vector computers: long registers, pipelined execution, but no parallelism (within the instruction)

## Today

- x86: most SIMD instructions fully parallel
  - “Short Vector SIMD”
  - Some exceptions on some archs (e.g., vdivpd)
- NEC Tsubasa: 32-way parallelism but SIMD width = 256 (DP)



# Scalar execution units

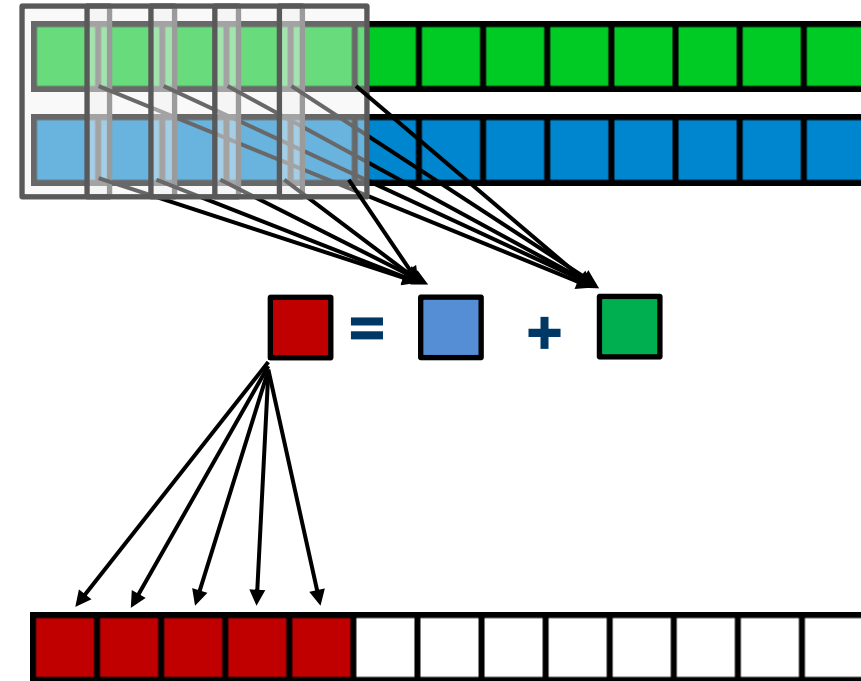
```
for (int j=0; j<size; j++){  
    A[j] = B[j] + C[j];  
}
```

## Register widths

- 1 operand



## Scalar execution



# Data-parallel execution units (short vector SIMD)

```
for (int j=0; j<size; j++){  
    A[j] = B[j] + C[j];  
}
```

## Register widths

- 1 operand



- 2 operands (SSE)



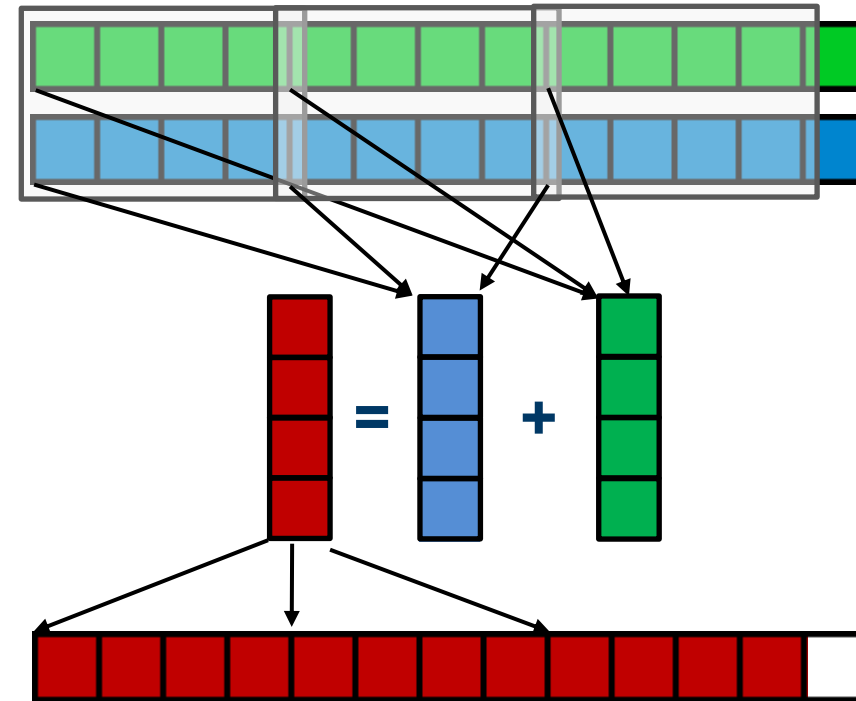
- 4 operands (AVX)



- 8 operands (AVX512)



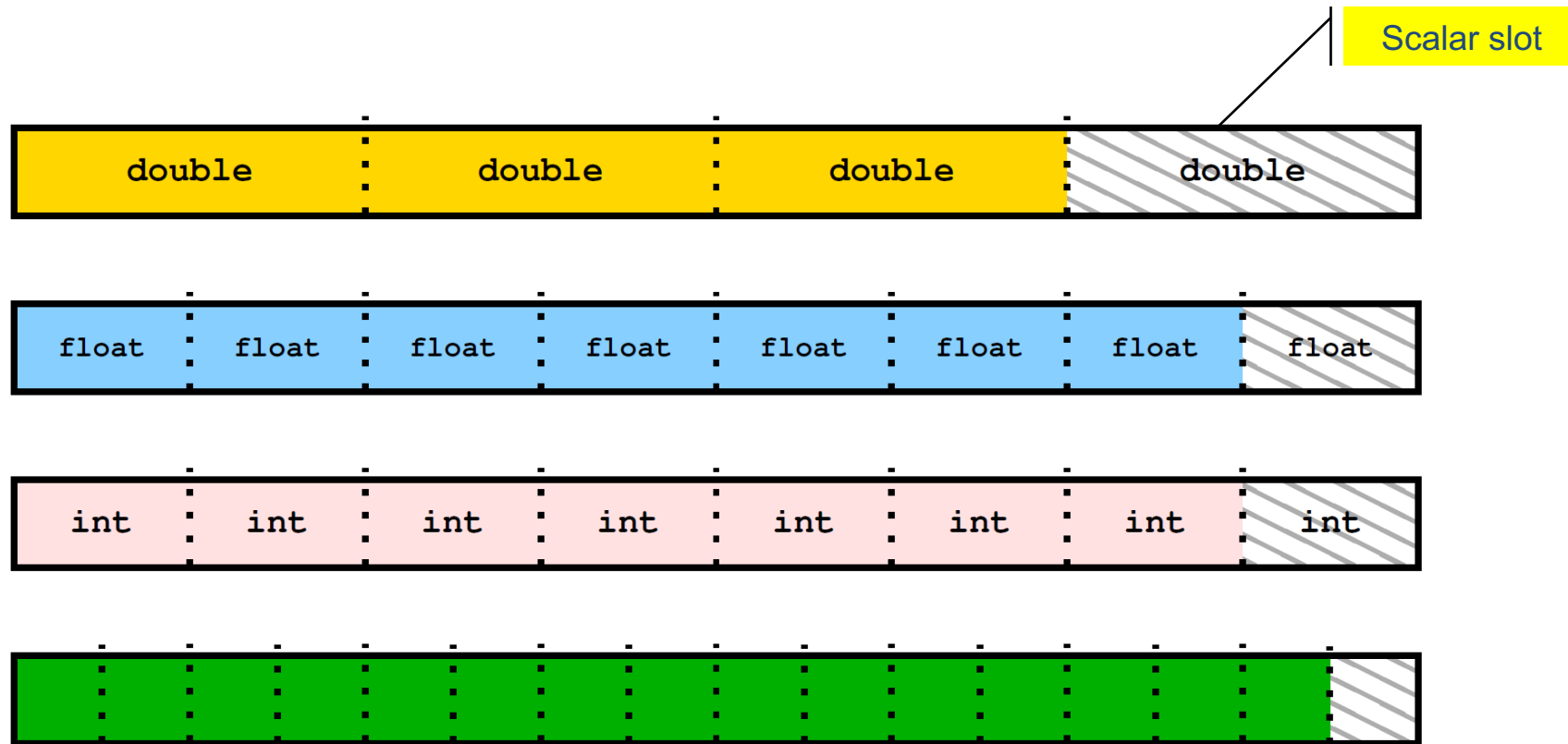
## SIMD execution



Best code requires vectorized  
LOADs, STOREs, and arithmetic!

# Data types in 32-byte SIMD registers

Supported data types depend on actual SIMD instruction set



# SIMD

## The Basics



# SIMD processing – Basics

Steps (done by the compiler) for “SIMD processing”

```
for(int i=0; i<n;i++)  
    C[i]=A[i]+B[i];
```

“Loop unrolling”

```
for(int i=0; i<n;i+=4){  
    C[i]  =A[i]  +B[i];  
    C[i+1]=A[i+1]+B[i+1];  
    C[i+2]=A[i+2]+B[i+2];  
    C[i+3]=A[i+3]+B[i+3];}  
//remainder loop handling
```

This  
should  
not be  
done  
by  
hand!



Load 256 Bits starting from address of A[i] to register R0

Add the corresponding 64 Bit entries in R0 and R1 and store the 4 results to R2

Store R2 (256 Bit) to address starting at C[i]

```
LABEL1:  
VLOAD R0 ← A[i]  
VLOAD R1 ← B[i]  
V64ADD[R0,R1] → R2  
VSTORE R2 → C[i]  
i ← i+4  
i < (n-4)? JMP LABEL1  
//remainder loop handling
```

# SIMD processing: roadblocks

No SIMD vectorization for loops with data dependencies:

```
for(int i=1; i<n; i++)  
    A[i] = A[i-1] * s;
```

“**Pointer aliasing**” may prevent SIMDfication

```
void f(double *A, double *B, double *C, int n) {  
    for(int i=0; i<n; ++i)  
        C[i] = A[i] + B[i];  
}
```

C/C++ allows that  $A \rightarrow \&C[-1]$  and  $B \rightarrow \&C[-2]$

$\rightarrow C[i] = C[i-1] + C[i-2]$ : dependency  $\rightarrow$  No SIMD

If “**pointer aliasing**” is not used, tell the compiler:

**-fno-alias** (Intel), **-Msafeptr** (PGI), **-fargument-noalias** (gcc)

**restrict** keyword (C only!):

```
void f(double *restrict A, double *restrict B, double *restrict C, int n) {...}
```

# How to leverage SIMD: your options

## Options:

- The **compiler** does it for you  
(but: aliasing, alignment, language, abstractions)
- Compiler directives (**pragmas**)
- Alternative **programming models** for compute kernels (OpenMP 4, OpenCL, ispc)
- **Intrinsics** (restricted to C/C++)
- Implement directly in **assembly language**

## Example: x86 SIMD (SSE) intrinsics

```
#include <x86intrin.h>

...
for (int j=0; j<size; j+=16){
    t0 = _mm_loadu_ps(data+j);
    t1 = _mm_loadu_ps(data+j+4);
    t2 = _mm_loadu_ps(data+j+8);
    t3 = _mm_loadu_ps(data+j+12);
    sum0 = _mm_add_ps(sum0, t0);
    sum1 = _mm_add_ps(sum1, t1);
    sum2 = _mm_add_ps(sum2, t2);
    sum3 = _mm_add_ps(sum3, t3);
}
```

# Vectorization compiler options (Intel)

- The compiler will vectorize starting with `-O2`.
- To enable specific SIMD extensions use the `-x` option:

- `-xSSE2` vectorize for SSE2 capable machines

Available SIMD extensions:

`SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, ...`

- `-xAVX` on Sandy/Ivy Bridge processors
- `-xCORE-AVX2` on Haswell/Broadwell
- `-xCORE-AVX512` on Skylake/CascadeLake

Recommended for AMD Zen:  
`-O3 -mavx2`

Recommended option:

- `-xHost` will optimize for the Intel architecture you compile on
- To really enable 512-bit SIMD with current Intel compilers you need to set:  
`-qopt-zmm-usage=high`

# User-mandated vectorization (OpenMP 4)

- Since OpenMP 4.0, SIMD features are a part of the OpenMP standard
- `#pragma omp simd` enforces vectorization
- Essentially a standardized “go ahead, no dependencies here!”
  - **Do not lie** to the compiler here!
- Prerequisites:
  - Countable loop
  - Innermost loop
  - Must conform to for-loop style of OpenMP worksharing constructs
- There are additional clauses:  
`reduction, simdlen, private, collapse, ...`

```
for (int j=0; j<n; j++) {  
    #pragma omp simd reduction(+:b[j])  
    for (int i=0; i<n; i++) {  
        b[j] += a[j][i];  
    }  
}
```

# OpenMP SIMD support

## Additional specifications enable better SIMD vectorization

```
double precision :: t,sum
integer :: i,j
! ...
j = 1
!$OMP SIMD REDUCTION(+:sum) LINEAR(j:2)
do i = 1,N
    sum = sum + a(i)*a(i)
    j = j+2
enddo
!$OMP END SIMD
```

j has linear relationship  
with loop counter in SIMD  
direction

## SIMD clause can be combined with OpenMP work sharing

```
!$OMP DO SIMD SCHEDULE(SIMD:STATIC,c)
do i = 1,N
    a(i) = exp(b(i))
enddo
!$OMP END DO SIMD
```

Compiler will use  
SIMD version of  
function if present

Extend chunk  
size to next  
SIMD width  
multiple

# OpenMP SIMD support

Functions and subroutines can be declared as SIMD vectorizable and called from SIMD loops

```
double precision function hyp3d(a,b,c) result(h)
```

```
!$OMP DECLARE SIMD
```

```
double precision :: a,b,c
```

```
h = sqrt(a*a + b*b + c*c)
```

```
end function
```

```
[...]
```

Makes compiler  
generate SIMD  
version(s) of the  
function

C version:

```
#pragma omp declare simd
double hyp3d(double a,
             double b,
             double c) {
    return sqrt(a*a + b*b + c*c);
}
```

```
double precision, dimension(N) :: a,b,c,hyp
```

```
!$OMP PARALLEL DO SIMD
```

```
do i = 1,N
```

```
hyp(i) = hyp3d(a(i),b(i),c(i))
```

```
enddo
```

```
!$OMP END PARALLEL DO SIMD
```

SIMD loop calls  
SIMD version of  
function

# SIMD

Reading Assembly Language  
(Don't Panic)



# Assembly: Why and how?

## Why check the assembly code?

Sometimes the only way to make sure the compiler “did the right thing”

Example: “LOOP WAS VECTORIZED” message is printed, but an unnecessary dependency chain limits the performance!

## Get the assembly code from the compiler (Intel/GCC/clang):

```
icc -S -O3 -xHost triad.c -o triad.s
```

## Disassemble a binary:

```
objdump -d ./a.out | less
```

The x86 ISA is documented in:

Intel Software Development Manual (SDM) 2A and 2B  
AMD64 Architecture Programmer's Manual Vol. 1-5

# Basics of the x86-64 ISA

- Instructions have 0 to 3 operands (4 with AVX-512)
- Operands can be registers, memory references or immediates
- Opcodes (binary representation of instructions) vary from 1 to 15 (?) bytes
- There are two assembler syntax forms: **Intel (left)** and **AT&T (right)**
- Addressing Mode:  $\text{BASE} + \text{INDEX} * \text{SCALE} + \text{DISPLACEMENT}$
- C:  $\mathbf{A[i]}$  equivalent to  $\mathbf{* (A+i)}$  (a pointer has a type:  $\mathbf{A+i*8}$ )

## Intel syntax

```
movaps [rdi + rax*8+48], xmm3
add rax, 8
js 1b
```

```
401b9f: 0f 29 5c c7 30
401ba4: 48 83 c0 08
401ba8: 78 a6
```

## AT&T syntax

```
movaps    %xmm3, 48(%rdi,%rax,8)
addq      $8, %rax
js        ..B1.4
```

```
movaps    %xmm3,0x30(%rdi,%rax,8)
addq      $0x8,%rax
js        401b50 <triad_asm+0x4b>
```

# Basics of the x86-64 ISA with extensions

16 general purpose registers (64bit):

`rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8-r15`

alias with eight 32 bit register set:

`eax, ebx, ecx, edx, esi, edi, esp, ebp`

8 opmask registers (16 bit or 64 bit, AVX512 only):

`k0-k7`

Floating Point SIMD registers (aliased):

`xmm0-xmm15 (...xmm31)` SSE (128bit)

`ymm0-ymm15 (...xmm31)` AVX (256bit)

`zmm0-zmm31` AVX-512 (512bit)

SIMD instructions are distinguished by:

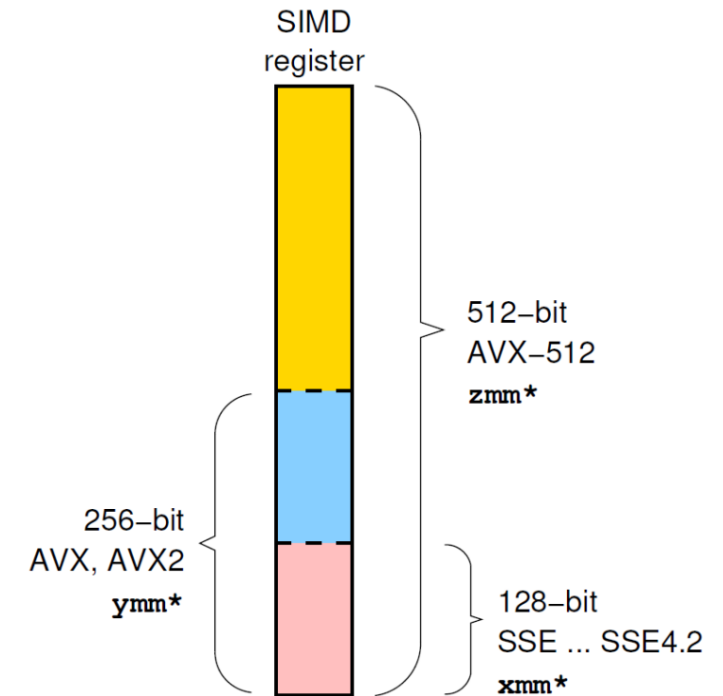
VEX/EVEX prefix: `v`

Operation: `mul, add, mov`

Modifier: nontemporal (`nt`), unaligned (`u`), aligned (`a`), high (`h`)

Width: scalar (`s`), packed (`p`)

Data type: single (`s`), double (`d`)



# Case Study: Sum reduction (DP)

```
double sum = 0.0;

for (int i=0; i<size; i++) {
    sum += data[i];
}
```

To get object code use  
`objdump -d` on object file or  
executable or compile with `-S`

Assembly code w/ `-O1` (Intel syntax, Intel compiler):

```
.label:
    addsd    xmm0, [rdi + rax * 8]
    inc      rax
    cmp      rax, rsi
    jl       .label
```

AT&T syntax:  
`addsd 0(%rdi,%rax,8),%xmm0`

# Sum reduction (DP) – AVX512

Assembly code w/ `-O3 -xCORE-AVX512 -qopt-zmm-usage=high` :

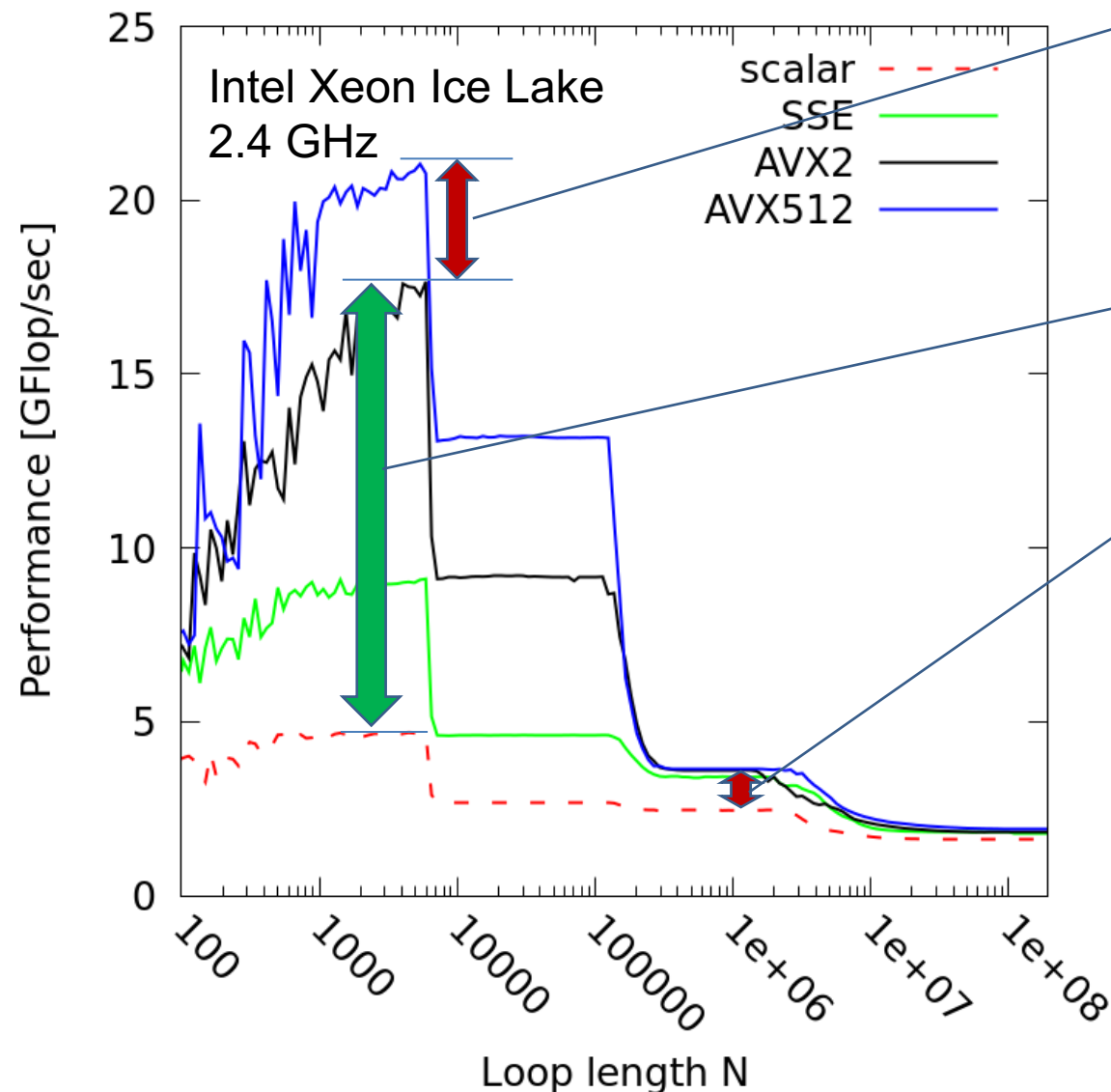
```
.label:
    vaddpd    zmm1, zmm1, [rdi+rcx*8]
    vaddpd    zmm4, zmm4, [64+rdi+rcx*8]
    vaddpd    zmm3, zmm3, [128+rdi+rcx*8]
    vaddpd    zmm2, zmm2, [192+rdi+rcx*8]
    add       rcx, 32
    cmp       rcx, rdx
    jb        .label
;
    vaddpd    zmm1, zmm1, zmm4
    vaddpd    zmm2, zmm3, zmm2
    vaddpd    zmm1, zmm1, zmm2
; [... SNIP ...]
    vshuff32x4 zmm2, zmm1, zmm1, 238
    vaddpd    zmm1, zmm2, zmm1
    vpermpd   zmm3, zmm1, 78
    vaddpd    zmm4, zmm1, zmm3
    vpermpd   zmm5, zmm4, 177
    vaddpd    zmm6, zmm4, zmm5
    vaddsd    xmm0, xmm6, xmm0
```

Bulk loop code  
(8x4-way unrolled)

Remainder omitted

Sum up 32  
partial sums into  
`xmm0.0`

# Sum reduction (DP) – sequential performance



ICX can only do  
1.1 AVX-512  
Loads per cy

Expected speedup  
in L1 with  
scalar → AVX2

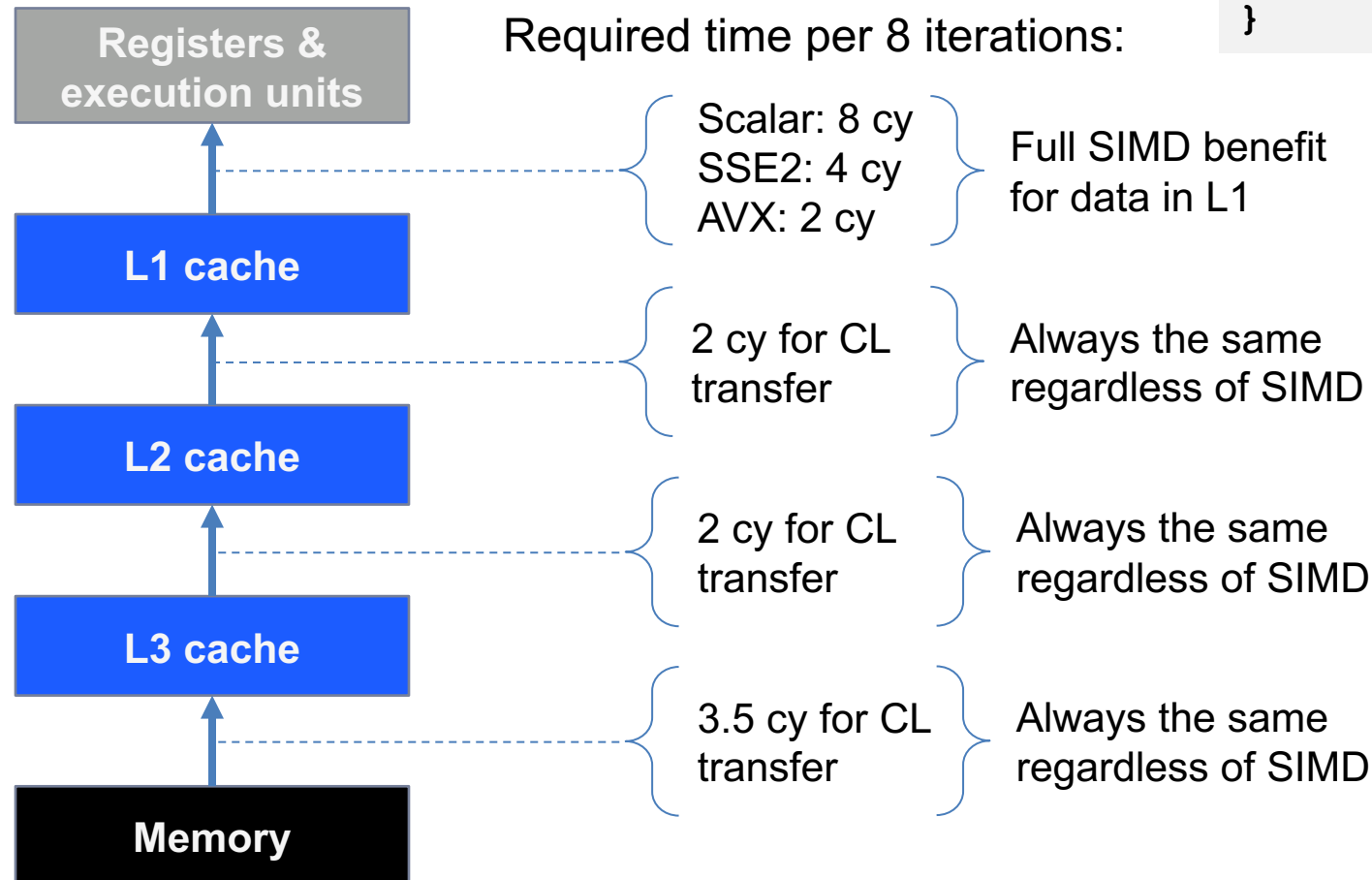
Limited speedup in  
L3 and memory

- SIMD is an **in-core** performance feature!
- If the bottleneck is data transfer beyond L1, its benefit is limited
- Other in-core bottlenecks can reduce the expected speedup from SIMD

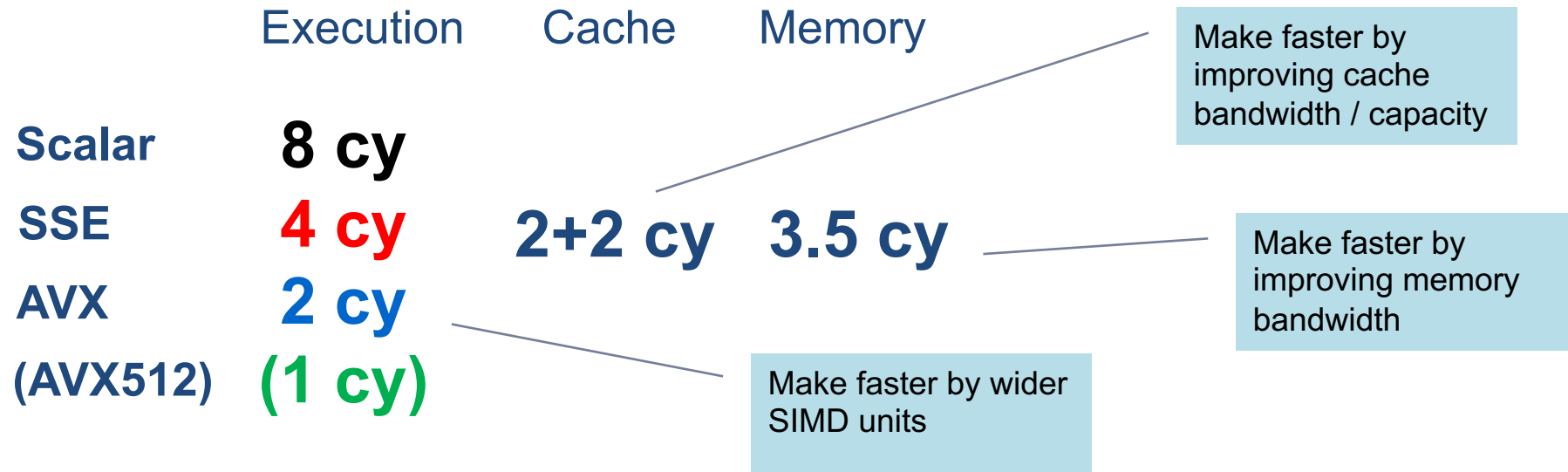
# Limits of the SIMD benefit

- Why does SIMD usually not give the expected speedup? □ Analyze time contributions with data from memory (DP sum reduction on Ivy Bridge)

```
for (int i=0; i<size; i++){  
    sum += data[i];  
}
```



# Limits of SIMD processing



On Intel x86 processors, these contributions must be added to get the runtime:

|        | L1 [cy] | L2 [cy] | L3 [cy] | Memory [cy] | Sum [cy] |
|--------|---------|---------|---------|-------------|----------|
| Scalar | 8       | 2       | 2       | 3.5         | 15.5     |
| SSE2   | 4       | 2       | 2       | 3.5         | 11.5     |
| AVX    | 2       | 2       | 2       | 3.5         | 9.5      |

diminishing returns  
(Amdahl's Law!)

# SIMD

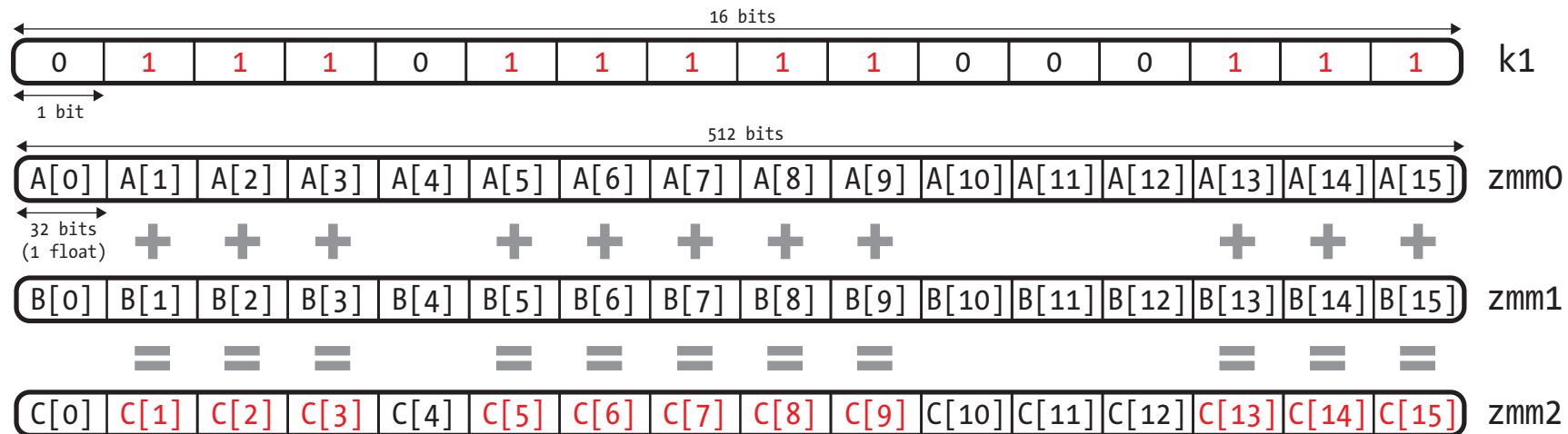
Masked SIMD execution and gather instructions



# Example for masked execution

**Masking** is very helpful in cases such as, e.g., conditionals or remainder loop handling

Example: `vaddps zmm2{k1}, zmm1, zmm0`



Available on x86 since AVX-512

# SIMD with masking: conditional sum reduction

```
double sum = 0.0;

for (int i=0; i<size; i++){
    if(data[i]>0.0)
        sum += data[i];
}
```

Bulk loop code  
(8x4-way unrolled)

.label:

```
vmovups    zmm5, [r12+rsi*8]
vmovups    zmm6, [r12+rsi*8+64]
vmovups    zmm7, [r12+rsi*8+128]
vmovups    zmm8, [r12+rsi*8+192]
vcmpgtpd   k1, zmm5, zmm4
vcmpgtpd   k2, zmm6, zmm4
vcmpgtpd   k3, zmm7, zmm4
vcmpgtpd   k4, zmm8, zmm4
vaddpd     zmm0{k1}, zmm0, zmm5
vaddpd     zmm3{k2}, zmm3, zmm6
vaddpd     zmm2{k3}, zmm2, zmm7
vaddpd     zmm1{k4}, zmm1, zmm8
add        rsi, 32
cmp        rsi, rdx
jb         .label
```

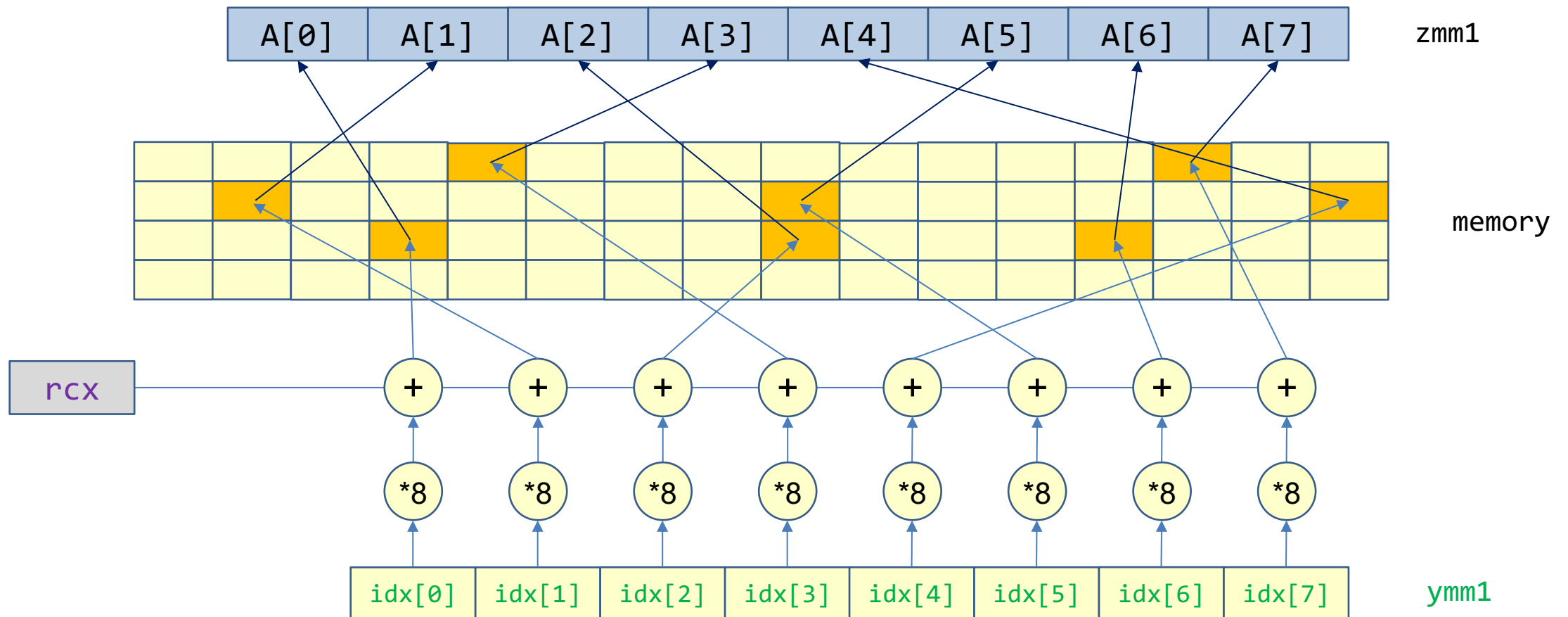
SIMD mask  
generation

masked SIMD  
ADDs  
(accumulates)

# SIMD gather instructions

**Gather** instructions load scattered elements into a SIMD register using a short vector of indices

Example: `vgatherdpd zmm1, [rcx+ymm1*8]`



# SIMD gather instructions: sparse matrix-vector multiplication

```
for (int row = 0; row < num_rows; ++row) {  
    double sum=0.0;  
    for (int j = row_ptr[row]; j < row_ptr[row + 1]; ++j)  
        sum += values[j] * x[col_idx[j]];  
    y[row] = sum;  
}
```

- Gather instructions can be subject to masking
- Data transfers are still managed through the cache hierarchy
  - Cache line concept applies

```
.label:  
    vpcmpgtq    k2, zmm3, zmm4  
    vpaddq     zmm4, zmm4, zmm1  
    vmovdqu32  ymm6{k2}{z}, [r12+r14*4]  
    vmovaps    zmm7, zmm2  
    kmovw     k1, k2  
    vgatherdpd zmm7{k1}, [rcx+ymm6*8]  
    vfmadd231pd zmm5{k2}, zmm7, [rdi+r14*8]  
    add       r14, 8  
    cmp       r14, rbx  
    jnb       .label
```

Remainders  
handled via  
masking

Index load

(no MVE for brevity)

# Rules and guidelines for vectorizable loops

1. **Inner** loop
2. **Countable** (loop length can be determined at loop entry)
3. Single entry and single exit
4. **Straight line** code (no conditionals) – unless masks can be used
5. No function calls (except intrinsic math functions and SIMD functions)

Better performance with:

1. Simple inner loops with unit stride (**contiguous** data access)
2. **Minimize indirect addressing**
3. Align data structures to SIMD width boundary (minor impact)

In C use the **restrict** keyword and/or **const** qualifiers and/or compiler options to rule out array/pointer aliasing

**Keep it simple, stupid!**