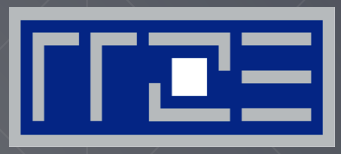


# Single Instruction Multiple Data (SIMD)

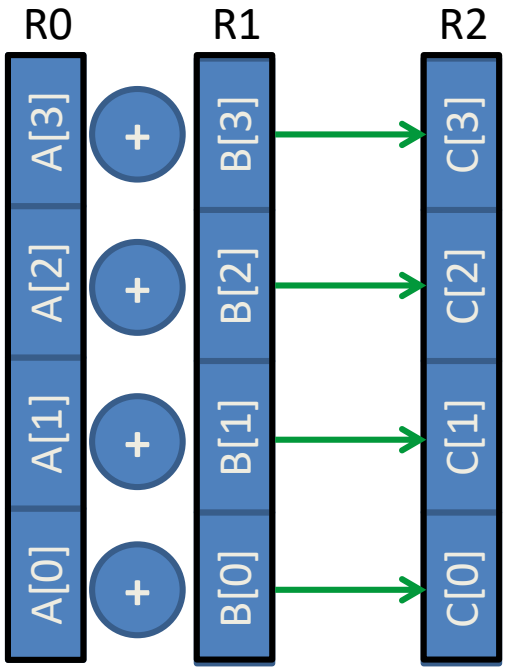
# SIMD terminology

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

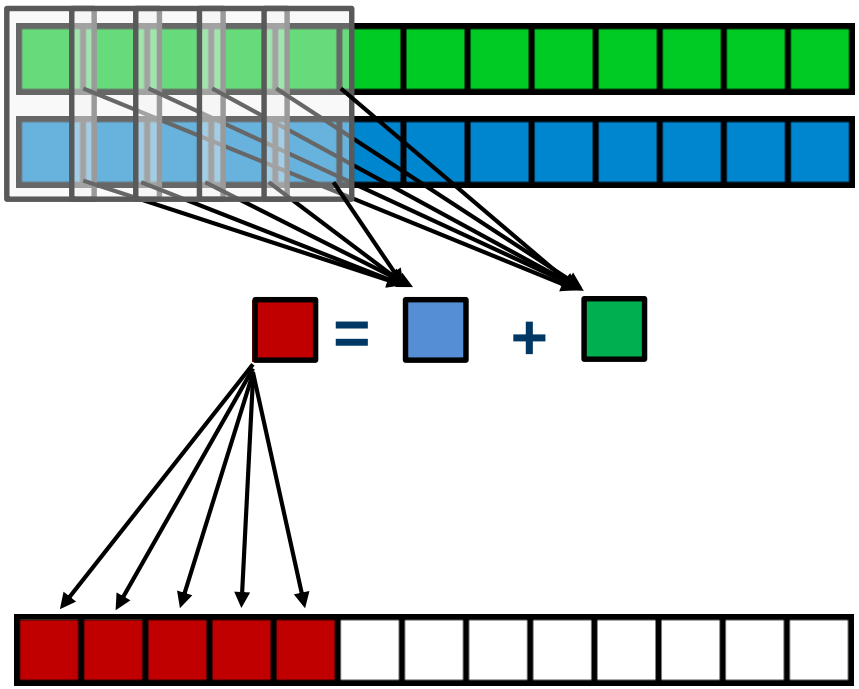
- 1 operand

- 2 operands (SSE)

- 4 operands (AVX)

- 8 operands (AVX512)

## Scalar execution



# Data-parallel execution units (short vector SIMD)

*Single Instruction Multiple Data (SIMD)*

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

## Register widths

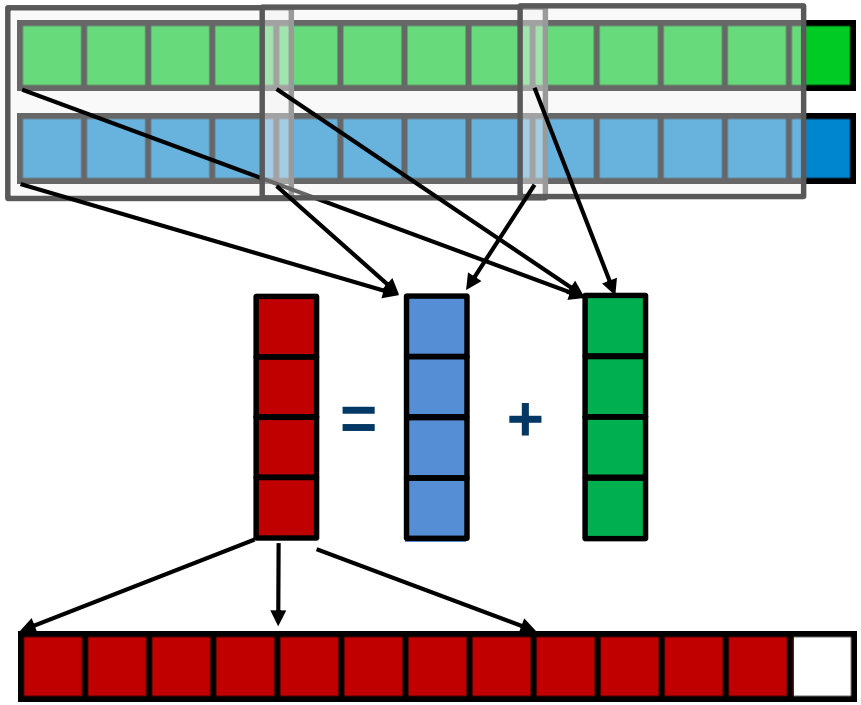
- 1 operand

- 2 operands (SSE)

- 4 operands (AVX)

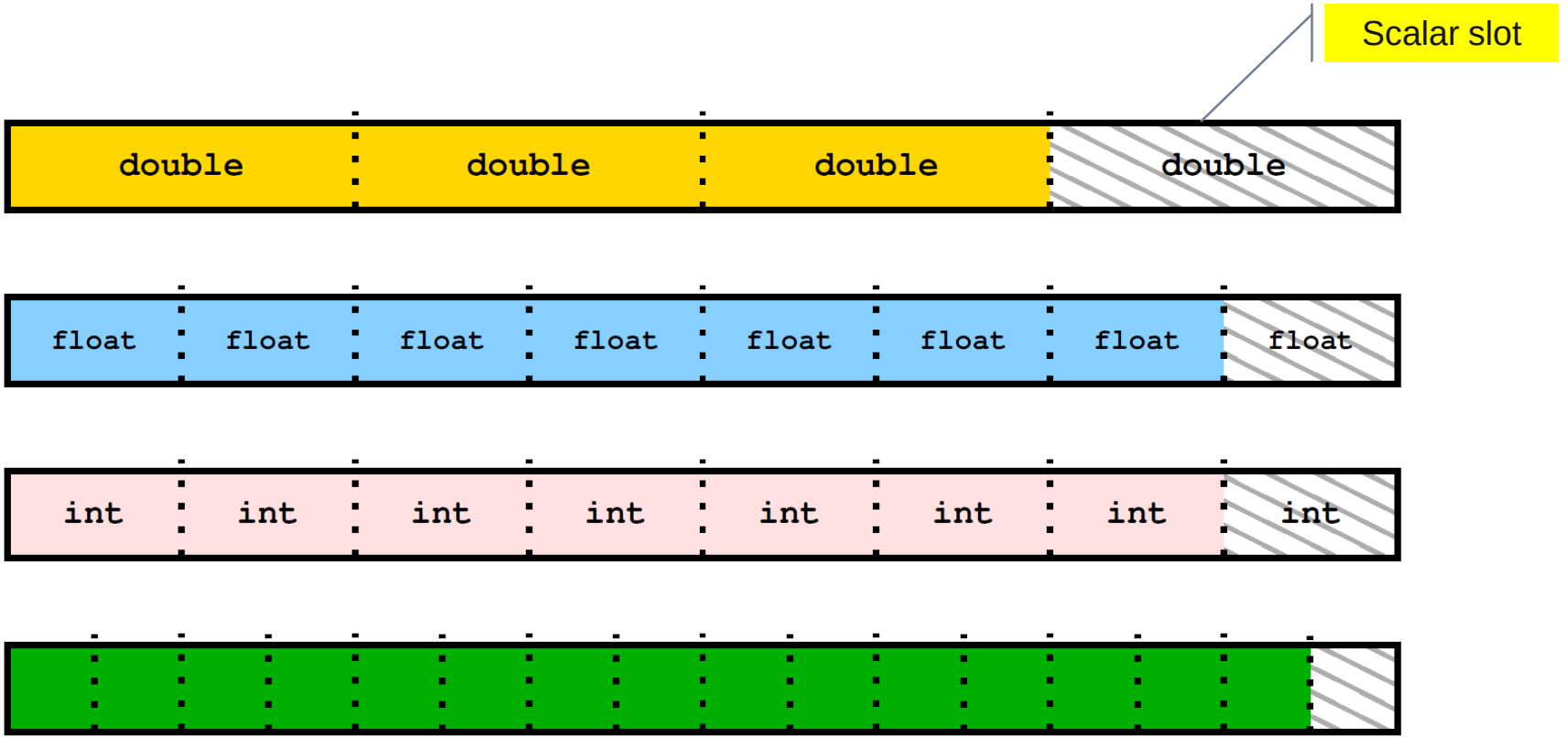
- 8 operands (AVX512)

## SIMD execution

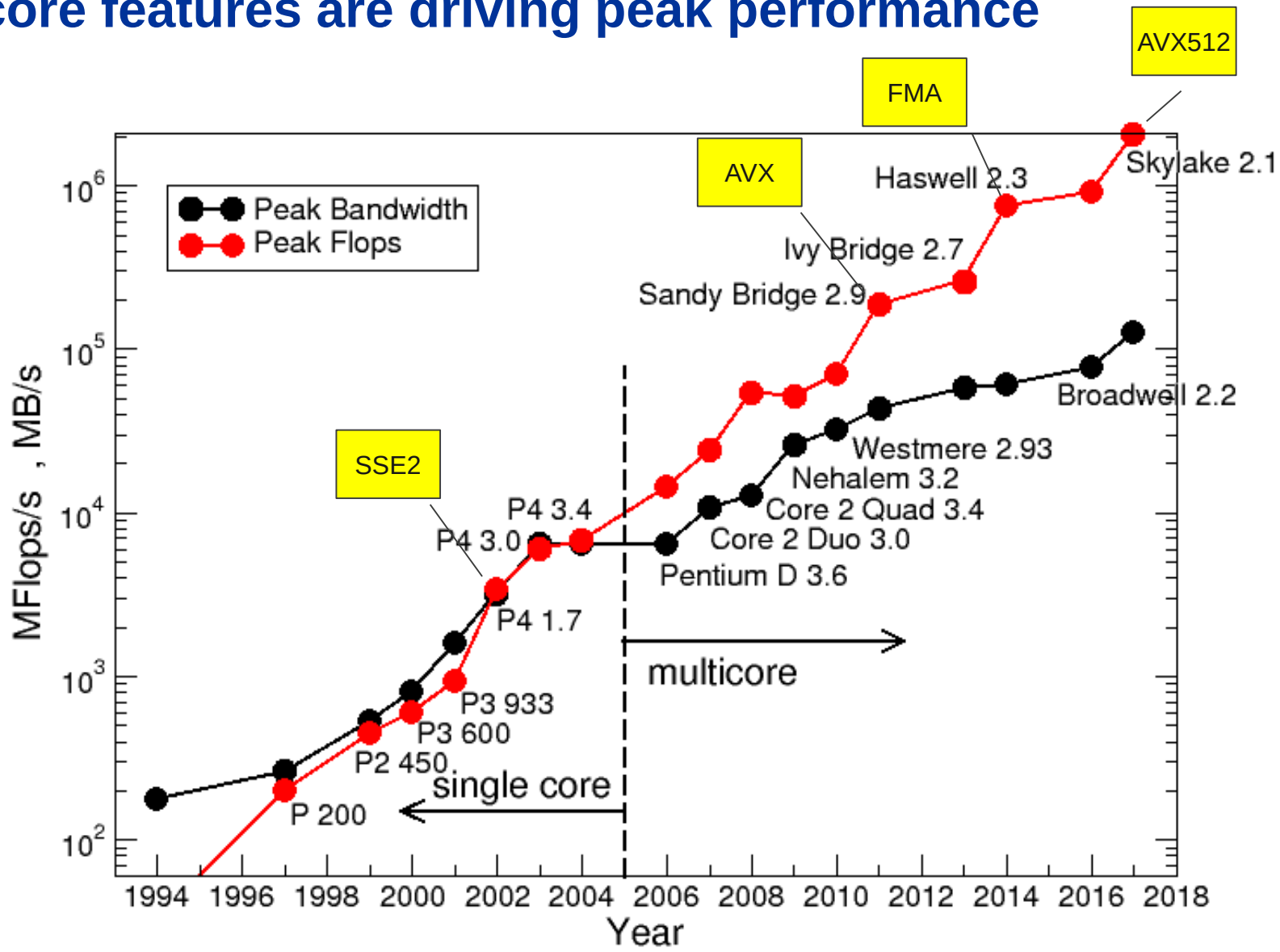


# Example: Data types in 32-byte SIMD registers (AVX[2])

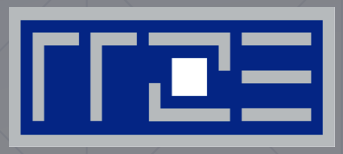
- Supported data types depend on actual SIMD instruction set



# In-core features are driving peak performance



# 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"

This should not be done by hand!

```
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
```

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 – Basics

No SIMD vectorization for loops with data dependencies:

```
for(int i=0; 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** → **&C[-1]** and **B** → **&C[-2]**

→ **C[i] = C[i-1] + C[i-2]:** dependency → 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 (OpenCL, ispc)
- Intrinsics (restricted to C/C++)
- Implement directly in assembler

To use intrinsics the following headers are available:

- **xmmintrin.h (SSE)**
- **pmmintrin.h (SSE2)**
- **immintrin.h (AVX)**
- **x86intrin.h (all extensions)**

```
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 (certain models)
  - **-xMIC-AVX512** on Xeon Phi Knights Landing

  Recommended option:
  - **-xHost** will optimize for the architecture you compile on (Caveat: do not use on standalone KNL, use MIC-AVX512)
  - 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!

- Prerequesites:
  - Countable loop
  - Innermost loop
  - Must conform to for-loop style of OpenMP worksharing constructs
- There are additional clauses:

**`reduction, vectorlength, private, collapse, ...`**

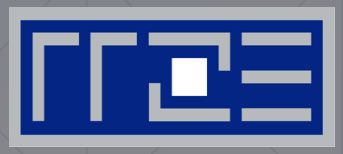
```
for (int j=0; j<n; j++) {
  #pragma omp simd reduction(+:b[j:1])
  for (int i=0; i<n; i++) {
     b[j] += a[j][i];
  }
}
```

# x86 Architecture:
## *SIMD and Alignment*

- Alignment issues
  - Alignment of arrays should optimally be on SIMD-width address boundaries to allow packed aligned loads (and NT stores on x86)
  - Otherwise the compiler will revert to unaligned loads/stores
  - Modern x86 CPUs have less (not zero) impact for misaligned LOAD/STORE, but **Xeon Phi KNC relies heavily on it!**
  - How is manual alignment accomplished?

- Stack variables: **`alignas`** keyword (C++11/C11)
- Dynamic allocation of aligned memory (**`align`** = alignment boundary)
  - C before C11 and C++ before C++17:
    ```
    posix_memalign(void **ptr, size_t align, size_t size);
    ```
  - C11 and C++17:
    ```
    aligned_alloc(size_t align, size_t size);
    ```

# SIMD



## Reading Assembly Language

# Assembler: 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 Loads & Stores may still be scalar!
- Get the assembler code (Intel compiler):

```
icc -S -O3 -xHost triad.c -o a.out
```

- Disassemble Executable:

```
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: BASE + INDEX * SCALE + DISPLACEMENT
- C: **`A[i]`** equivalent to **`*(A+i)`** (a pointer has a type: **`A+i*8`**)

Intel syntax

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

AT&T syntax

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

```
401b9f: 0f 29 5c c7 30      movaps %xmm3,0x30(%rdi,%rax,8)
401ba4: 48 83 c0 08         addq   $0x8,%rax
401ba8: 78 a6               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 (16bit or 64bit, AVX512 only):

**`k0-k7`**

Floating Point **SIMD** Registers:

| | |
|---|---|
| **`xmm0-xmm15 (xmm31)`** | SSE (128bit) alias with 256-bit and 512-bit registers |
| **`ymm0-ymm15 (xmm31)`** | AVX (256bit) alias with 512-bit registers |
| **`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`**) |

# ISA support on Intel chips

Skylake supports all **legacy** ISA extensions:

**`MMX, SSE, AVX, AVX2`**

Furthermore **KNL** supports:

- AVX-512 Foundation (F), KNL and Skylake
- AVX-512 Conflict Detection Instructions (CD), KNL and Skylake
- AVX-512 Exponential and Reciprocal Instructions (ER), KNL
- AVX-512 Prefetch Instructions (PF), KNL

AVX-512 extensions only supported on **Skylake**:

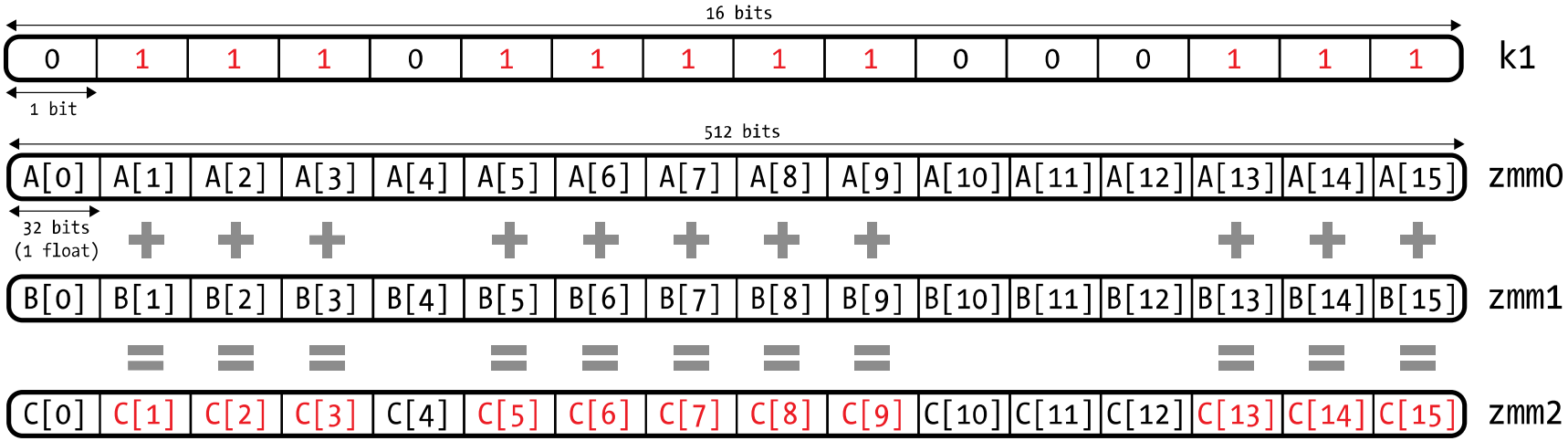
- AVX-512 Byte and Word Instructions (BW)
- AVX-512 Doubleword and Quadword Instructions (DQ)
- AVX-512 Vector Length Extensions (VL)

**ISA Documentation**:

*Intel Architecture Instruction Set Extensions Programming Reference*

# Example for masked execution

Masking for predication is very helpful  in cases such as e.g. remainder loop handling or conditional handling.



# Case Study: Sum reduction (double precision)

```
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 (double precision) – AVX512 version

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
```

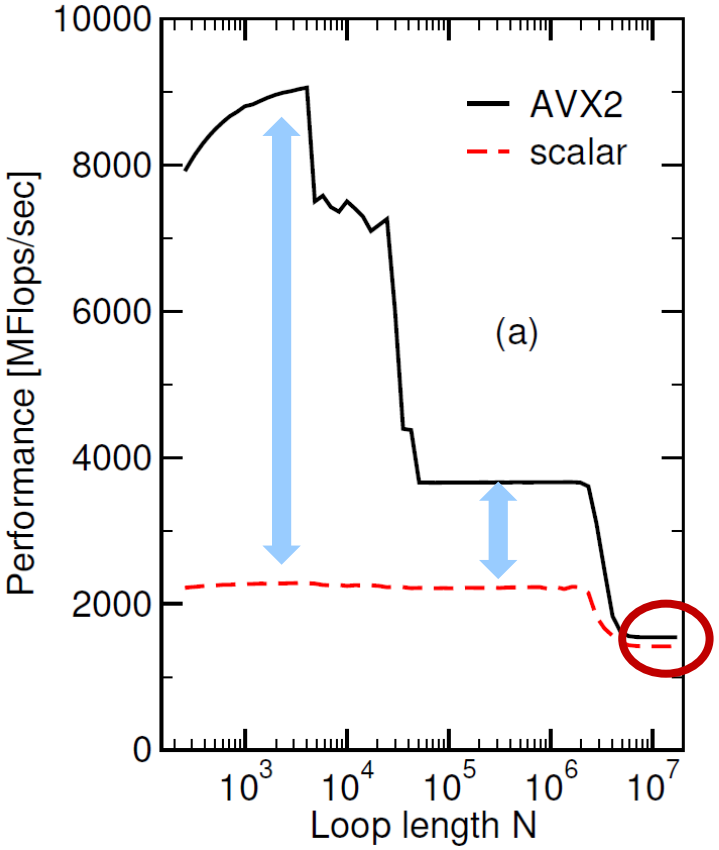
Remainder omitted

Bulk loop code (8x4-way unrolled)

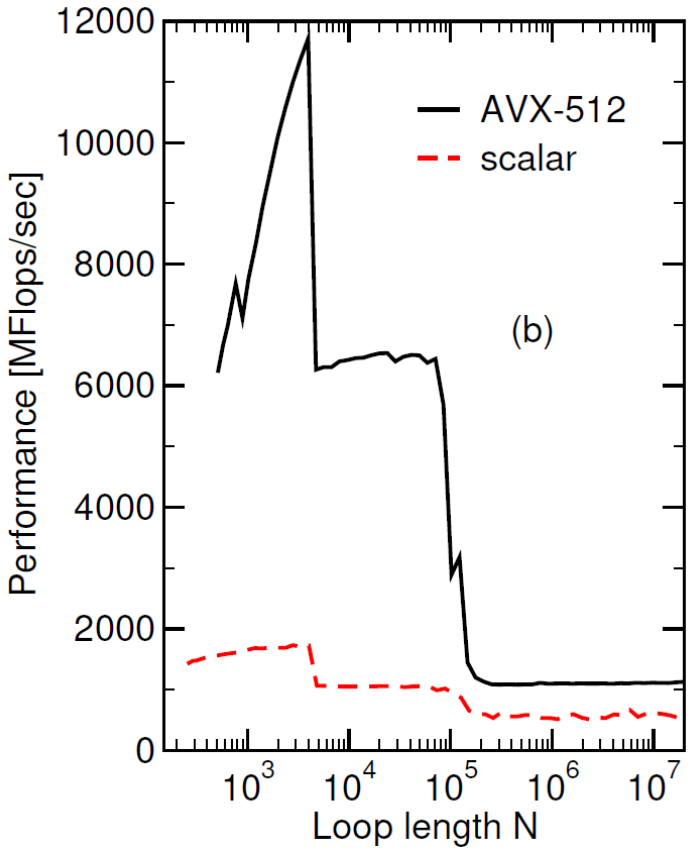
Sum up 32 partial sums into **`xmm0.0`**

# Sum reduction (double precision) – sequential performance

Xeon "Broadwell" E5-2697v4




Xeon Phi 7210




SIMD is an in-core performance feature! If the bottleneck is data transfer, its benefit is limited.

# Rules 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)
5. No (unresolvable) read-after-write data dependencies
6. No function calls (exception intrinsic math 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
4. In C use the **`restrict`** keyword and/or **`const`** qualifiers and/or compiler options to rule out array/pointer aliasing