# Advanced OpenMP Features

Christian Terboven, Dirk Schmidl

IT Center, RWTH Aachen University

Member of the HPC Group

{terboven,schmidl}@itc.rwth-aachen.de

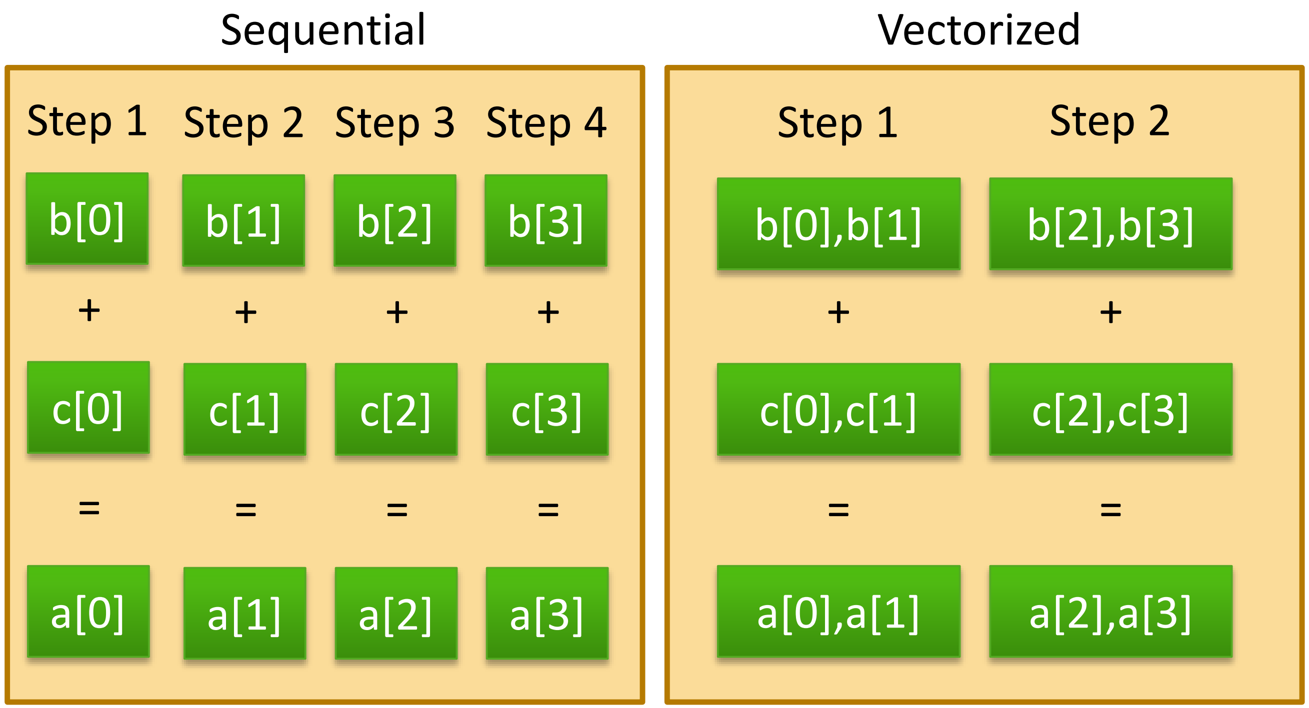
# Vectorization

# 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

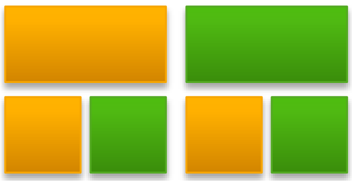
```
double a[4],b[4],c[4];
…

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



## Vector lengths on Intel architectures

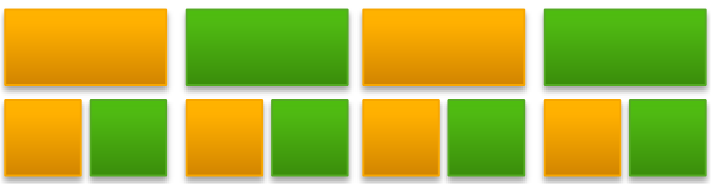
→ 128 bit: SSE = Streaming SIMD Extensions



2 x double

4 x float

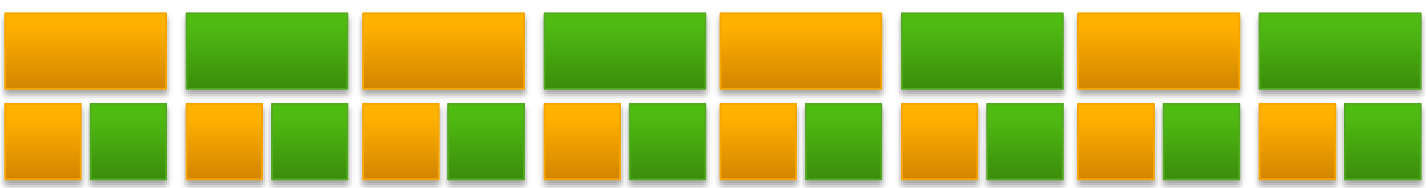
→ 256 bit: AVX = Advanced Vector Extensions



4 x double

8 x float

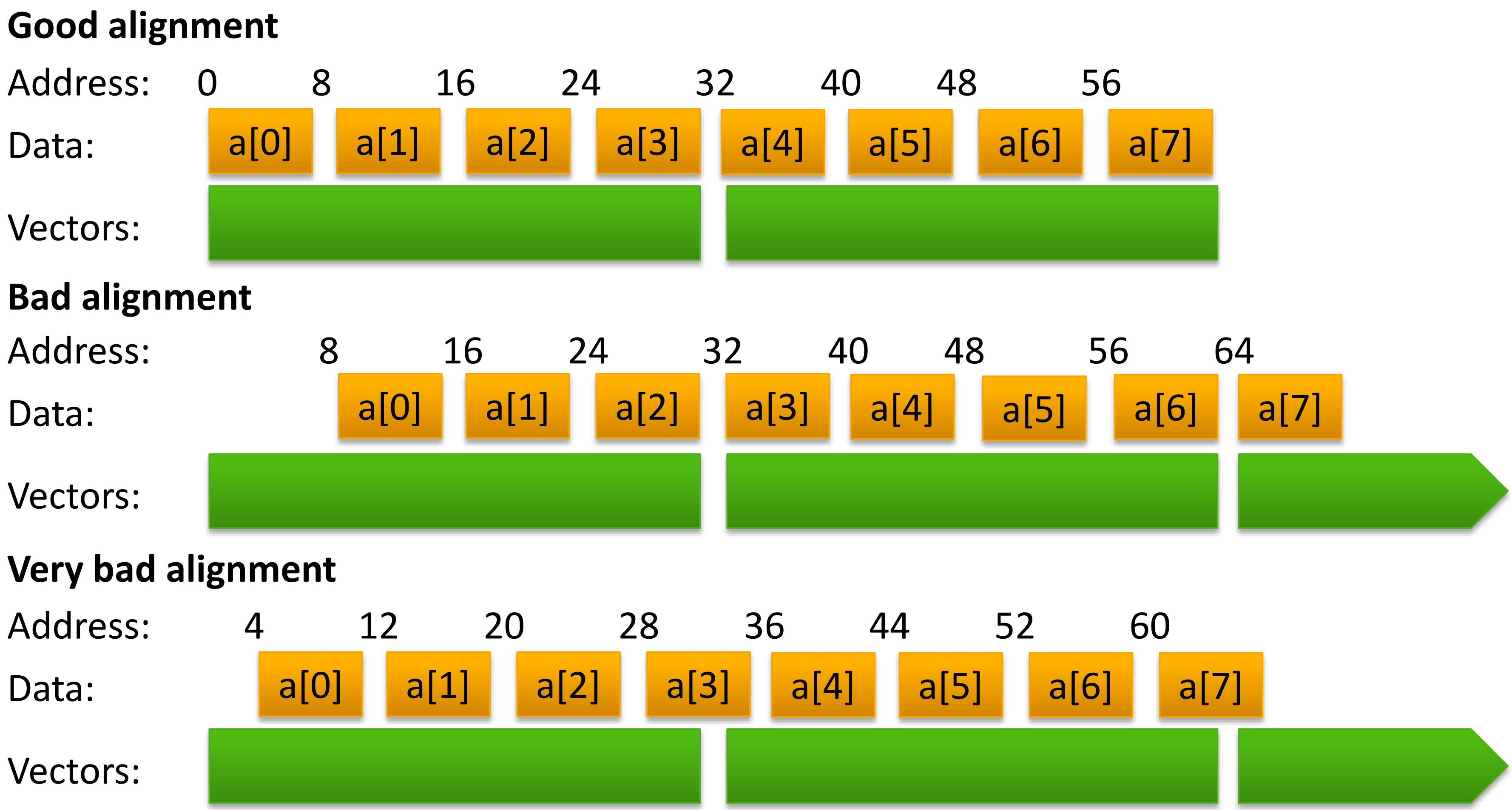
→ 512 bit: AVX-512



8 x double

16 x float

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



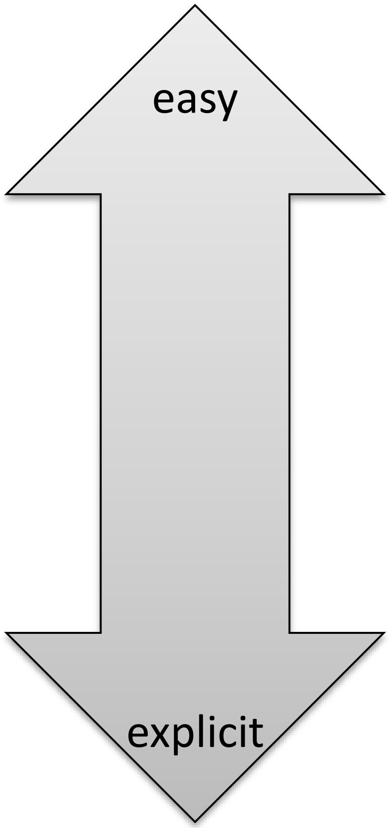
## Ways to Vectorize

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]
```

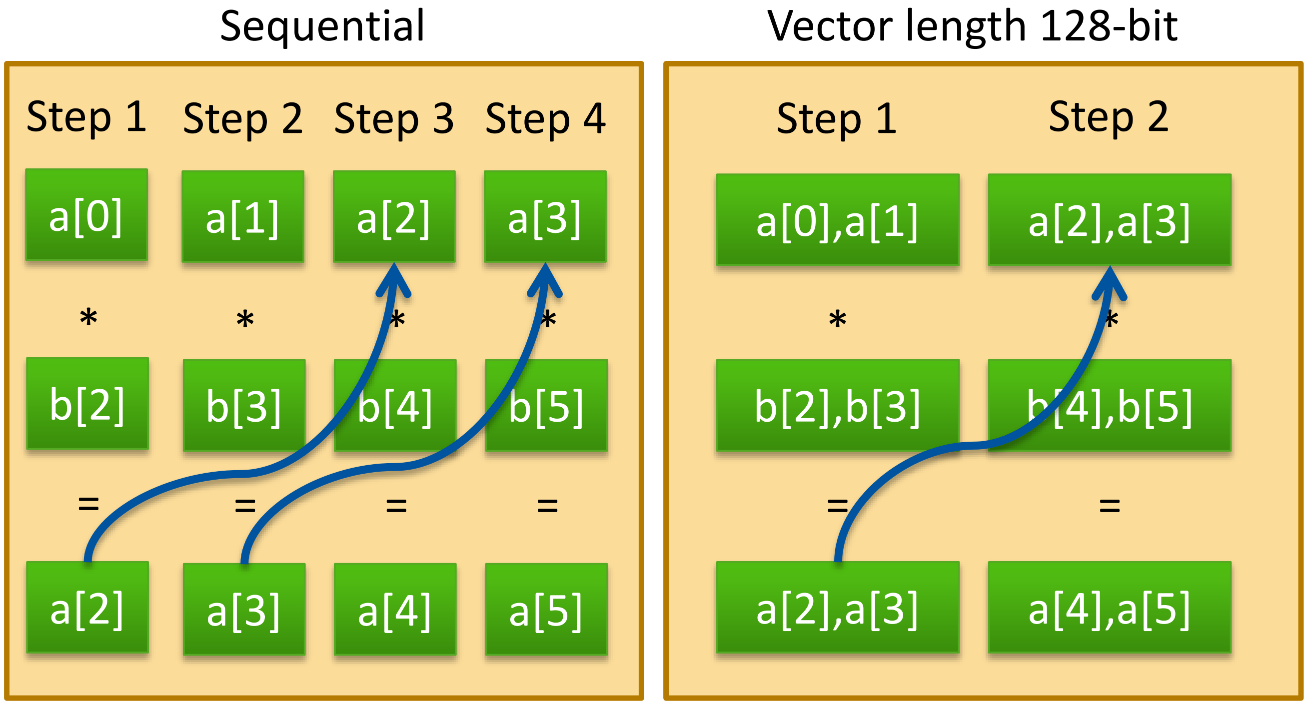
- **where clauses are:**
  - → linear(*list[:linear-step]*), a variable increases linearly in every loop iteration
  - → aligned(*list[:alignment]*), specifies that data is aligned
  - → private(*list*), as usual
  - → lastprivate(*list*) , as usual
  - → reduction(*reduction-identifier:list*) , as usual
  - → 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];
}
```

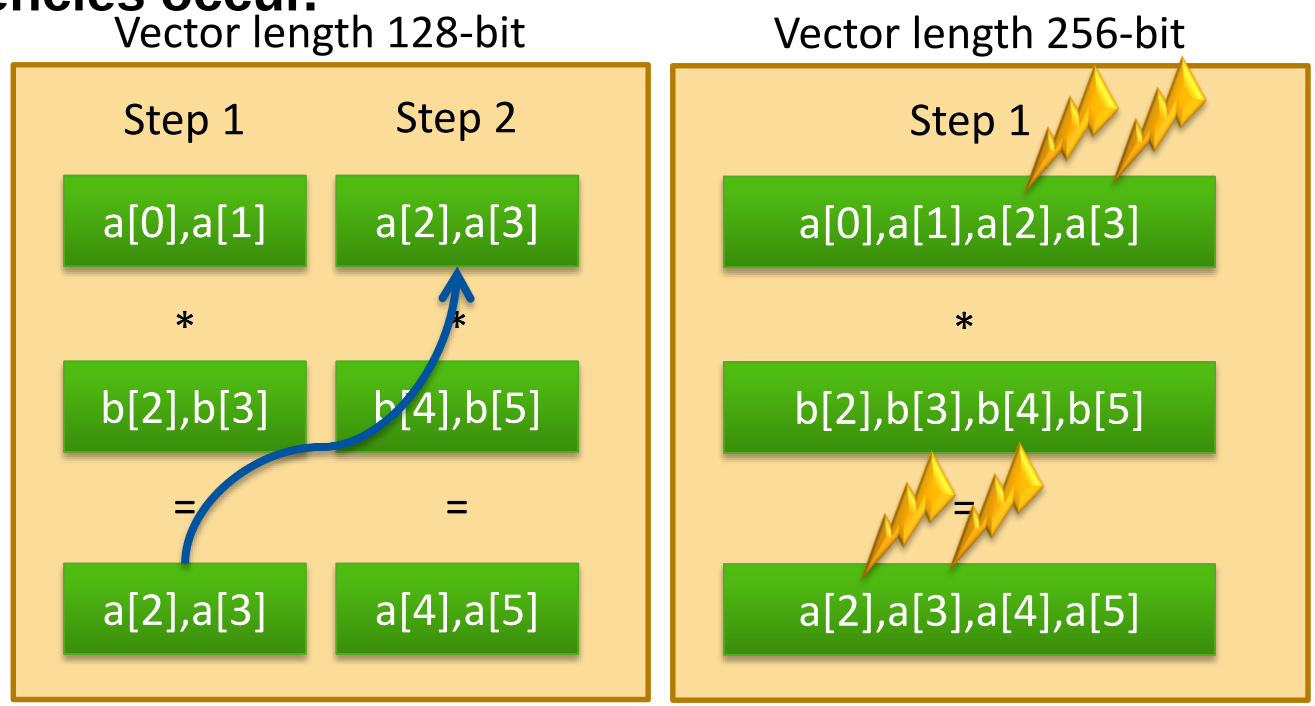


# The SIMD construct

- **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.**

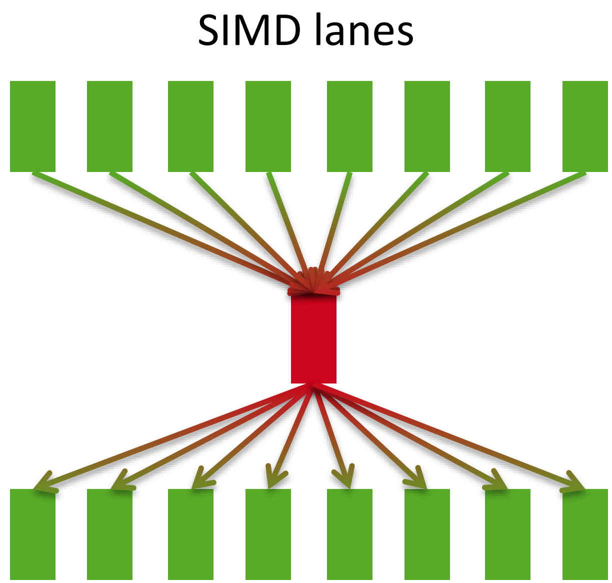
```
for(i=0 ; i < N ; i++)
{

      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

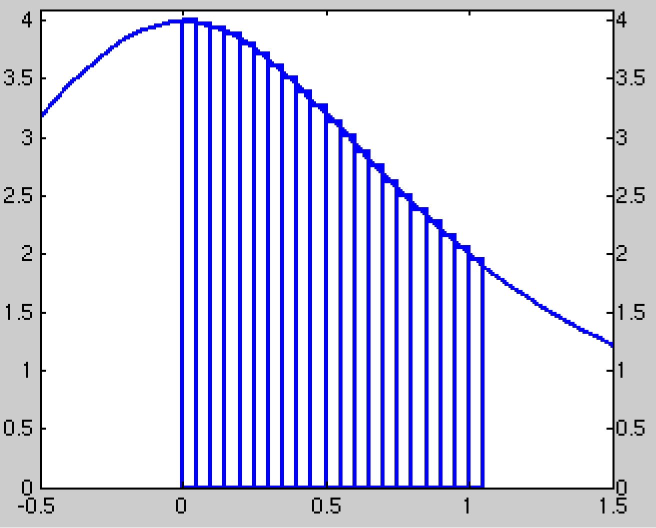
```c
#pragma omp declare simd
double f(double x)
{
   return (4.0 / (1.0 + x*x));
}
```

File: pi.c

```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 on:**
  - → Westmere CPU with SSE (128-bit vectors)
  - → Intel Xeon Phi with AVX-512 (512-bit vectors)

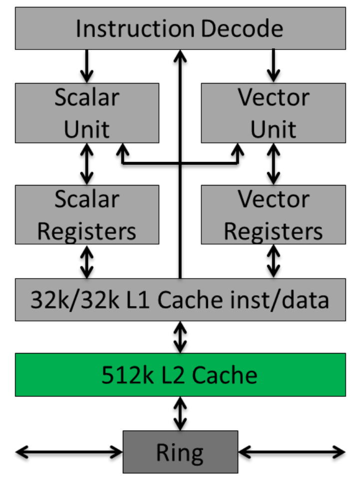
| | Runtime Westmere | Speedup Westmere | Runtime Xeon Phi | Speedup Xeon Phi |
|---|---|---|---|---|
| non vectorized | 1.44 sec | 1 | 16.25 sec | 1 |
| vectorized | 0.72 sec | 2 | 1.82 sec | 8.9 |

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

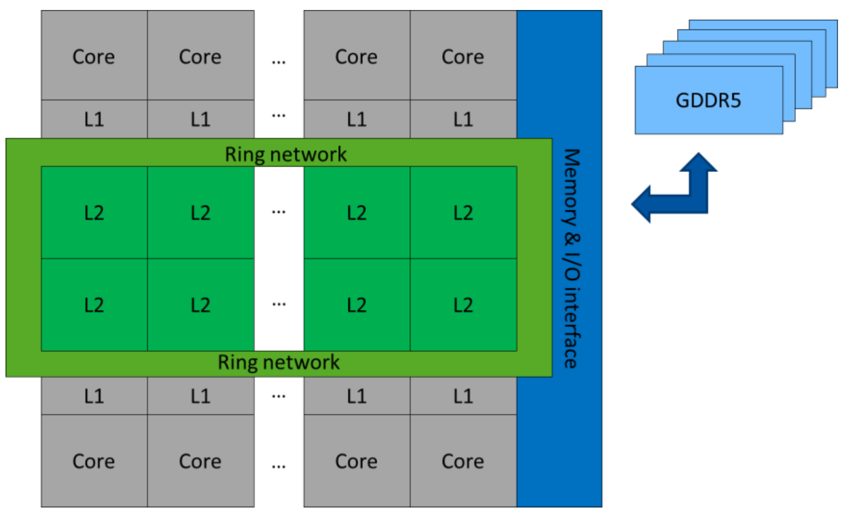
# OpenMP for Accelerators

# Intel Xeon Phi

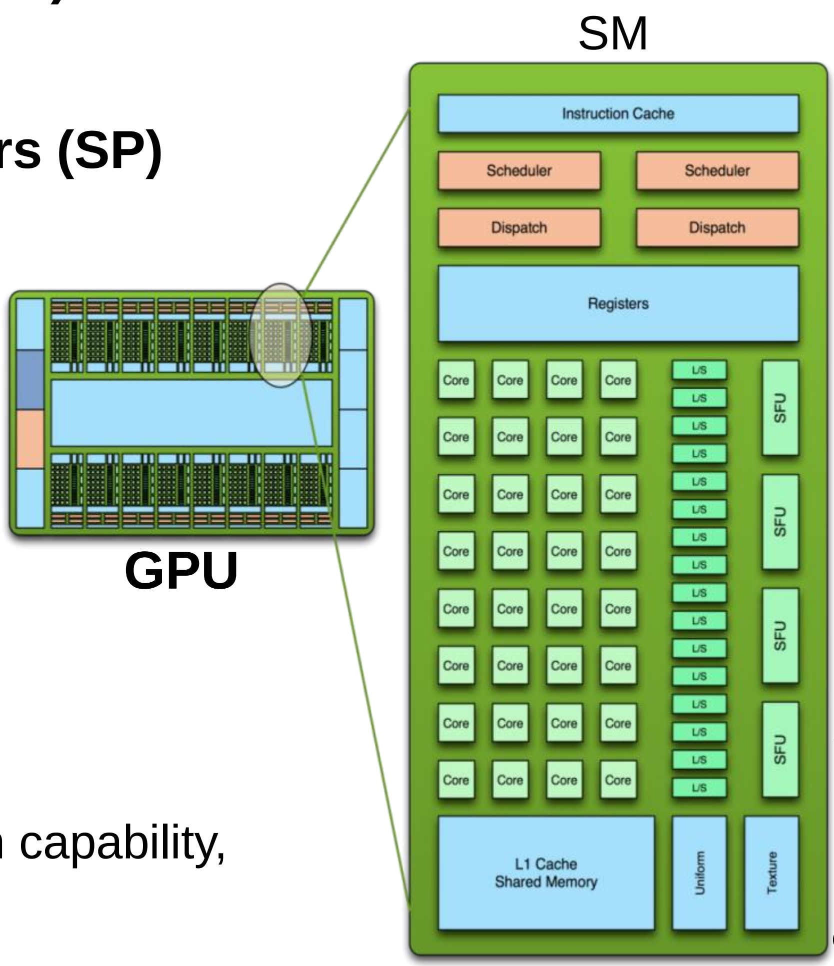
Source: Intel




Intel Xeon Phi Coprocessor
- 1 x Intel Xeon Phi @ 1090 MHz
- 60 Cores (in-order)
- ~ 1 TFLOPS DP Peak
- 4 hardware threads per core (SMT)
- 8 GB GDDR5 memory
- 512-bit SIMD vectors (32 registers)
- Fully-coherent L1 and L2 caches
- Plugged into PCI Express bus



- **3 billion transistors**
- **14-16 streaming multiprocessors (SM)**
  - → Each comprises 32 cores
- **448-512 cores/ streaming processors (SP)**
  - → i.a. Floating point & integer unit
- **Memory hierarchy**





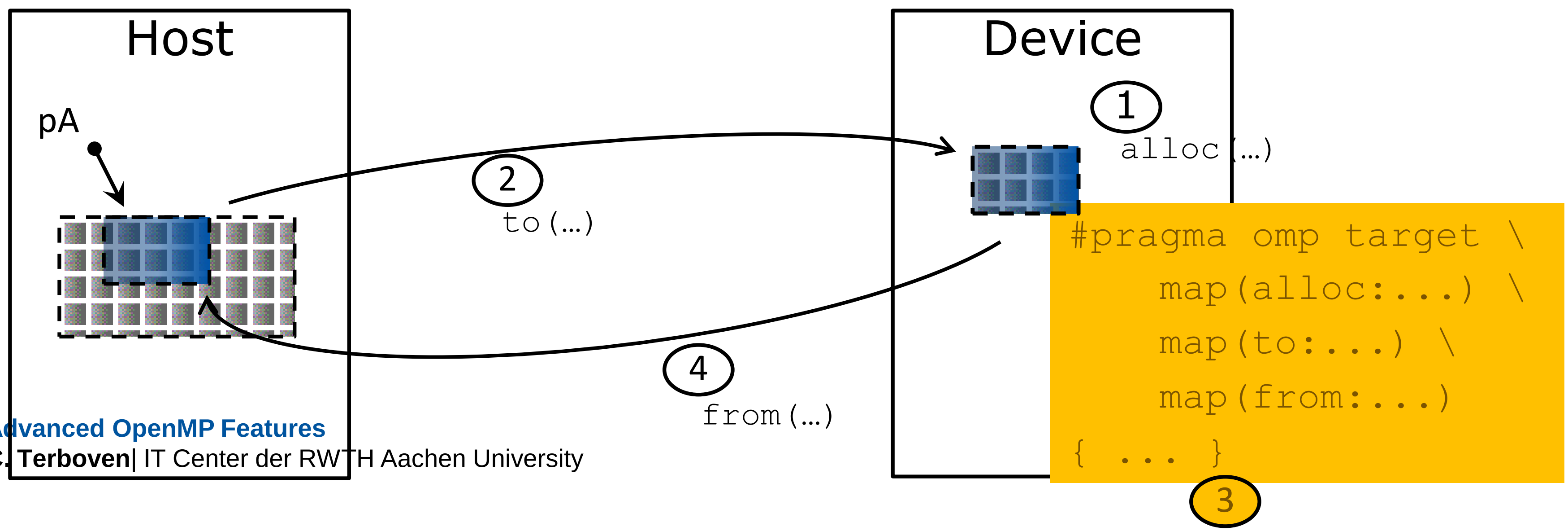

- **Peak performance**
  - → SP: 1.03 TFlops
  - → DP: 515 GFlops
- **ECC support**
- **Compute capability: 2.0**
  - → Defines features, e.g. double precision capability, memory access pattern

## Data environment is lexically scoped

- → Data environment is destroyed at closing curly brace
- → Allocated buffers/data are automatically released

## Use target construct to

- → Transfer control from the host to the device
- → Establish a data environment (if not yet done)
- → Host thread waits until offloaded region completed



# Example: SAXPY

# SAXPY: Serial (Host)

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE



  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }

  // y is needed and modified on the host here

  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }

  free(x); free(y); return 0;
}
```

# SAXPY: OpenMP 4.0 (Intel MIC)

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE
#pragma omp target map(tofrom:y[0:n]) map(to:x[0:n])
#pragma omp parallel for
for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }

  // y is needed and modified on the host here
#pragma omp target map(tofrom:y[0:n]) map(to:x[0:n])
#pragma omp parallel for
  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }
free(x); free(y); return 0;
}
```

# SAXPY: OpenMP 4.0 (Intel MIC)

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE
#pragma omp target data map(to:x[0:n])
{
#pragma omp target map(tofrom:y[0:n])
#pragma omp parallel for
for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }

  // y is needed and modified on the host here
#pragma omp target map(tofrom:y[0:n])
#pragma omp parallel for
  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }
}
  free(x); free(y); return 0;
}
```

# SAXPY: OpenMP 4.0 (NVIDIA GPGPU)

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE
#pragma omp target data map(to:x[0:n])
{
#pragma omp target map(tofrom:y[0:n])
#pragma omp teams
#pragma omp distribute
#pragma omp parallel for
for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }
  // y is needed and modified on the host here
#pragma omp target map(tofrom:y[0:n])
#pragma omp teams
#pragma omp distribute
#pragma omp parallel for
  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }
}
  free(x); free(y); return 0;
}
```

# Target Construct

# Target Data Construct

- **Creates a device data environment for the extent of the region**
  - → when a target data construct is encountered, a new device data environment is created, and the encountering task executes the target data region
  - → when an if clause is present and the if-expression evaluates to false, the device is the host

- **C/C++:**

The syntax of the **target data** construct is as follows:

**#pragma omp target data** *[clause[[,] clause],...] new-line*
*structured-block*

where *clause* is one of the following:

**device(** *integer-expression* **)**
**map(** *[map-type* **:** *] list* **)**
**if(** *scalar-expression* **)**

# Map Clause

- **Map a variable from the current task's data environment to the device data environment associated with the construct**
  - → the list items that appear in a map clause may include array sections
  - → ***alloc***-type: each new corresponding list item has an undefined initial value
  - → ***to***-type: each new corresponding list item is initialized with the original lit item's value
  - → ***from***-type: declares that on exit from the region the corresponding list item's value is assigned to the original list item
  - → ***tofrom***-type: the default, combination of to and from

- **C/C++:**

The syntax of the **map** clause is as follows:

**map(** *[map-type* **:** *] list* **)**

# Target Construct

- **Creates a device data environment and execute the construct on the same device**
  - → superset of the target data constructs - in addition, the target construct specifies that the region is executed by a device and the encountering task waits for the device to complete the target region

- **C/C++:**

The syntax of the **target** construct is as follows:

**#pragma omp target** *[clause[[,] clause],...] new-line*
*structured-block*

where *clause* is one of the following:

**device(** *integer-expression* **)**
**map(** *[map-type* **:** *] list* **)**
**if(** *scalar-expression* **)**

# Example: Target Construct

```
#pragma omp target device(0)
#pragma omp parallel for
  for (i=0; i<N; i++) ...
```

```
#pragma omp target
#pragma omp teams num_teams(8) num_threads(4)
#pragma omp distribute
  for ( k = 0; k < NUM_K; k++ )
  {
    #pragma omp parallel for
    for ( j = 0; j < NUM_J; j++ )
    {
       ...
    }
  }
```

# Target Update Construct

- **Makes the corresponding list items in the device data environment consistent with their original list items, according to the specified motion clauses**

- **C/C++:**

The syntax of the **target update** construct is as follows:

**#pragma omp target update** *motion-clause[, clause[[,] clause],...] new-line*

where *motion-clause* is one of the following:

**to(** *list* **)**
**from(** *list* **)**

and where *clause* is one of the following:

**device(** *integer-expression* **)**
**if(** *scalar-expression* **)**

# Declare Target Directive

- **Specifies that [static] variables, functions (C, C++ and Fortran) and subroutines (Fortran) are mapped to a device**
  - → if a list item is a function or subroutine then a device-specific version of the routines is created that can be called from a target region
  - → if a list item is a variable then the original variable is mapped to a corresponding variable in the initial device data environment for all devices (if the variable is initialized it is mapped with the same value)
  - → all declarations and definitions for a function must have a declare target directive

- **C/C++:**

The syntax of the **declare target** directive is as follows:

**#pragma omp declare target** *new-line*
*declarations-definition-seq*
**#pragma omp end declare target** *new-line*

# Teams Construct (1/2)

- **Creates a league of thread teams where the master thread of each team executes the region**
  - → the number of teams is determined by the num_teams clause, the number of threads in each team is determined by the num_threads clause, within a team region team numbers uniquely identify each team (omp_get_team_num())
  - → once created, the number of teams remeinas constant for the duration of the teams region

- **The teams region is executed by the master thread of each team**
- **The threads other than the master thread to not begin execution until the master thread encounteres a parallel region**

- **Only the following constructs can be closely nested in the team region: distribute, parallel, parallel loop/for, parallel sections and parallel workshare**

# Teams Construct (2/2)

- **A teams construct must be contained within a target construct, which must not contain any statements or directives outside of the teams construct**
- **After the teams have completed execution of the teams region, the encountering thread resumes execution of the enclosing target region**

- **C/C++:**

The syntax of the **teams** construct is as follows

**#pragma omp teams** *[clause[[,] clause],...] new-line*
*structured-block*

where *clause* is one of the following:

**num_teams(** *integer-expression* **)**
**num_threads(** *integer-expression* **)**
**default(shared | none)**
**private(** *list* **)**
**firstprivate(** *list* **)**
**shared(** *list* **)**
**reduction(** *operator* **:** *list* **)**

# Distribute Construct

- **Specifies that the iteration of one or more loops will be executed by the thread teams, the iterations are distributed across the master threads of all teams**
  - → there is no implicit barrier at the end of a distribute construct
  - → a distribute construct must be closely nested in a teams region

- **C/C++:**

The syntax of the **distribute** construct is as follows:

**#pragma omp distribute** *[clause[[,] clause],...] new-line*
*for-loops*

Where *clause* is one of the following:

**private(** *list* **)**
**firstprivate(** *list* **)**
**collapse(** *n* **)**
**dist_schedule(** *kind[, chunk_size]* **)**

All associated for-loops must have the canonical form described in Section 2.5.

# Questions?