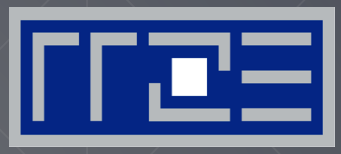
ERLANGEN REGIONAL COMPUTING CENTER



SIMD: Data parallel execution

J. Eitzinger

HLRS, 15.6.2018

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Data parallel execution units (SIMD)

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

Register widths

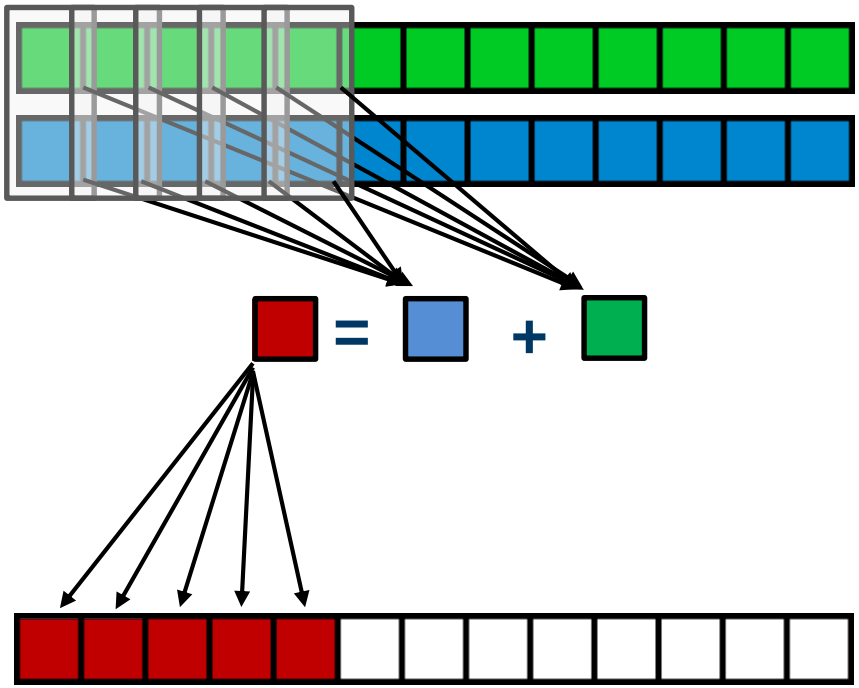
- 1 operand

- 2 operands (SSE)

- 4 operands (AVX)

- 8 operands (AVX512)

Scalar execution



Data parallel execution units (SIMD)

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

Register widths

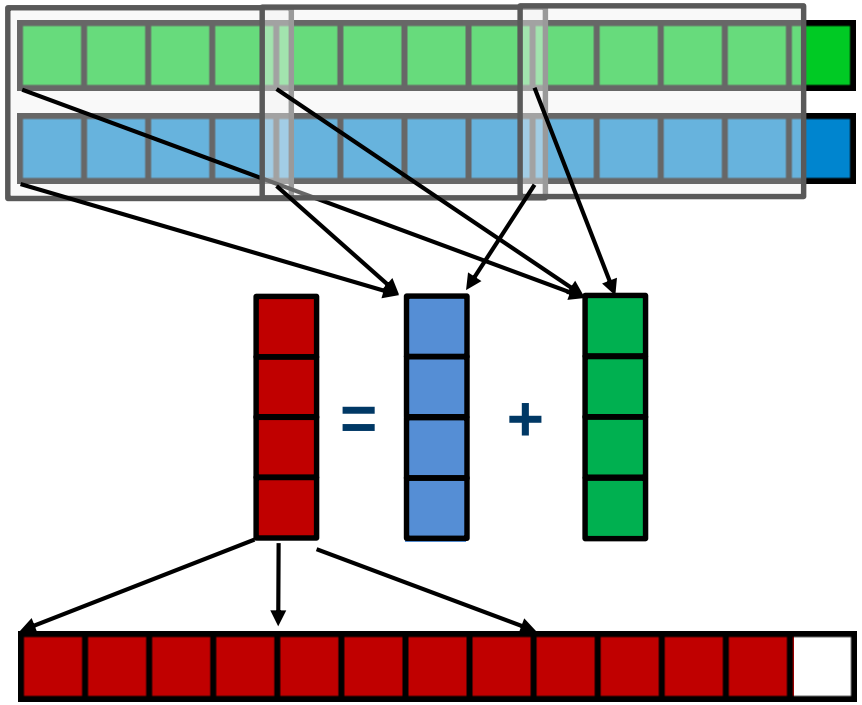
- 1 operand

- 2 operands (SSE)

- 4 operands (AVX)

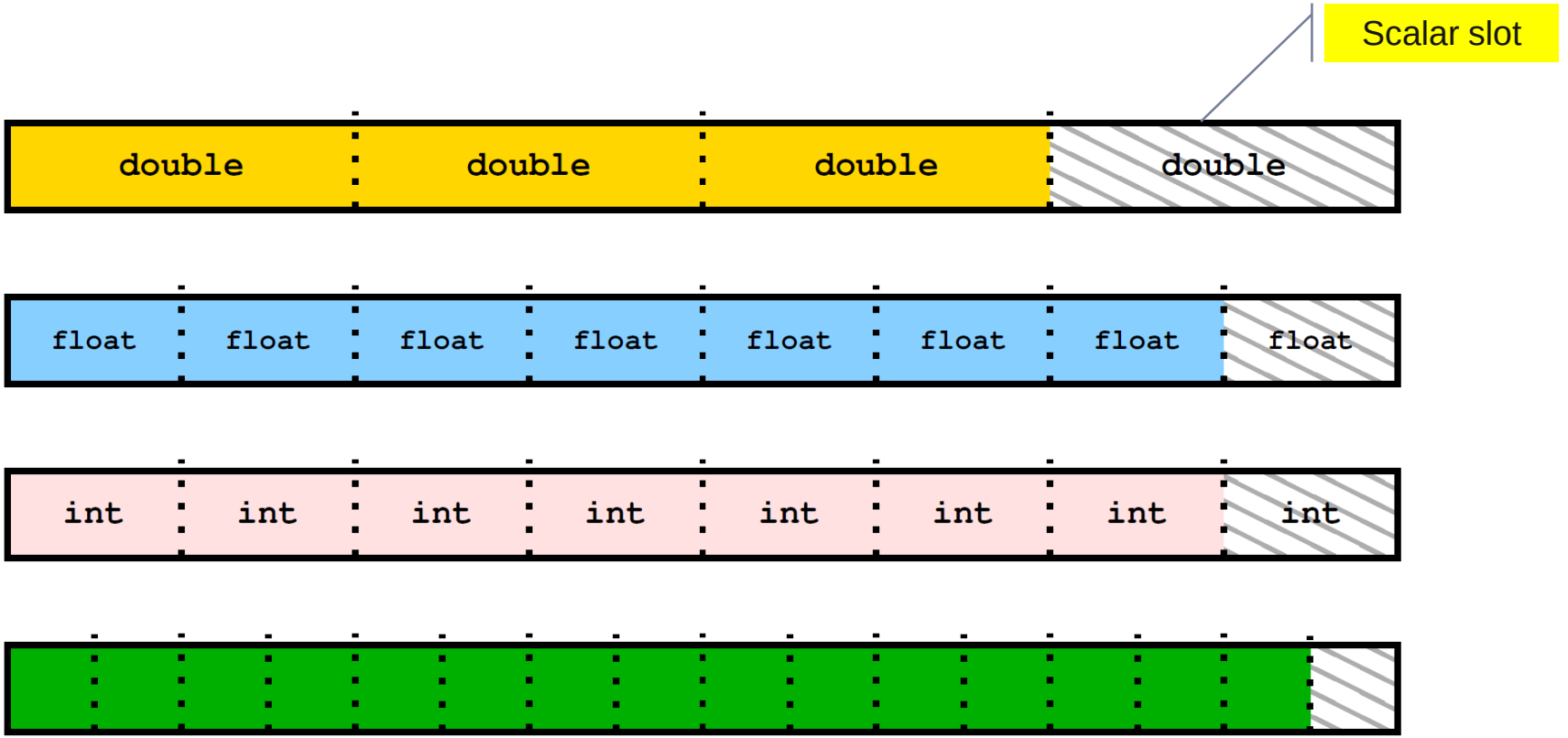
- 8 operands (AVX512)

SIMD execution

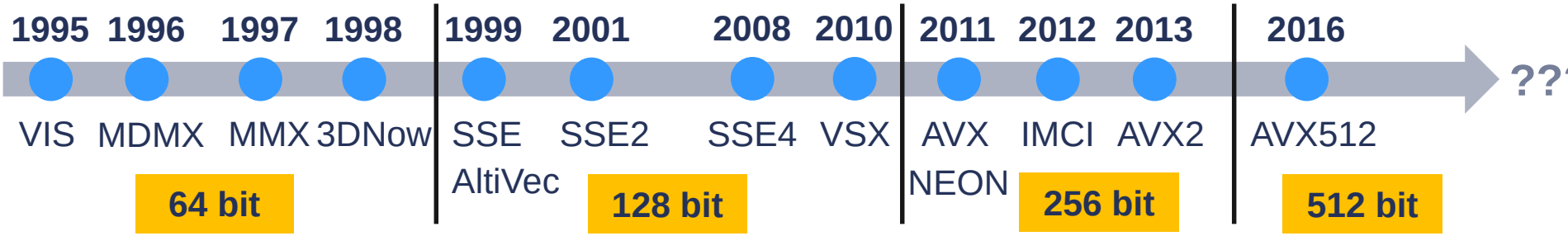


Data types in 32-byte SIMD registers

- **Supported data types depend on actual SIMD instruction set**



History of short vector SIMD



Vendors package other ISA features under the SIMD flag:

`monitor/mwait`, NT stores, prefetch ISA, application specific instructions, FMA

x86

Intel 512bit

AMD 256bit

Power

IBM 128bit

ARM

A5 upward

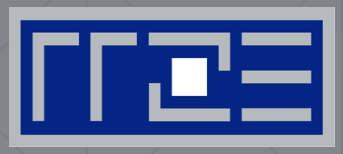
64bit / 128bit

SPARC64

Fujitsu 256bit

K Computer 128bit

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];
```

"inner 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 – 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 it to 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

- The compiler does it for you (but: aliasing, alignment, language)
- 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 64b SIMD with current Intel compilers you need to set:

    -qopt-zmm-usage=high

User-mandated vectorization (OpenMP 4)

- Since OpenMP 4 the **`OMP SIMD`** construct/clause is available
- **`#pragma omp simd`** enforces vectorization where compiler heuristics fail
- Essentially a standardized way of saying “It’s OK, go ahead, loop iterations are truly independent!”
- Prerequesites: countable, inner loop

- There are additional clauses: **`safelen, linear, collapse, reduction, vectorlength, private,`** …

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

- Caution: Do not lie! Using the #pragma omp simd the compiler may generate incorrect code if the loop violates the vectorization rules! It may also generate **inefficient** code!

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 17 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`**)

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

```
movaps    %xmm4, 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         add    $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**:

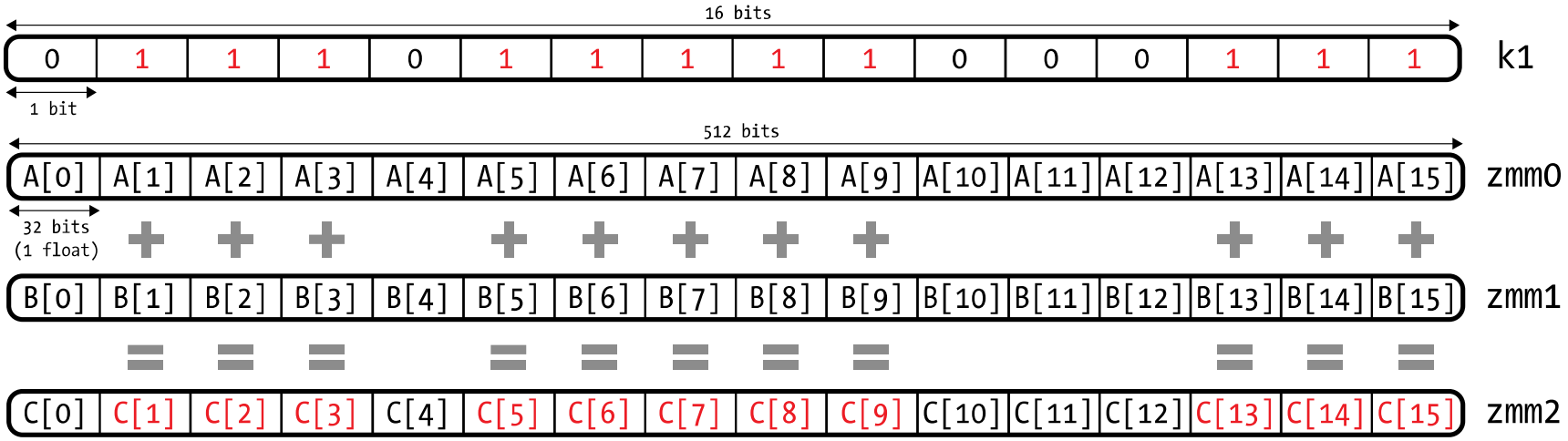
- AVX-512 Byte and Word Instructions (BW)
- AVX-512 Doubleword and Quadword Instructions (DQ)
- AVX-512 Vector Length Extensions (VL)

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: Simplest code for the summation of the elements of a vector (single precision)

```
float 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`

AT&T syntax:
`addss 0(%rdx,%rax,4),%xmm0`

Instruction code:

```
401d08:   f3 0f 58 04 82
401d0d:   48 83 c0 01
401d11:   39 c7
401d13:   77 f3
```

```
addss   xmm0,[rdx + rax * 4]
add     rax,1
cmp     edi,eax
ja      401d08
```

(final sum across xmm0 omitted)

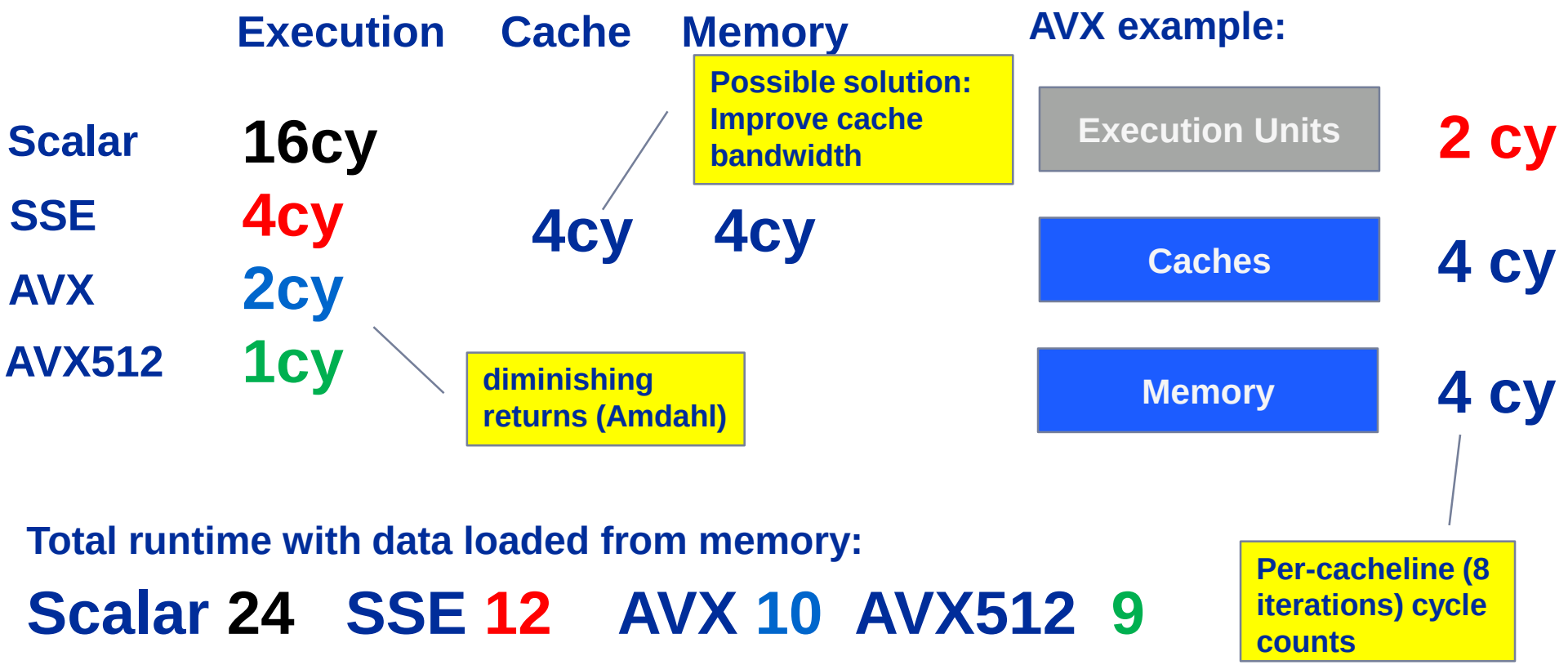
Instruction address

Opcodes

Assembly code

Limits of SIMD processing

- Only part of application may be vectorized, arithmetic vs. load/store (Amdahls law), data transfers
- Memory saturation often makes SIMD obsolete



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