

Erlangen Regional
Computing Center



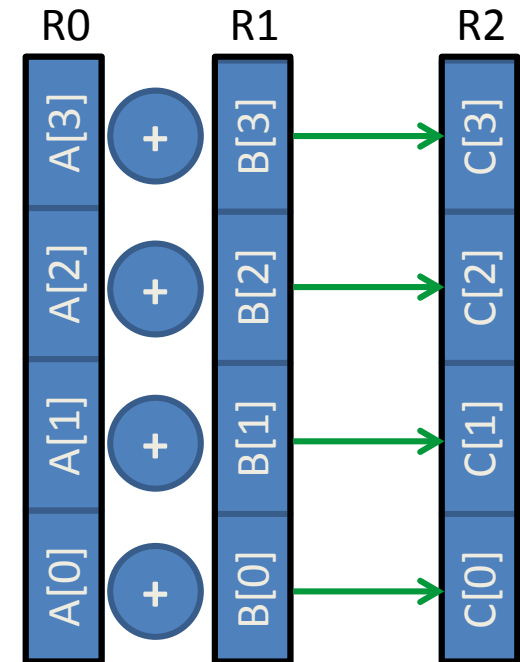
Single Instruction Multiple Data (SIMD) processing

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)



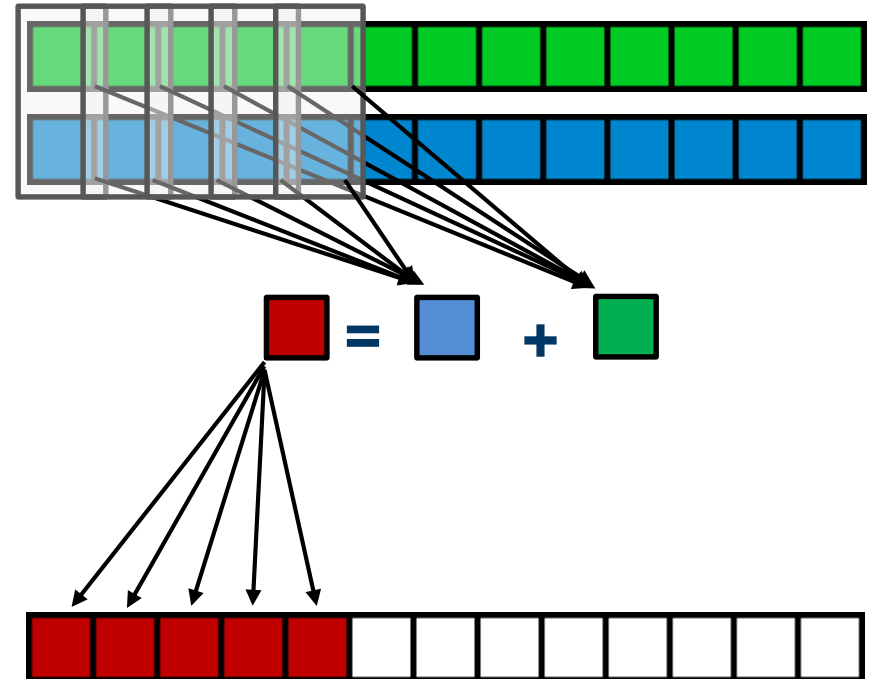
```
for (int j=0; j<size; j++){  
    A[j] = B[j] + C[j];  
}
```

Register widths

- 1 operand



Scalar execution



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

Register widths

- 1 operand



- 2 operands (SSE)



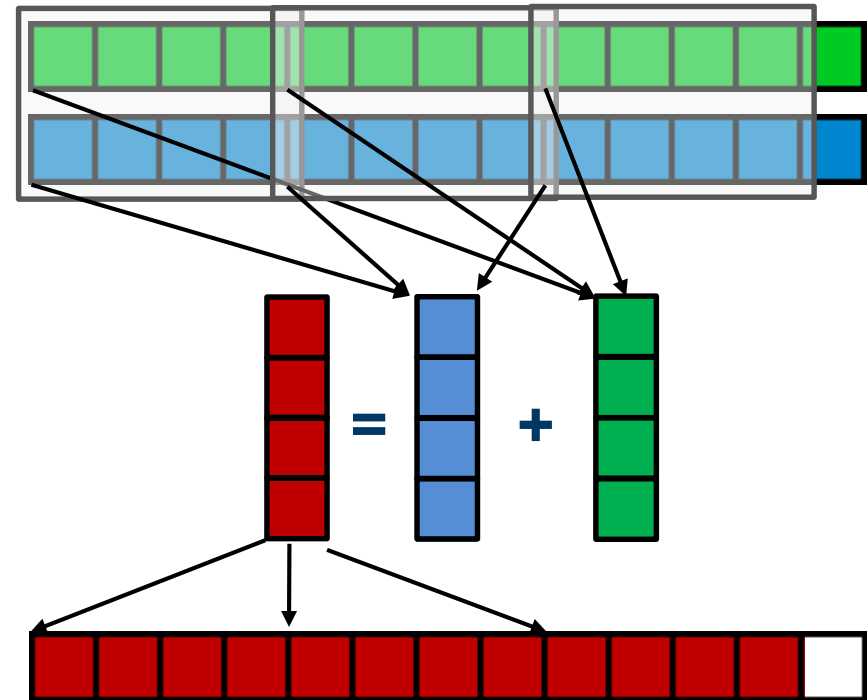
- 4 operands (AVX)



- 8 operands (AVX512)

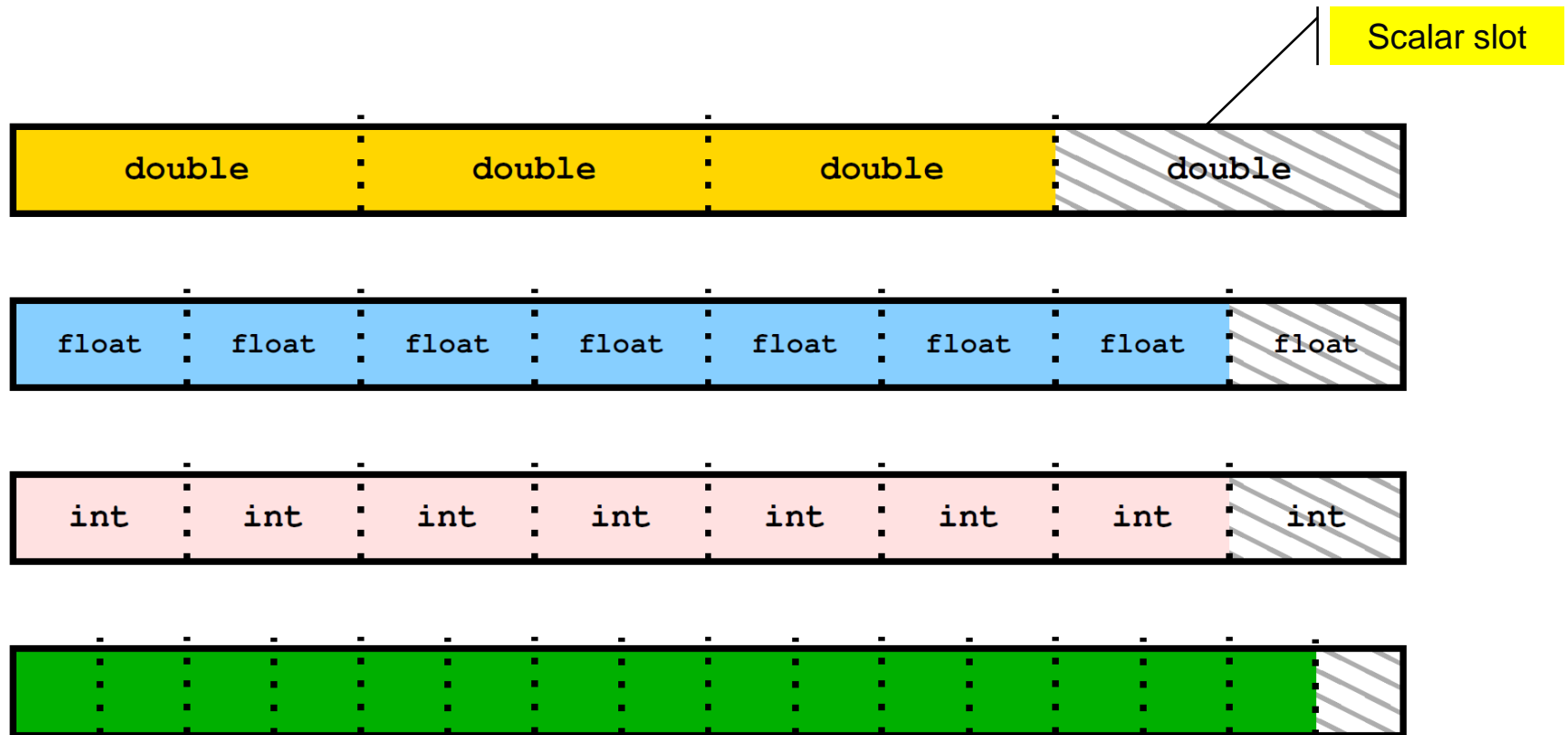


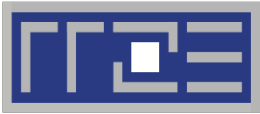
SIMD execution



Best code requires vectorized
LOADs, STOREs, and arithmetic!

Supported data types depend on actual SIMD instruction set





Erlangen Regional
Computing Center



SIMD

The 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
```

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) {...}
```


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**)
- `pmmmintrin.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);  
}
```

- 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
- To really enable 512-bit SIMD with current Intel compilers you need to set:
`-qopt-zmm-usage=high`

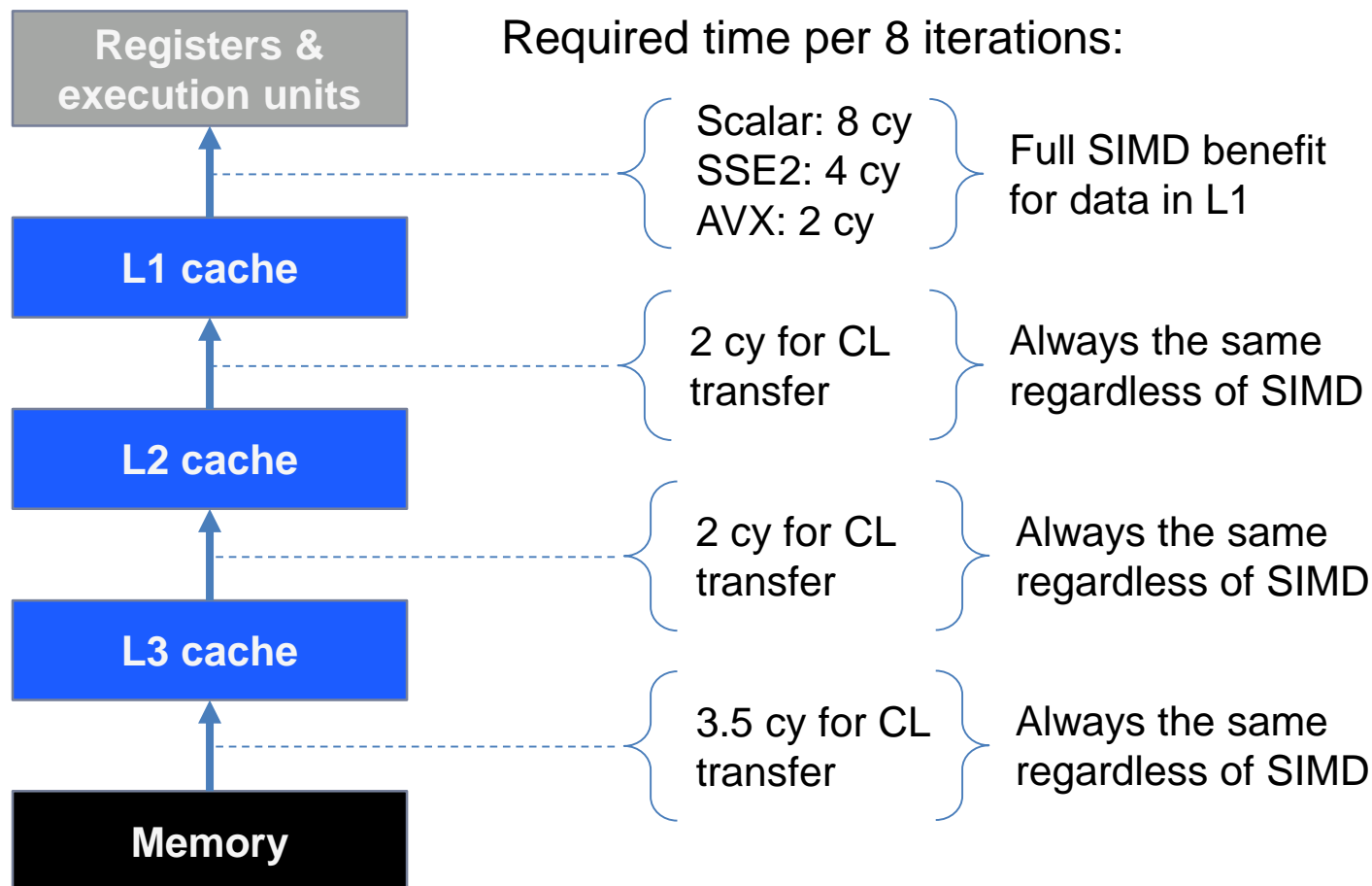
- Since OpenMP 4.0 SIMD features are a part of the OpenMP standard
- `#pragma omp simd` enforces vectorization
- Essentially a standardized “go ahead, no dependencies here!”
 - **Do not lie** to the compiler here!
- Prerequisites:
 - Countable loop
 - Innermost loop
 - Must conform to for-loop style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

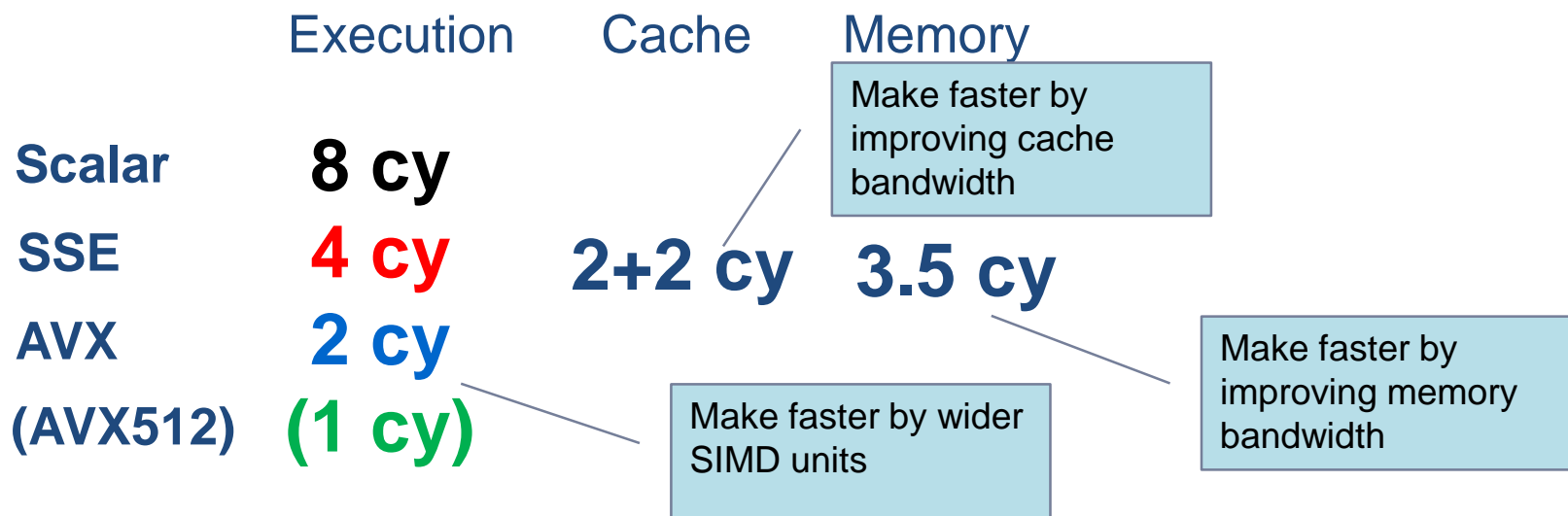
Limits of the SIMD benefit

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 have to 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!)

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 (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 (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!