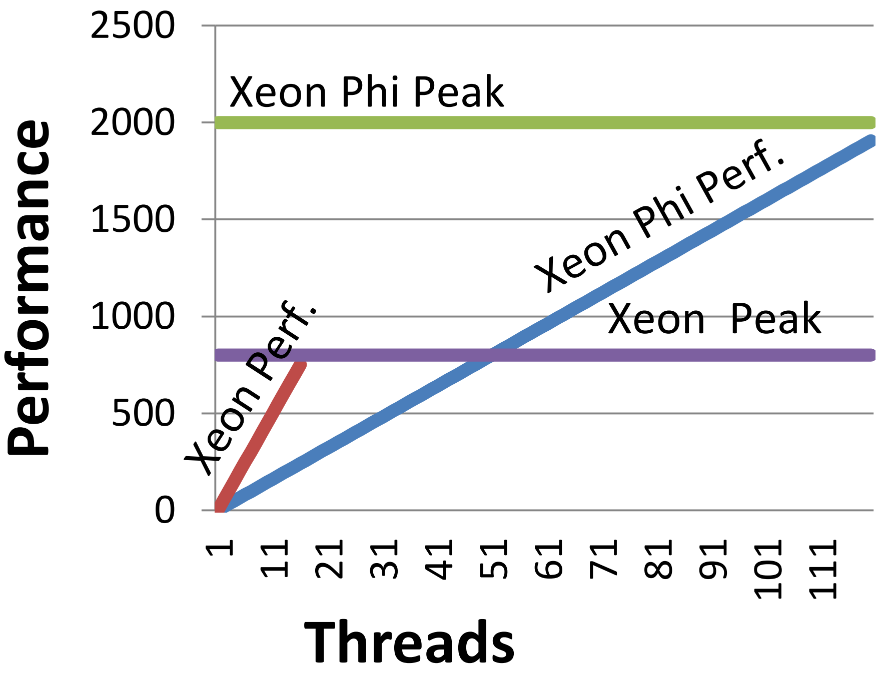
Programming Accelerators with OpenMP

Dirk Schmidl
IT Center, RWTH Aachen University
Member of the HPC Group
schmidl@itc.rwth-aachen.de

Christian Terboven
IT Center, RWTH Aachen University
Deputy lead of the HPC Group
terboven@itc.rwth-aachen.de

Motivation

- **Demand for more compute power**
- **Reach higher performance with more threads**
- **Power consumption: Better performance / watt ratio**
- **GPUs are one alternative, but: CUDA is hard to learn / program**
- **Intel Xeon Phi can be programmed with established programming paradigms like OpenMP, MPI, Pthreads**

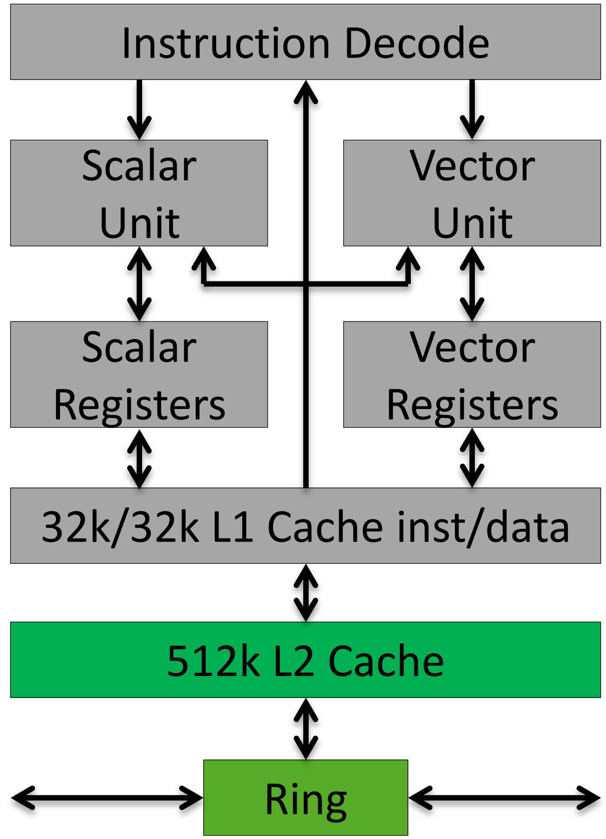


Source: James Reinders, Intel

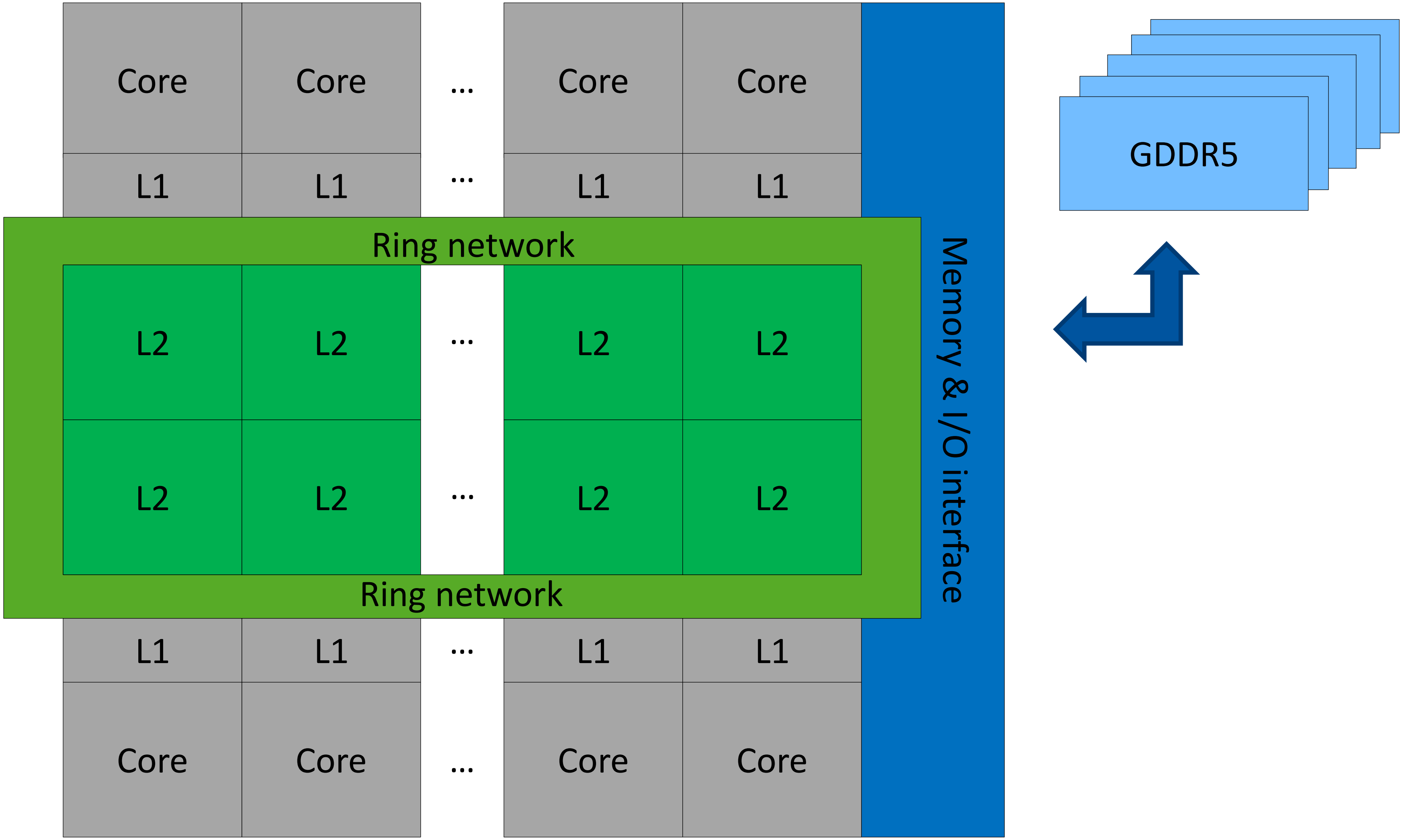
Architecture (1/2)

Source: Intel

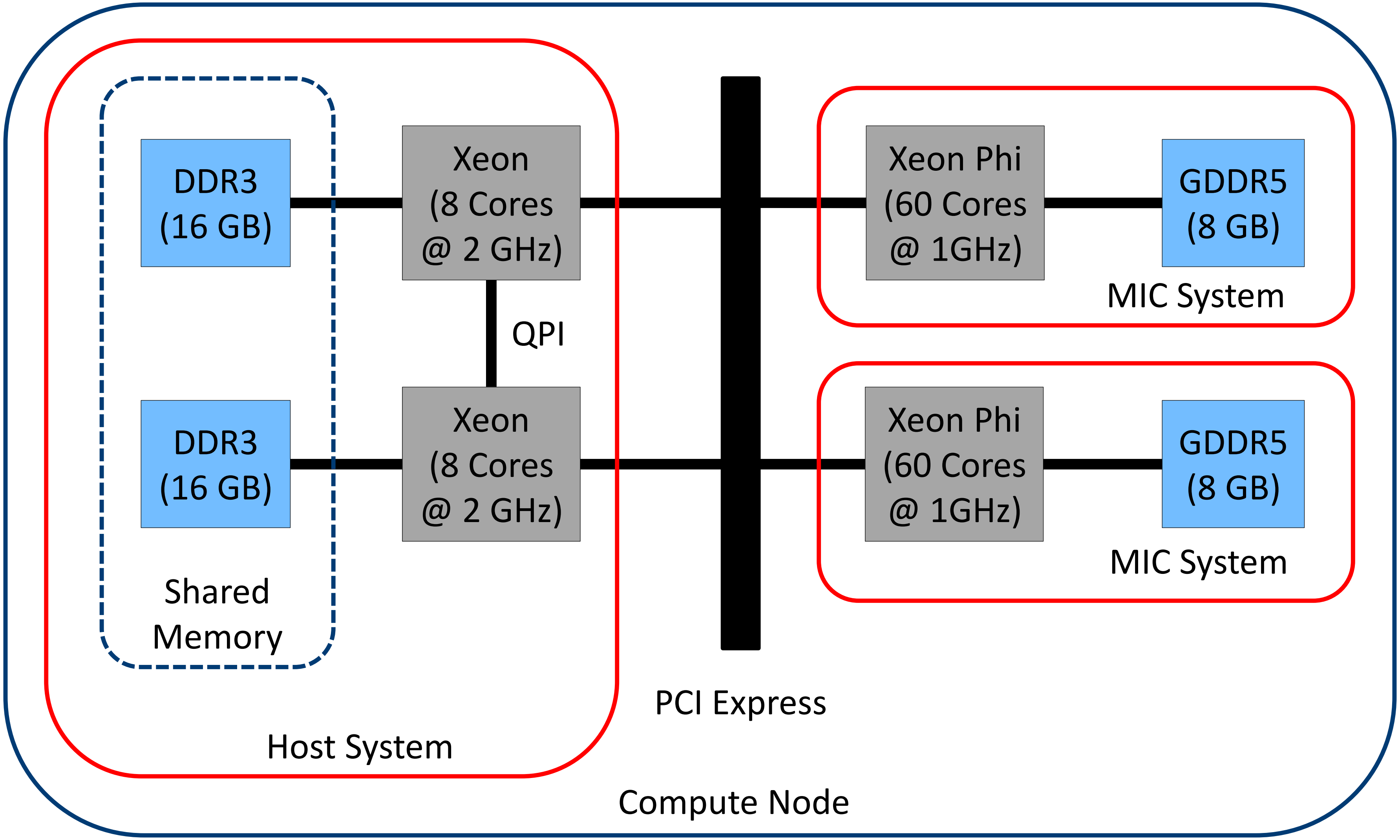
Intel Xeon Phi Coprocessor
- 1 x Intel Xeon Phi @ 1090 MHz
- 60 Cores (in-order)
- ~ 1 TFLOPS DP Peak
- 4 hardware threads per core (SMT)
- 8 GB GDDR5 memory
- 512-bit SIMD vectors (32 registers)
- Fully-coherent L1 and L2 caches
- Plugged into PCI Express bus



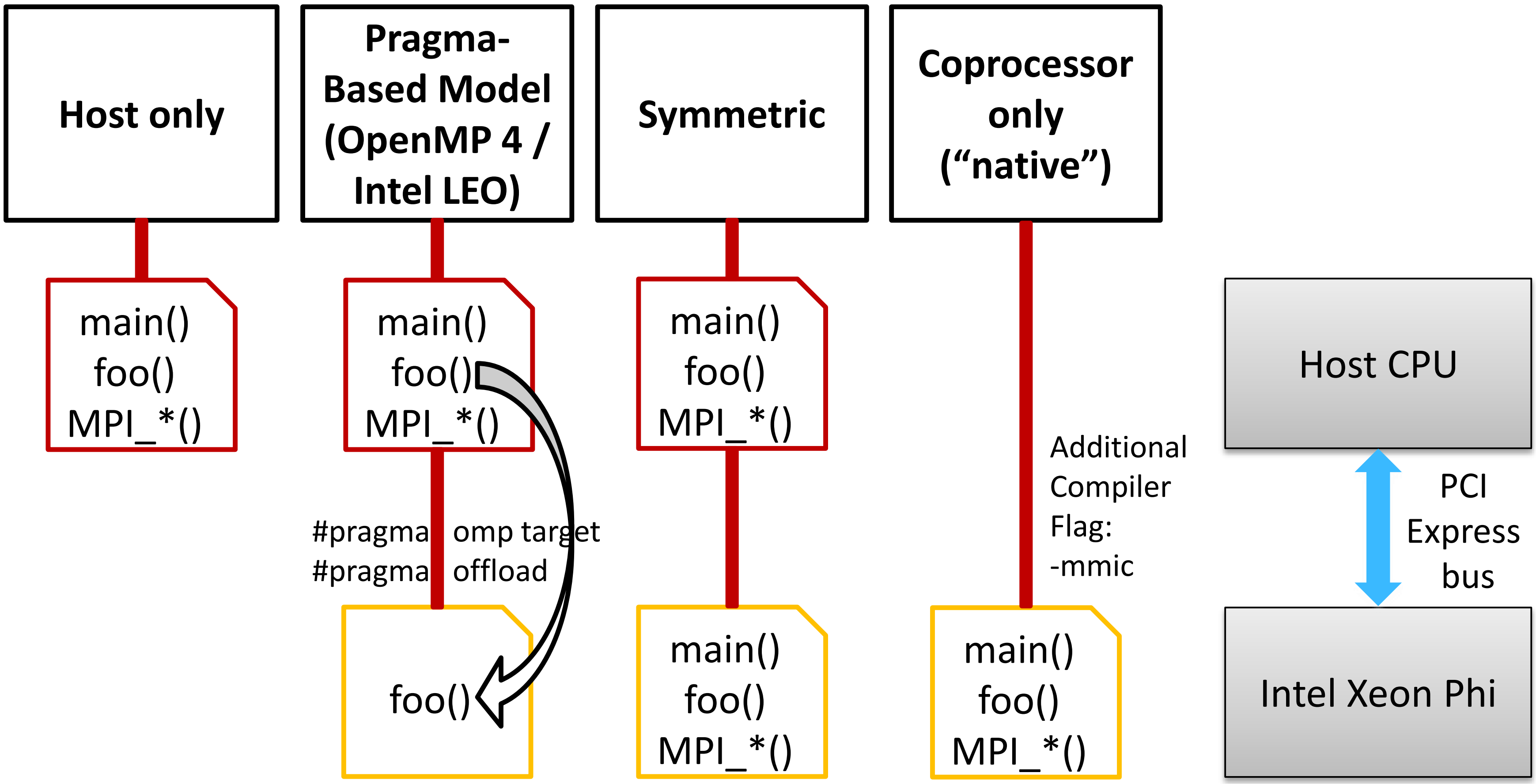
Architecture (2/2)



Xeon Phi Nodes at RWTH Aachen



Programming Models



Native/Symmetric Execution

- **The Xeon Phi is x86 based, but <u>not</u> binary compatible to the host**
  - → recompile the application with "-mmic" to create an executable for Xeon Phi coprocessors
  - → For native execution:
    - → login via ssh
    - → start the executable
  - → For MPI (symmetric) execution:
    - → Compile two executables, one for the host and one for the coprocessor with a certain suffix, e.g. ".mic"
    - → Set "I_MPI_MIC=enabled" and "I_MPI_MIC_POSTFIX=.mic"
    - → Start the executable with mpiexec on host and coprocessor

Performance Expectations

- **Comparison with our MPI nodes (2 x Intel Westmere CPUs)**

	Xeon Phi 5110P	2 x Xeon X5675
Frequency [GHz]	1.09	3.07
Cores	60	12
Peak Performance [GFLOPS]	1171	144
Peak Performance (no SIMD) [GFLOPS]	146	72
Memory Bandwidth (STREAM) [GB/s]	160	40
Memory Bandwidth (single core) [GB/s]	5	12

→ Only scalable, vectorizable codes benefit of the Xeon Phi at the moment

→ Performance is not for free, tuning will be necessary

OpenMP for Accelerators

Agenda

- **What is an Accelerator?**
- **Execution and Data Model**
- **Target Construct**
- **Example: SAXPY**
- **Live Experiment: PI**
- **Outlook: what will come next?**

What is an Accelerator?

in OpenMP

What kind of devices shall be supported?

- **In how differs an accelerator from just another core?**
  → different functionality, i.e. optimized for something special
  → different (possibly limited) instruction set
  → heterogeneous device
- **Assumptions used as design goals for OpenMP 4.0:**
  → every accelerator device is attached to one host device
  → it is probably heterogeneous
  → it may not be programmable in the same language as the host, or it may not implement all operations available on the host
  → it may or may not share memory with the host device
  → some accelerators are specialized for loop nests

Execution and Data Model

Execution Model

- **Host-centric: the execution of an OpenMP program starts on the *host device* and it may offload *target regions* to *target devices***
  - → In principle, a target region also begins as a single thread of execution: when a target construct is encountered, the target region is executed by the implicit device thread and the encountering thread/task [on the host] waits at the construct until the execution of the region completes

- **If a target device is not present, or not supported, or not available, the target region is executed by the host device**

- **If a construct creates a *data environment*, the data environment is created at the time the construct is encountered**

Data Model

- **When an OpenMP program begins, each device has an initial *device data environment***

- **Directives accepting data-mapping attribute clauses determine how an *original* variable is mapped to a *corresponding* variable in a device data environment**

  →original: the variable on the host

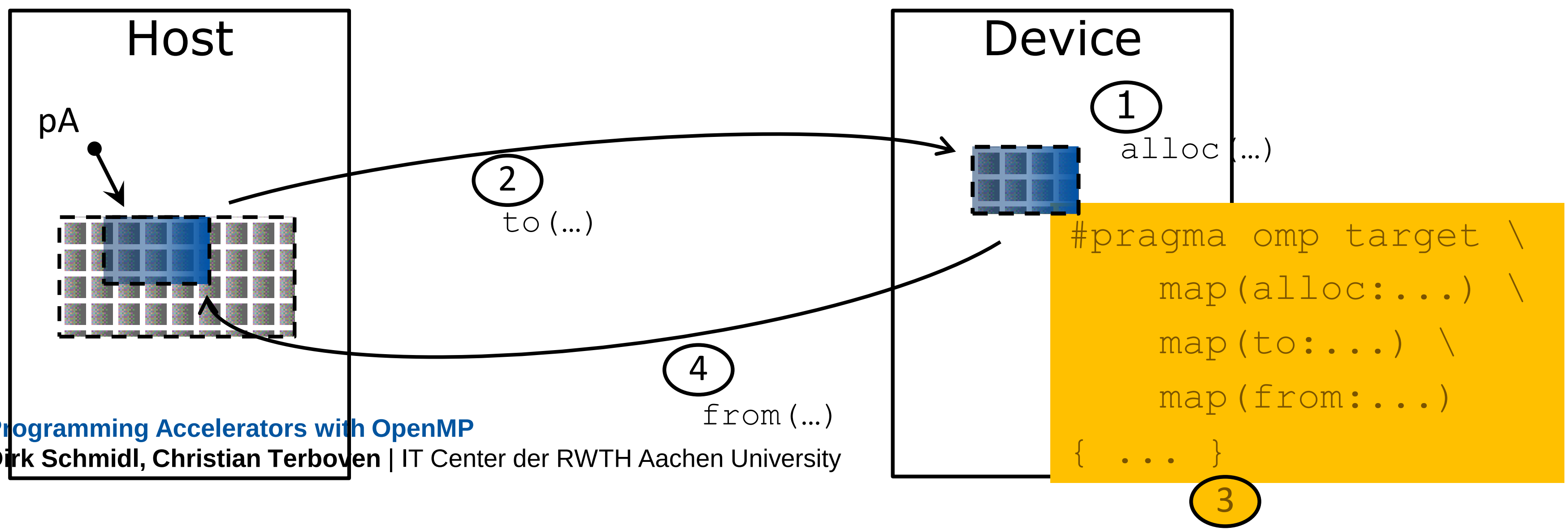
  →corresponding: the variable on the device

  →the corresponding variable in the device data environment may share storage with the original variable (danger of data races)

- **If a corresponding variable is present in the enclosing device data environment, the new device data environment inherits the corresponding variable from the enclosing device**

Execution + Data Model

- **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




```
#pragma omp target \
    map(alloc:...) \
    map(to:...) \
    map(from:...)
{ ... }
```

Device constructs

```
#pragma omp target data device(0) map(alloc:tmp[:N])
                    map(to:input[:N]) map(from:res)
  {
#pragma omp target device(0)
#pragma omp parallel for
    for (i=0; i<N; i++)
      tmp[i] = some_computation(input[i], i);

    do_some_other_stuff_on_host();

#pragma omp target device(0)
#pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
      res += final_computation(tmp[i], i)
  }
```

data region

shaping and slicing

offload

host

target

host

target

host

- **This is a very simple example, but you see functionality similar to OpenACC 1.0. Device support has to come with the implementation.**

Example: Execution and Data Model

- **Environment Variable OMP_DEFAULT_DEVICE=<int>: sets the device number to use in target constructs**

```
double B[N] = ...; // some initialization
#pragma omp target map(tofrom:B)
#pragma omp parallel for
for (i=0; i<N; i++)
   B[i] += sin(B[i]);
```

→ map variable B to device, then execute parallel region on device, works probably pretty well on Intel Xeon Phi

```
double B[N] = ...; // some initialization
#pragma omp target map(tofrom:B)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
#pragma omp distribute
for (i=0; i<N; i += num_blocks)
   #pragma omp parallel for
   for (b = i; b < i+bsize; b++)
      B[b] += sin(B[b]);
```

→ same as above, but code probably better optimized for NVIDIA GPGPUs

Comparing OpenMP with OpenACC

- **OpenMP 4.0 – for Intel Xeon Phi:**

```
#pragma omp target device(0) map(tofrom:B)
#pragma omp parallel for
for (i=0; i<N; i++)
   B[i] += sin(B[i]);
```

- **OpenMP 4.0 – for NVIDIA GPGPU:**

```
#pragma omp target device(0) map(tofrom:B)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
#pragma omp distribute
for (i=0; i<N; i += num_blocks)
   #pragma omp parallel for
   for (b = i; b < i+num_blocks; b++)
      B[b] += sin(B[b]);
```

Also possible (and equivalent) in OpenMP 4.0:

Combined directive

#pragma omp teams distribute parallel for

- **OpenACC – for NVIDIA GPGPU:**

```
#pragma acc parallel copy(B[0:N]) num_gangs(numblocks)\
            vector_length(bsize)
#pragma acc loop gang vector
  for (i=0; i<N; ++i) {
    B[i] += sin(B[i]);
  }
```

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* **)**

Note: map clause does not work correctly with Intel 14.0 compilers

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*

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++ )
    {
       ...
    }
  }
```

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**

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.

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* **)**

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: OpenACC v2 (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 acc data copyin(x[0:n])
{

#pragma acc parallel copy(y[0:n])
#pragma acc loop
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }
  // y is needed and modified on the host here
#pragma acc parallel copy(y[0:n])
#pragma acc loop
  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;
}
```

Live Experiment: PI

Host version

```
double f(double x) {
   return (double)4.0 / ((double)1.0 + (x*x));
}

void computePi() {
   double h = (double)1.0 / (double)iNumIntervals;
   double sum = 0, x;


#pragma omp parallel for private(x) reduction(+:sum)
   for (int i = 1; i <= iNumIntervals; i++) {
      x = h * ((double)i - (double)0.5);
      sum += f(x);
   }
   myPi = h * sum;
}
```

Accelerator version (Intel Xeon Phi)

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

void computePi() {
   double h = (double)1.0 / (double)iNumIntervals;
   double sum = 0, x;

#pragma omp target
#pragma omp parallel for private(x) reduction(+:sum)
   for (int i = 1; i <= iNumIntervals; i++) {
      x = h * ((double)i - (double)0.5);
      sum += f(x);
   }
   myPi = h * sum;
}
```

PI on Host and Accelerator

Outlook

Accelerators – current work

- **OpenMP 4.0 introduced the `target` construct for device programming, i.e. compute accelerators – some aspects were missing / undefined:**
  - → combinations of constructs, `#pragma omp target teams distribute parallel for simd`
  - → concurrent mapping of the same variable by two threads – still undefined

- **Current Plans for OpenMP 4.1:**
  - → async execution: currently the host waits for `target` operation to complete
    - → `task target` construct: asyn offload, host will continue execution
    - → dependencies between host and device: adding `target` will maintain them
  - → unstructured data movement: sometimes data regions are too static
    - → `target enter data` / `exit data` constructs to push / pull at encountering
  - → open discussion on support for multiple devices (i.e. work distribution)

Questions?