Concepts and Models of Parallel and Data-centric Programming

Shared Memory XIII

Lecture, Summer 2020

Dr. Christian Terboven <terboven@itc.rwth-aachen.de>

High Performance Computing

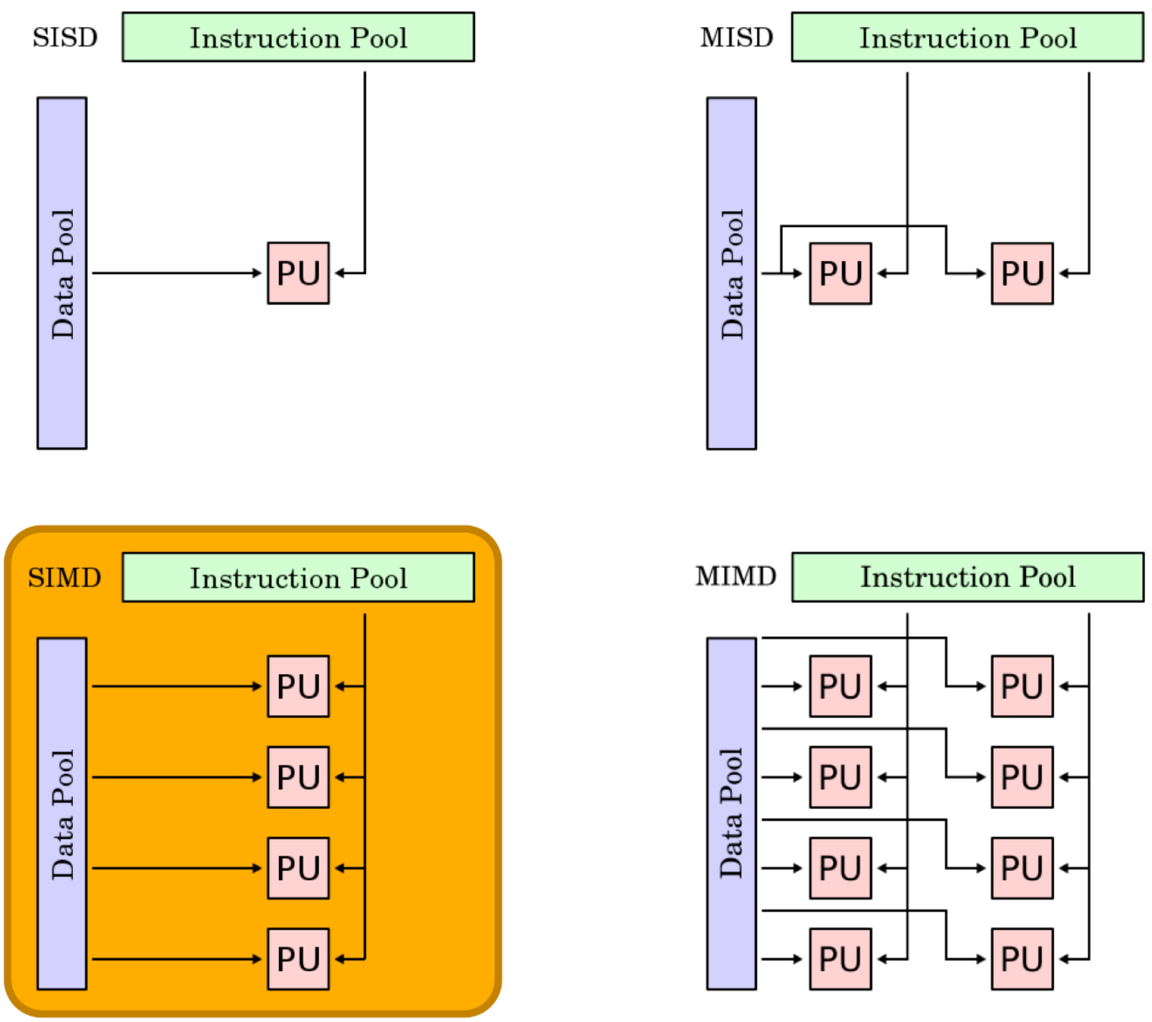
RWTH AACHEN UNIVERSITY

Outline

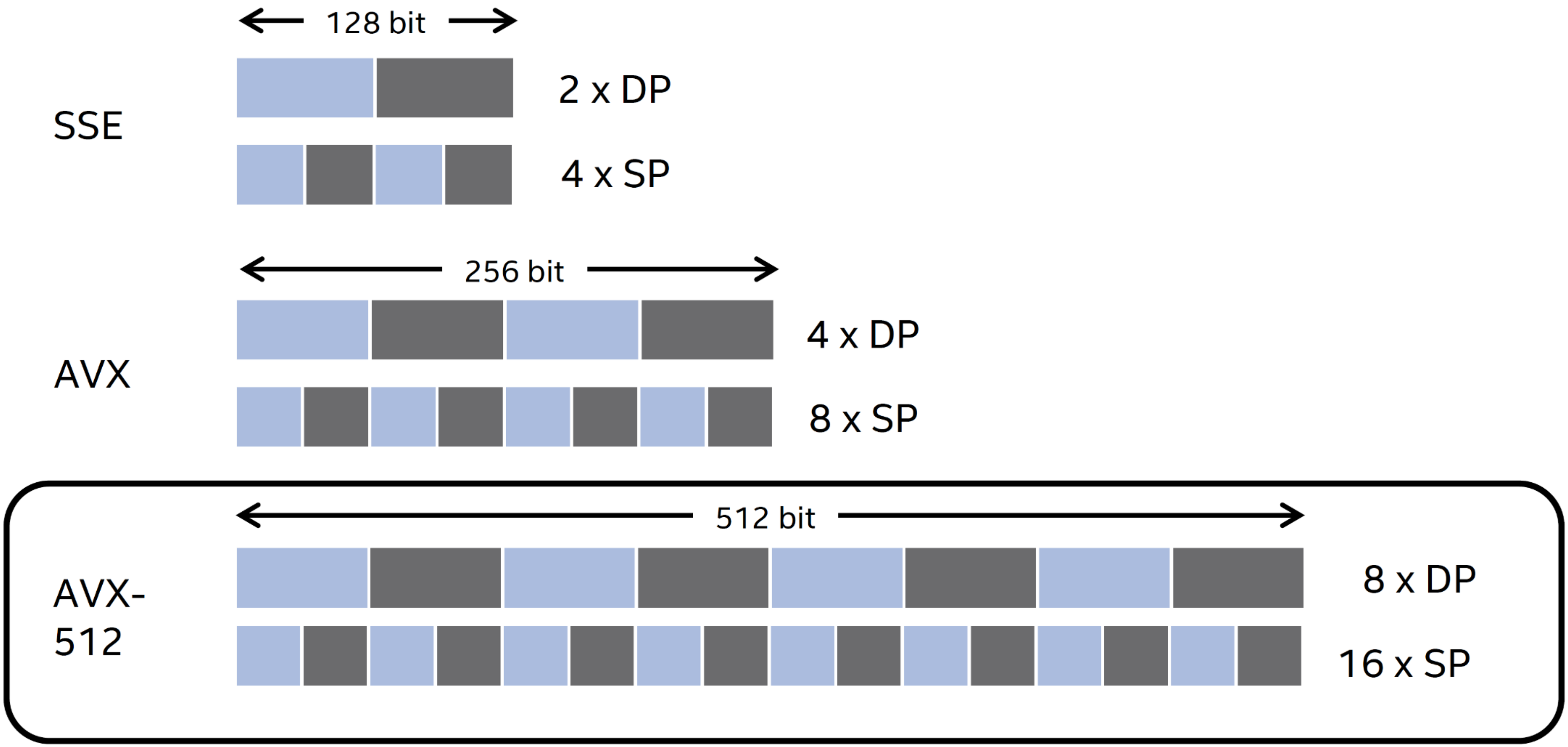
SIMD / Vectorization

(Intel-related slides contributed by Dr. Michael Klemm)

Flynn's taxonomy: SIMD

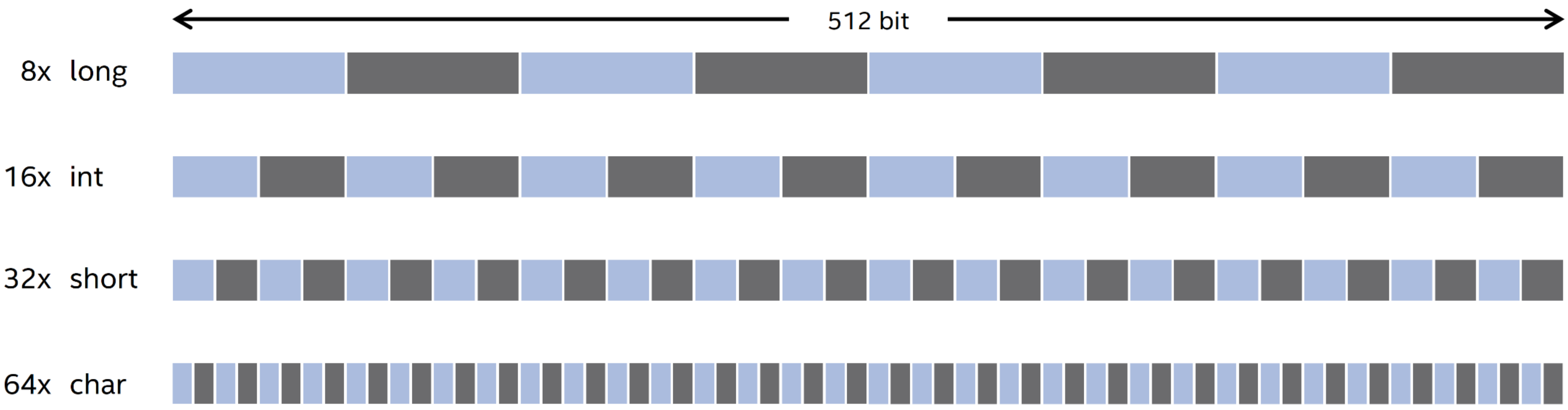


Evolution of SIMD on Intel / 1



Evolution of SIMD on Intel / 2

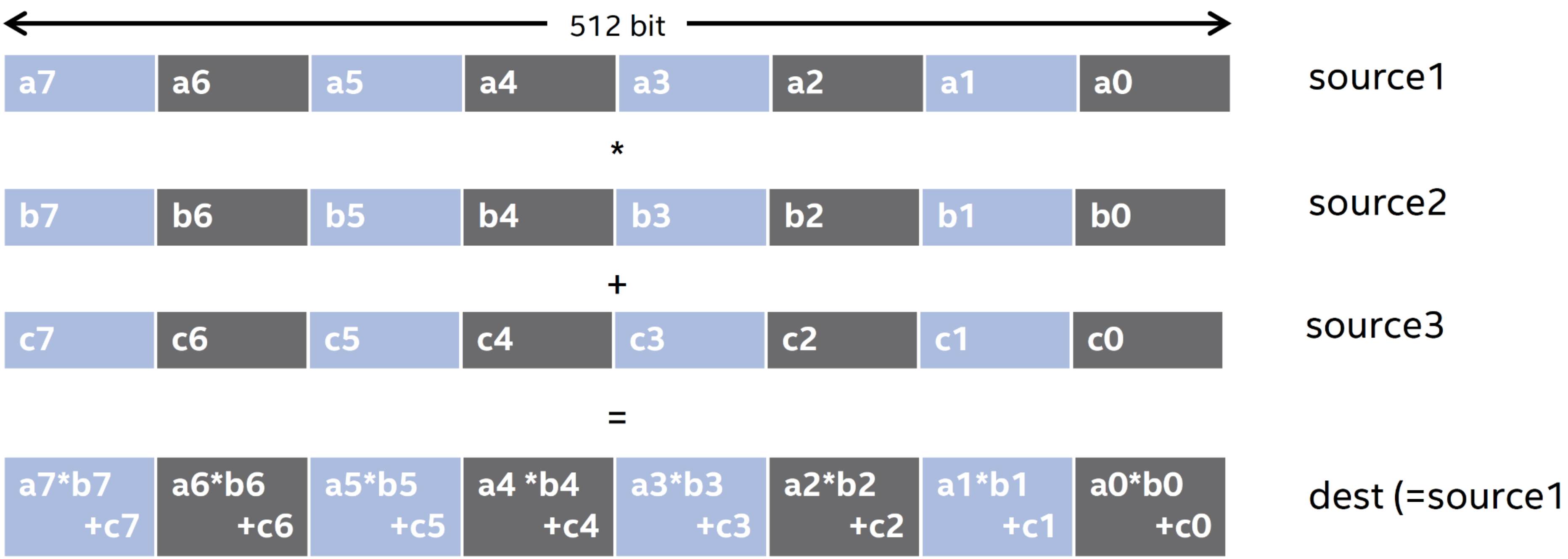
- SIMD instruction sets typically support many more data types



Evolution of SIMD on Intel / 3

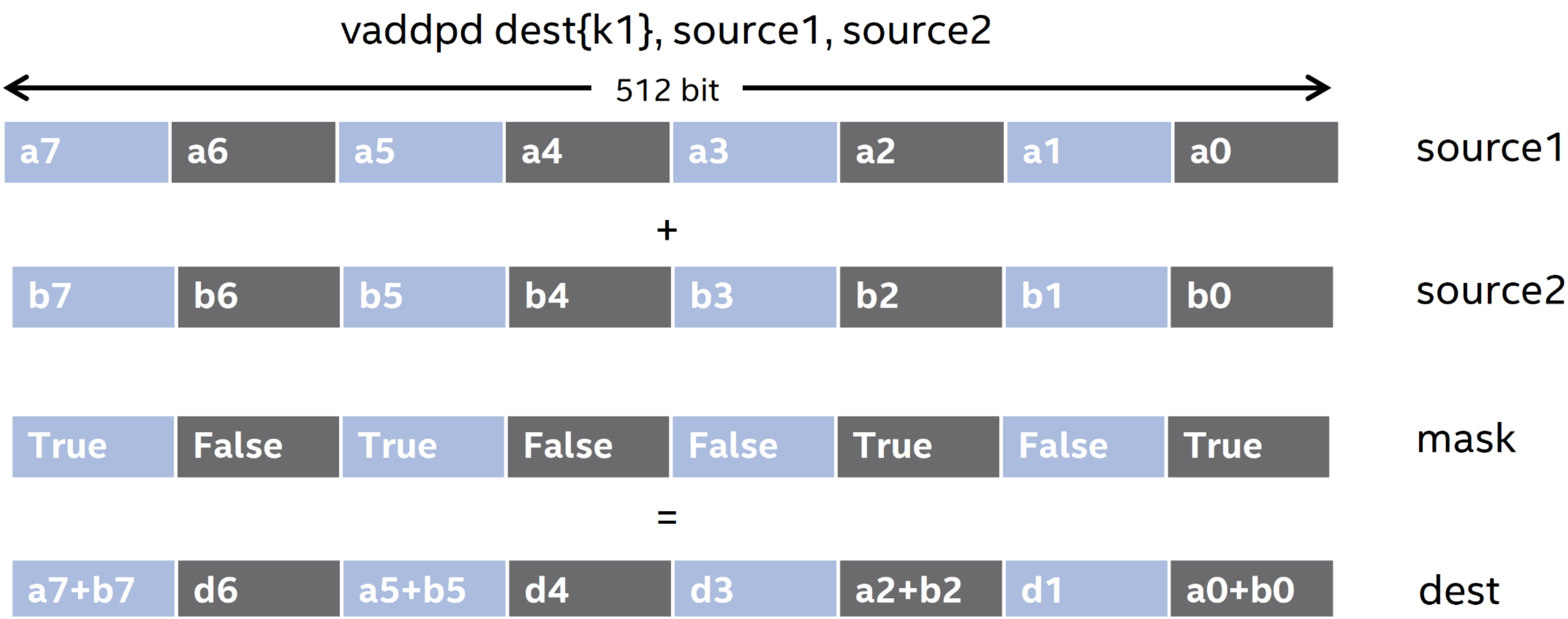
- Operations work on each individual SIMD element
- Two operations (here: multiply & add) may be fused into one SIMD instruction

vfmadd213pd source1, source2, source3



Evolution of SIMD on Intel / 4

- Mask register limit effect of instructions to a subset of the SIMD elements



- More operations: broadcast, swizzle, move, ...

Intrinsics

SIMD through C intrinsics

- Example: Intel AVX intrinsic functions

AVX SIMD Data Types

- __m256: a vector of 8 float entries
- __m256d: a vector of 4 double entries
- __m256i: a vector of 4 longs (or 8 int, ...)

Intrinsic functions

```
__m256 a = _mm256_add_pd(__m256 a, __m256 b)
```

Vector length:

_mm512
_mm256
_mm128

Functionality:

- add
- mul
- sub
- load

...

Type and precision:

pd: packed double
ps: packed single
sd: scalar double
ss: scalar single

SIMDifying saxpy / 1

- Scalar code:

```
void saxpy(float *y, float *x, float a, int n)
{
    for (int i = 0; i < n; ++i)
    {
        y[i] = a * x[i] + y[i];
    }
}
```

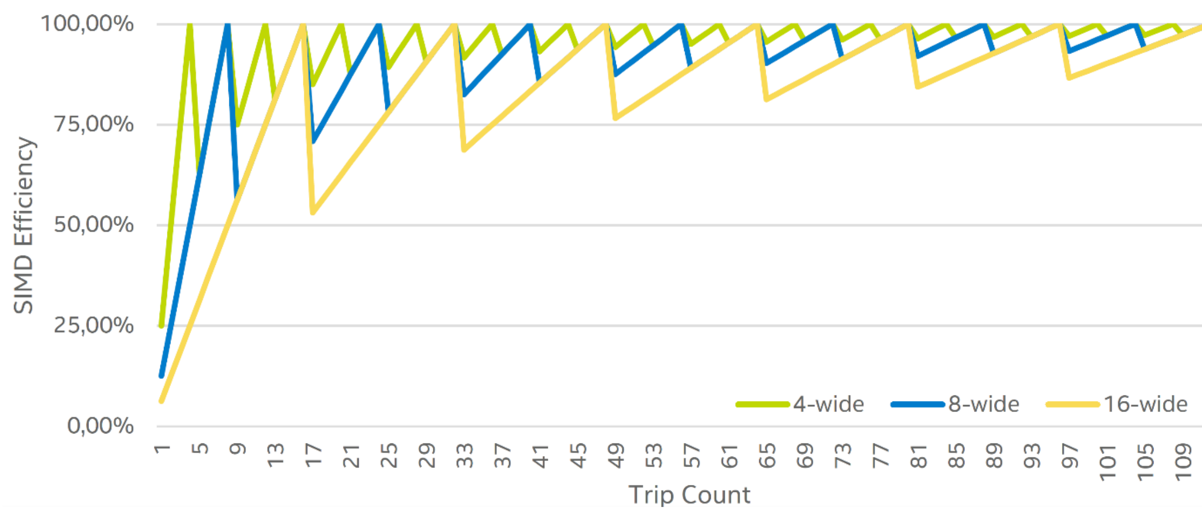
SIMDifying saxpy / 2

- Scalar code:

```c
void saxpy_simd(float *y, float *x, float a, int n)
{
    int ub = n - (n % 8); /* 8 floats per SIMD register */
    __mm256 vy, vx, va, tmp;
    va = _mm256_set1_ps(a);
    for (int i = 0; i < ub; i += 8)
    {
        vy = _mm256_loadu_ps(&y[i]); vx = _mm256_loadu_ps(&x[i]);
        tmp = _m256_mul_ps(va, vx);  vy = _mm256_add_ps(tmp, vy);
        _mm256_storeu_ps(&y[i], vy);
    }
    for (int i = ub; i < n; ++i)
    {
        y[i] = a * x[i] + y[i];
    }
}
```

Vectorization Efficiency

- Measure how well the code uses SIMD features
  - Corresponds to the average utilization of SIMD registers for a loop
  - Defined as (N: trip count, vl: vector length): $VE = \frac{N/vl}{\lceil N/vl \rceil}$
- For 8-wide SIMD:
  - N = 1: 12.50 %
  - N = 2: 25.00 %
  - N = 4: 50.00 %
  - N = 8: 100.00 %
  - N = 9: 56.25 %
  - N = 16: 100.00%



- Note: there are ways to provide a SIMD remainder loop for the example from the previous slide (or: variable vector length)

Parallel STL

Data Dependencies

- For two statements S1 and S2:

- S2 depends on S1, iff S1 must execute before S2
  - Control-flow dependence
  - Data dependence
  - Dependencies can be carried over between loop iterations

- Important flavors of data dependencies

FLOW

```
s1: a = 40
    b = 21
s2: c = a + 2
```

ANTI

```
    b = 40
s1: a = b + 1
s2: b = 21
```

Loop-carried Dependencies / 1

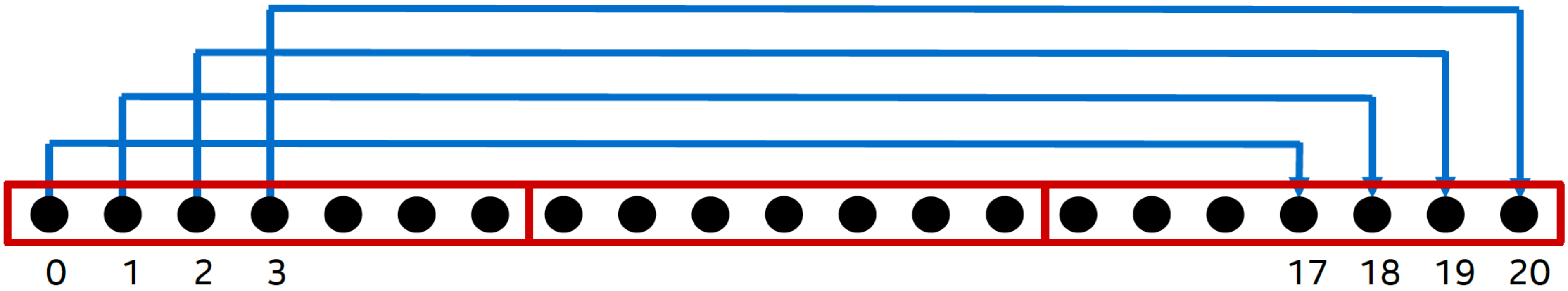
- Dependencies may occur across loop iterations
  - Then call loop-carried dependencies

- Distance of a dependency: number of loop iterations it spans

- Reason: some iterations have to complete before the next iteration can run
  - Would reversing the loop deliver wrong results? (not sufficient)

Loop-carried Dependencies / 2

- Can we parallelize or vectorize this loop?




```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```

- Parallelization: no
  (except for very specific loop schedules)
- Vectorization: yes
  (iff vector length is shorter than any distance of any dependency)

Execution Policies

- C++ execution policy: call an (STL) algorithm and and specify how it can be executed
  - Defined in header `<execution>`
  - Parallel and Vectorized: **parallel_unsequenced_policy**: `std::execution::par_unseq`
  - Parallel: **parallel_policy**: `std::execution::par`
  - Serial: **sequenced_policy**: `std::execution::seq`
  - C++20 is expected to bring „just vectorized“

- Parallel execution: programmer‘s responsibility to avoid data races and deadlocks

- Unsequenced execution: use of vectorization-unsafe operations not allowed (ex.: `std::mutex::unlock()` synchronizes with `std::mutex:lock()`)

Example: Execution Policies

```
// generate some (large) vector
std::vector<int> v = /* ... some code here ... */

// standard (sequential) sort
std::sort(v.begin(), v.end());

// enforce sequential execution
std::sort(std::seq, v.begin(), v.end());

// permit parallel execution
std::sort(std::par, v.begin(), v.end());

// permit vectorization as well
std::sort(std::par_unseq, v.begin(), v.end());
```

Parallelization-enabled Algorithms

- `for_each`: similar to `std::for_each` except returns `void`
- `for_each_n`: applies a function object to the first n elements of a sequence
- `reduce`: similar to `std::accumulate`, except out of order execution
- `exclusive_scan`: similar to `std::partial_sum`, excludes the i-th input element from the i-th sum
- `inclusive_scan`: similar to `std::partial_sum`, includes the i-th input element in the i-th sum
- `transform_reduce`: applies a functor, then reduces out of order
- `transform_exclusive_scan` - applies a functor, then calculates exclusive scan
- `transform_inclusive_scan` - applies a functor, then calculates inclusive scan