Vectorization with OpenMP

Dirk Schmidl <schmidl@itc.rwth-aachen.de>

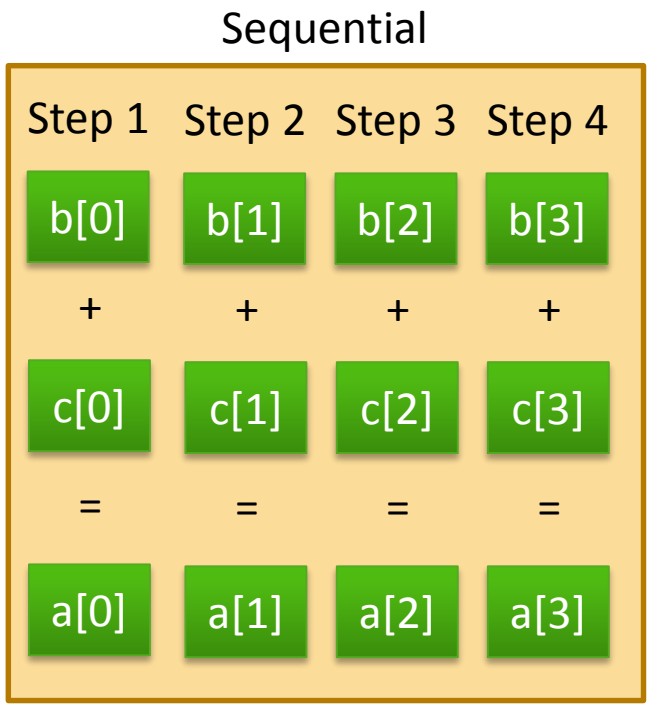
19.03.2015 / Aachen, Germany

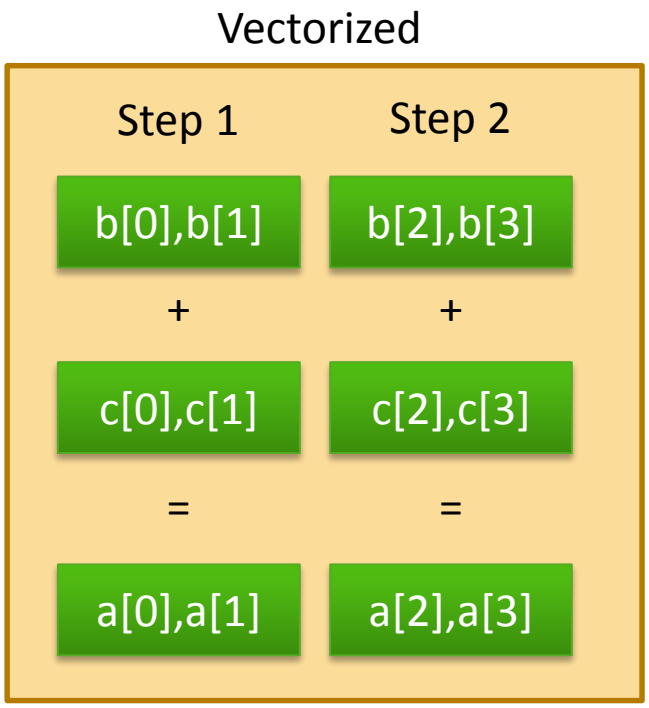
Stand: 19.03.2015

Vectorization

- **SIMD = S**ingle **I**nstruction **M**ultiple **D**ata
  - → Special hardware instructions to operate on multiple data points at once
  - → Instructions work on vector registers
  - → Vector length is hardware dependent

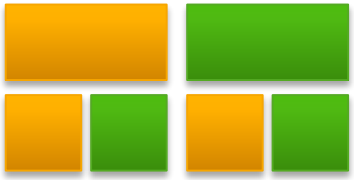
```
double a[4],b[4],c[4];
…

for(i=0 ; i < 4 ; i++)
{
      a[i]=b[i]+c[i];
}
```





Vectorization

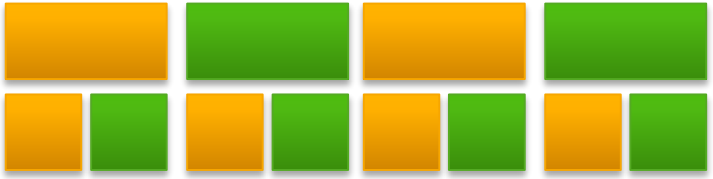
- **Vector lengths on Intel architectures**
  - → 128 bit: SSE = Streaming SIMD Extensions



2 x double

4 x float

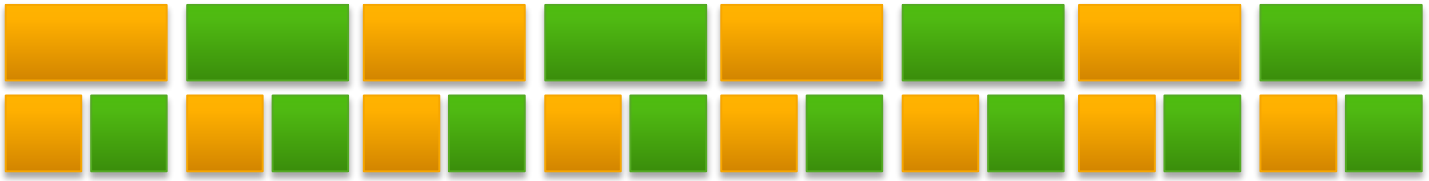
  - → 256 bit: AVX = Advanced Vector Extensions



4 x double

8 x float

  - → 512 bit: AVX-512



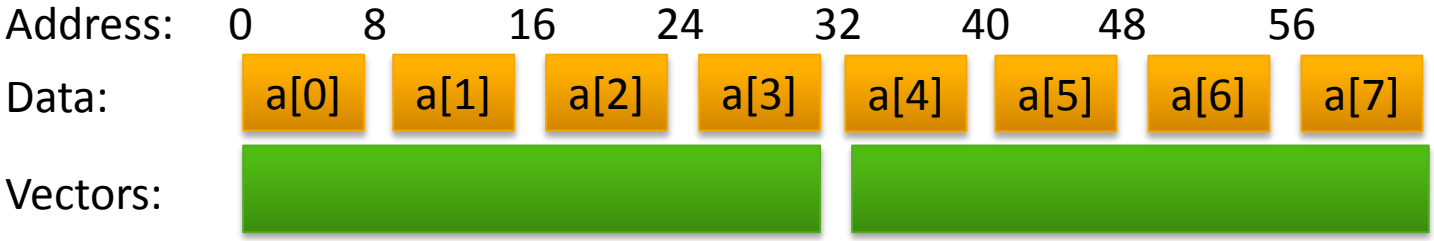
8 x double

16 x float

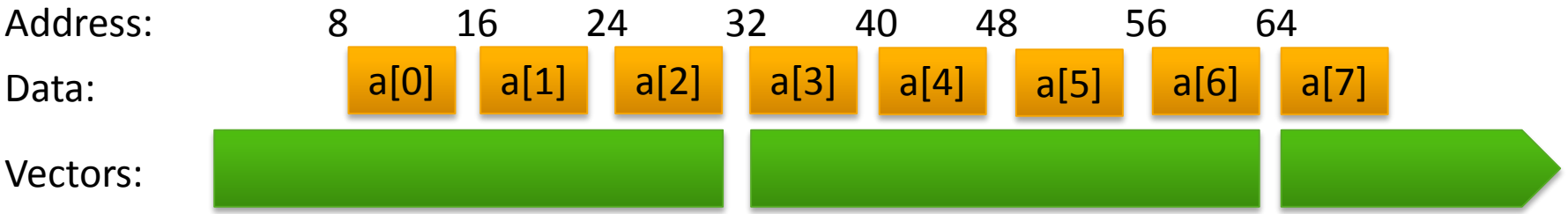
Data Alignment

- **Vectorization works best on aligned data structures.**

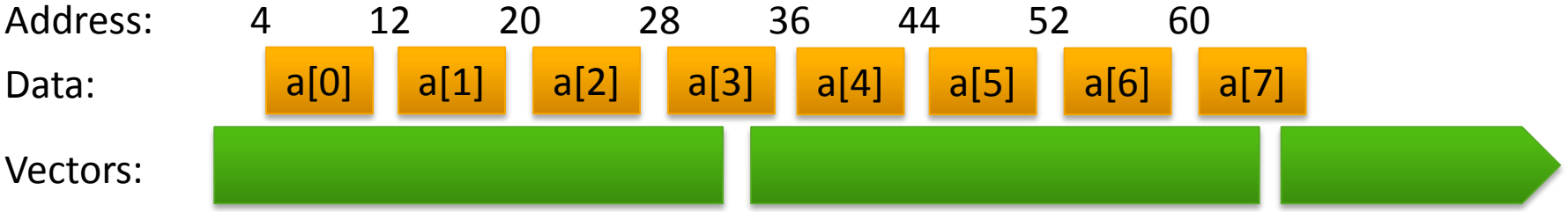
Good alignment



Bad alignment

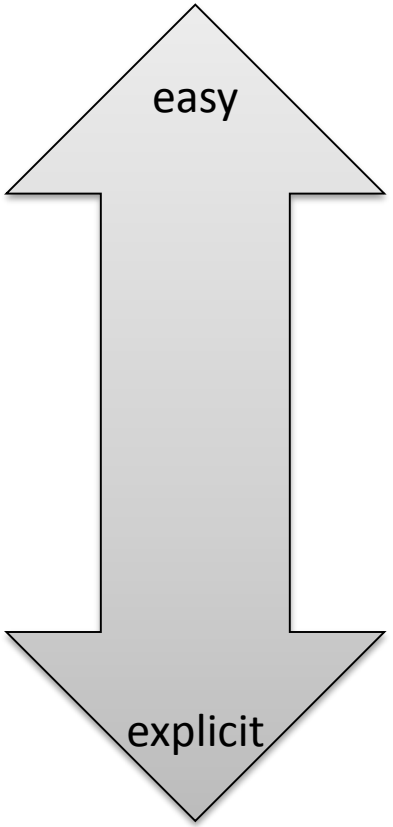


Very bad alignment



Ways to Vectorize

- Compiler
  auto-vectorization
- Explicit Vector Programming
  (e.g. with OpenMP)
- Inline Assembly
  (e.g.)
- Assembler Code
  (e.g. addps, mulpd, …)



The OpenMP SIMD constructs

The SIMD construct

- **The SIMD construct enables the execution of multiple iterations of the associated loops concurrently by means of SIMD instructions.**

C/C++:
```
#pragma omp simd [clause(s)]
  for-loops
```

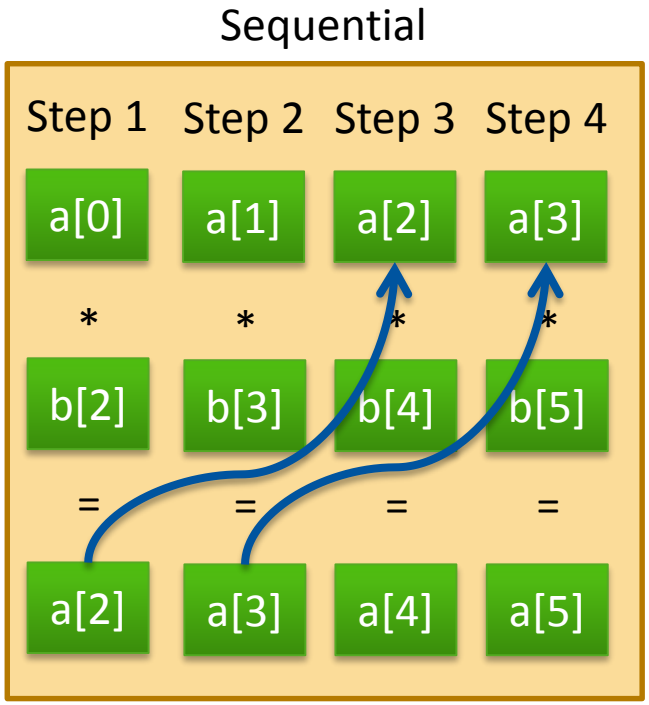
Fortran:
```
!$omp simd [clause(s)]
  do-loops
[!$omp end simd]
```

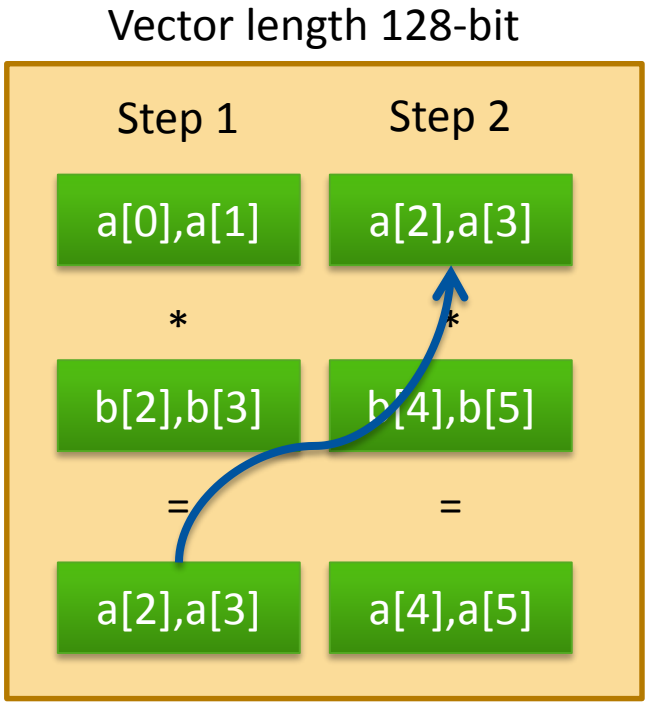
- **where clauses are:**
  - → linear(*list[:linear-step]*), a variable increases linearly in every loop iteration
  - → aligned(*list[:alignment]*), specifies that data is aligned
  - → private(*list*), as usual
  - → lastprivate(*list*) , as usual
  - → reduction(*reduction-identifier:list*) , as usual
  - → collapse(*n*), collapse loops first, and than apply SIMD instructions

The SIMD construct

- **The safelen clause allows to specify a distance of loop iterations where no dependencies occur.**

```
double a[6],b[6];
…

for(i=2 ; i < 6 ; i++)
{
      a[i]=a[i-2]*b[i];
}
```

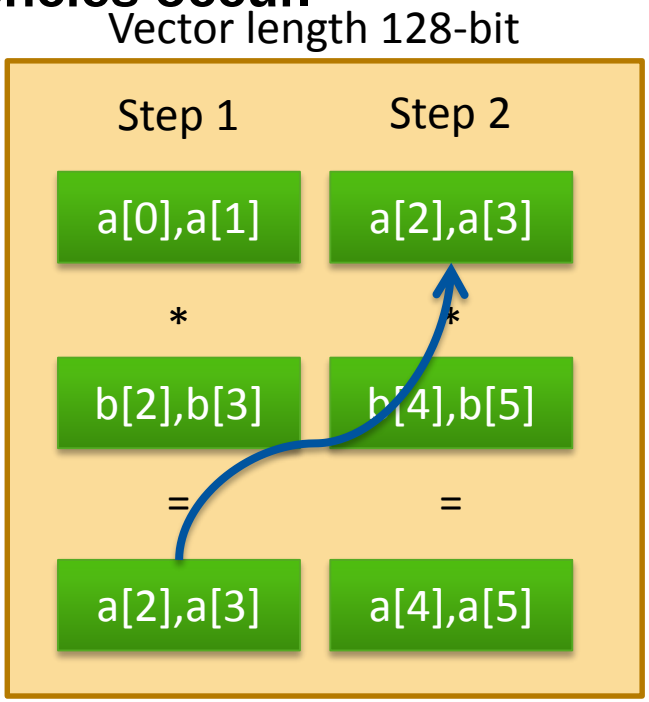




The SIMD construct

- **The safelen clause allows to specify a distance of loop iterations where no dependencies occur.**

```
double a[6],b[6];
…

for(i=2 ; i < 6 ; i++)
{
      a[i]=a[i-2]*b[i];
}
```



- **Any vector length smaller than or equal to the length specified by safelen can be chosen for vectorizaion.**
- **In contrast to parallel for/do loops the iterations are executed in a specified order.**

The loop SIMD construct

- **The loop SIMD construct specifies a loop that can be executed in parallel by all threads and in SIMD fashion on each thread.**

C/C++:
```
#pragma omp for simd [clause(s)]
  for-loops
```

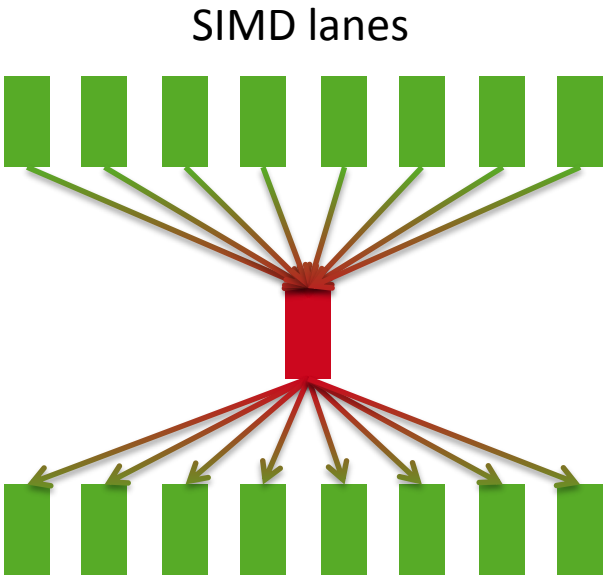
Fortran:
```
!$omp do simd [clause(s)]
  do-loops
[!$omp end do simd [nowait]]
```

- **Loop iterations are first distributed across threads, then each chunk is handled as a SIMD loop.**
- **Clauses:**
  - → All clauses from the *loop*- or SIMD-construct are allowed
  - → Clauses which are allowed for both constructs are applied twice, once for the threads and once for the SIMDization.

The declare SIMD construct

- **Function calls in SIMD-loops can lead to bottlenecks, because functions need to be executed serially.**

```
for(i=0 ; i < N ; i++)
{

      a[i]=b[i]+c[i];

      d[i]=sin(a[i]);

      e[i]=5*d[i];

}
```



Solutions:

- avoid or inline functions
- create functions which work on vectors instead of scalars

The declare SIMD construct

- **Enables the creation of multiple versions of a function or subroutine where one or more versions can process multiple arguments using SIMD instructions.**

C/C++:
```
#pragma omp declare simd [clause(s)]
[#pragma omp declare simd [clause(s)]]
  function definition / declaration
```

Fortran:
```
!$omp declare simd (proc_name)[clause(s)]
```

- **where clauses are:**
  - → simdlen(*length*), the number of arguments to process simultanously
  - → linear(*list[:linear-step]*), a variable increases linearly in every loop iteration
  - → aligned(*argument-list[:alignment]*), specifies that data is aligned
  - → uniform(*argument-list*), arguments have an invariant value
  - → inbranch / notinbranch, function is always/never called from within a conditional statement

File: f.c

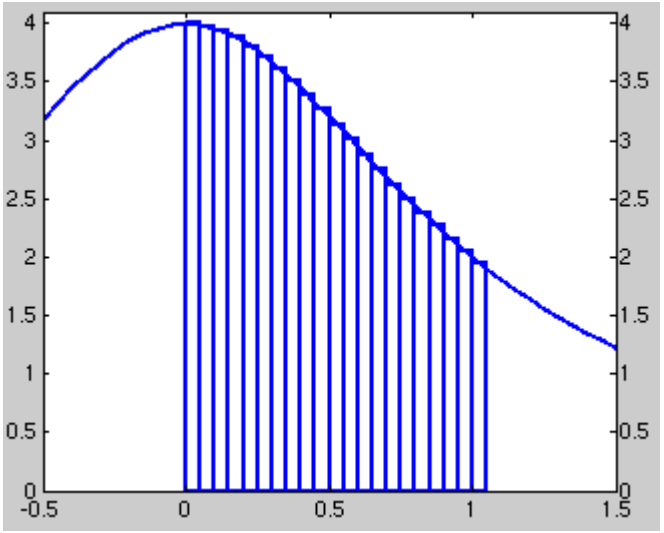
```
#pragma omp declare simd
double f(double x)
{
   return (4.0 / (1.0 + x*x));
}
```

File: pi.c

```
#pragma omp declare simd
double f(double x);
…
#pragma omp simd linear(i) private(fX) reduction(+:fSum)
for (i = 0; i < n; i++)
{
   fX = fH * ((double)i + 0.5);
   fSum += f(fX);
}
return fH * fSum;
```

Calculating Pi with numerical integration of:

$$\pi = \int_0^1 \frac{4}{1 + x^2}$$



Example 1: Pi

- **Runtime of the benchmark in sec. on the Xeon Phi**

Threads	no SIMD	SIMD
1	121.2	15.8
2	65.9	7.9
4	30.0	4.0
8	15.0	2.0
16	7.5	1.0
32	3.8	0.5
60	2.6	0.4
120	1.6	0.3
240	1.4	0.4

Note: Speedup for memory bound applications might be lower on both systems.