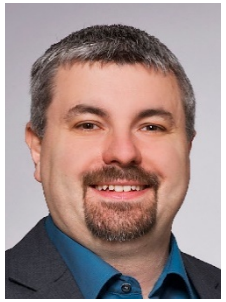
Programming OpenMP

Christian Terboven

Michael Klemm



Agenda (in total 7 Sessions)

- Session 1: OpenMP Introduction
- Session 2: Tasking
- Session 3: Optimization for NUMA and SIMD
- Session 4: What Could Possibly Go Wrong Using OpenMP
- **Session 5**: Introduction to **Offloading with OpenMP**
  - →Review of Session 3 / homework assignments
  - →OpenMP device and execution model
  - →Offload basics and exploiting parallelism
  - →Optimizing data transfers
  - →Homework assignments ☺
- Session 6: Advanced Offloading Topics
- Session 7: Selected / Remaining Topics

Programming OpenMP

Review

Christian Terboven
Michael Klemm

Questions?

PI

Example solution: PI w/ SIMD

```
#pragma omp simd private(fX) reduction(+:fSum)
for (i = 0; i < n; i += 1)
{
    fX = fH * ((double)i + 0.5);
    fSum += f(fX);
}
return fH * fSum;
```

Jacobi

Example solution: Jacobi opt. for NUMA / 1

```
/* Initilize initial condition and RHS */
 #pragma omp parallel for private(i, xx, yy) // or collapse(2) instead of private(i)
 for (j=0; j<m; j++){
   for (i=0; i<n; i++){
     xx = -1.0 + *dx * (i-1);
     yy = -1.0 + *dy * (j-1);
     U(j,i) = 0.0;
     F(j,i) = -alpha * (1.0 - xx*xx) * (1.0 - yy*yy)
               - 2.0 * (1.0 - xx*xx) - 2.0 * (1.0 - yy*yy);
   }
 }
```

Example solution: Jacobi opt. for NUMA / 2

```
#pragma omp parallel
   {

   /* copy new solution into old */
   #pragma omp for private(i) // or collapse(2) instead of private(i)
   for (j=0; j<m; j++)
     for (i=0; i<n; i++){
       UOLD(j,i) = U(j,i);
     }

   /* compute stencil, residual and update */
   #pragma omp for private(i, resid) reduction(+:error) // or collapse(2)
   for (j=1; j<m-1; j++){
     for (i=1; i<n-1; i++){
       resid =(   /* left out for brevity */ ) / b;

       /* update solution */
       U(j,i) = UOLD(j,i) - omega * resid;

       /* accumulate residual error */
       error =error + resid*resid;
     }
   }

   }
```

Introduction to Offloading with OpenMP

OpenMP device and execution model

Running Example for this Presentation: saxpy

OpenMP

```
void saxpy() {
    float a, x[SZ], y[SZ];
    // left out initialization
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp parallel for firstprivate(a)
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

Timing code (not needed, just to have a bit more code to show ☺)

This is the code we want to execute on a target device (i.e., GPU)

Timing code (not needed, just to have a bit more code to show ☺)

Running Example for this Presentation: saxpy

```
void saxpy() {
    float a, x[SZ], y[SZ];
    // left out initialization
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp parallel for firstprivate(a)
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

Timing code (not needed, just to have a bit more code to show ☺)

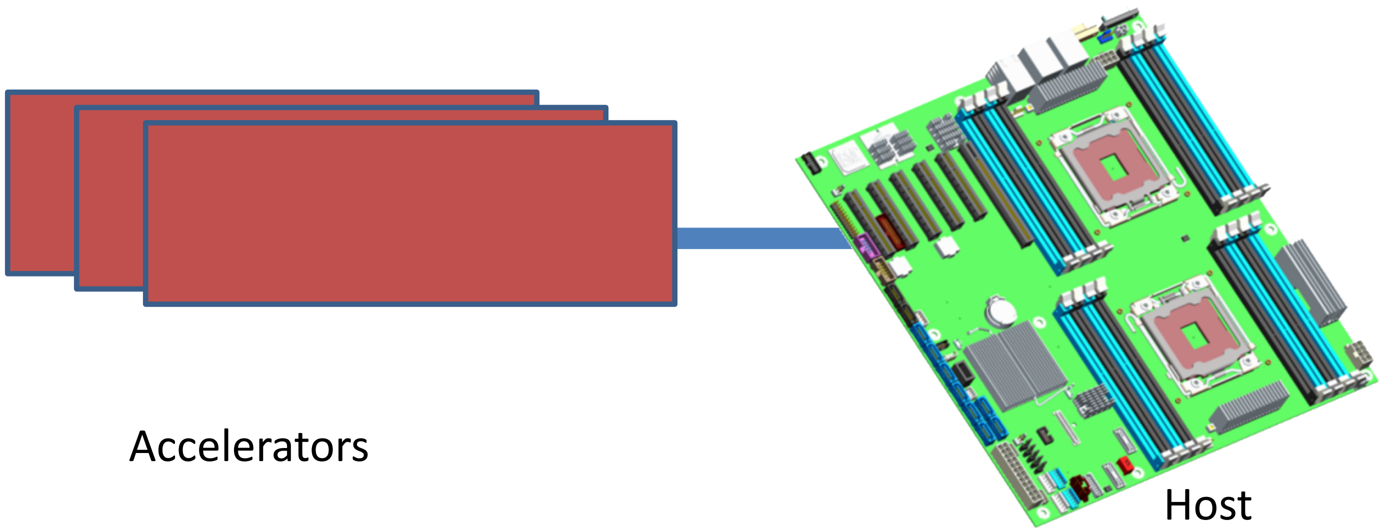
This is the code we want to execute on a target device (i.e., GPU)

Timing code (not needed, just to have a bit more code to show ☺)

Don't do this at home!
Use a BLAS library for this!

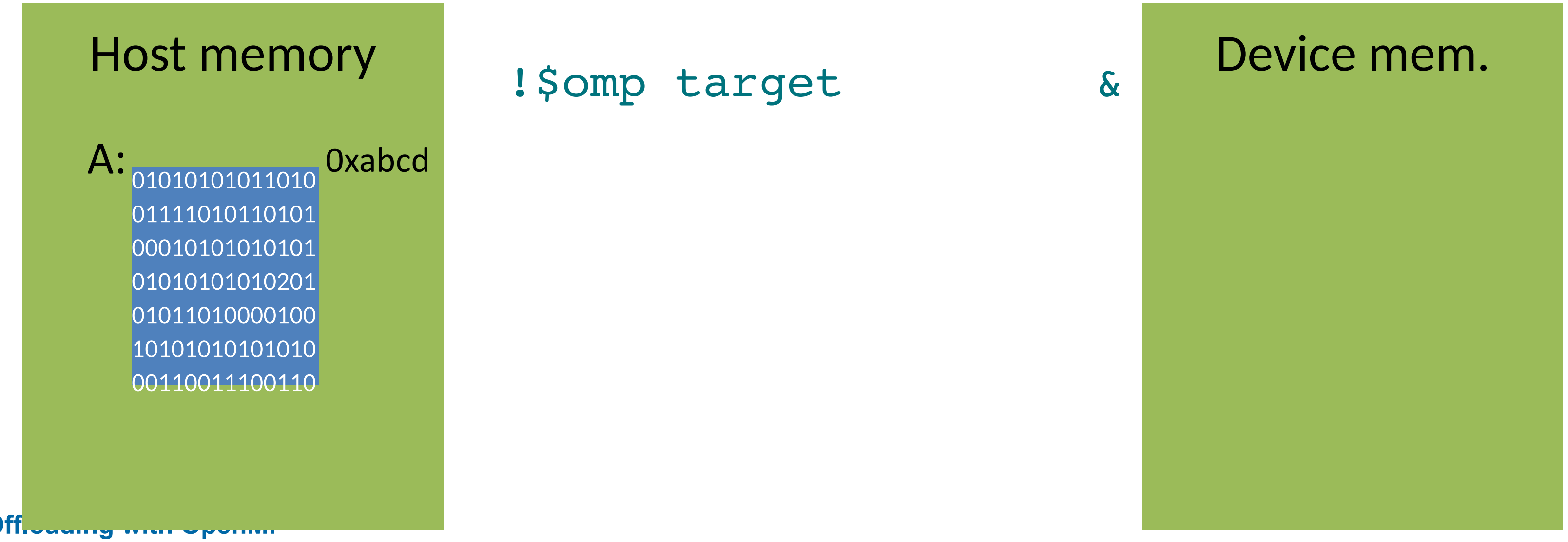
Device Model

- As of version 4.0 the OpenMP API supports accelerators/coprocessors
- Device model:
  → One host for “traditional” multi-threading
  → Multiple accelerators/coprocessors of the same kind for offloading



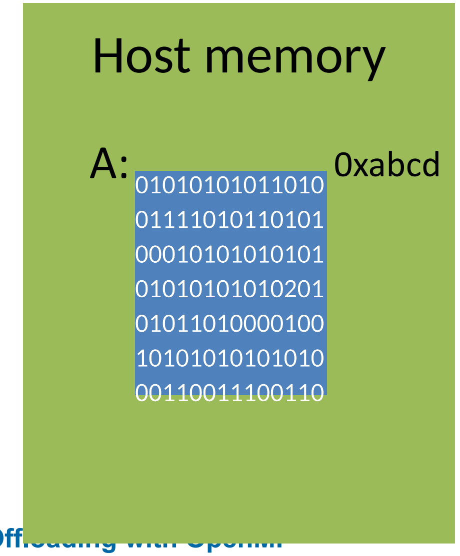
OpenMP Execution Model for Devices

- Offload region and its data environment are bound to the lexical scope of the construct
  - →Data environment is created at the opening curly brace
  - →Data environment is automatically destroyed at the closing curly brace
  - →Data transfers (if needed) are done at the curly braces, too:
    - →Upload data from the host to the target device at the opening curly brace.
    - →Download data from the target device at the closing curly brace.

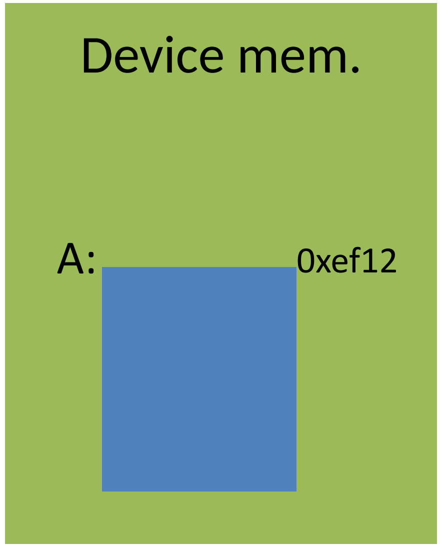


OpenMP Execution Model for Devices

- Offload region and its data environment are bound to the lexical scope of the construct
  - →Data environment is created at the opening curly brace
  - →Data environment is automatically destroyed at the closing curly brace
  - →Data transfers (if needed) are done at the curly braces, too:
    - →Upload data from the host to the target device at the opening curly brace.
    - →Download data from the target device at the closing curly brace.

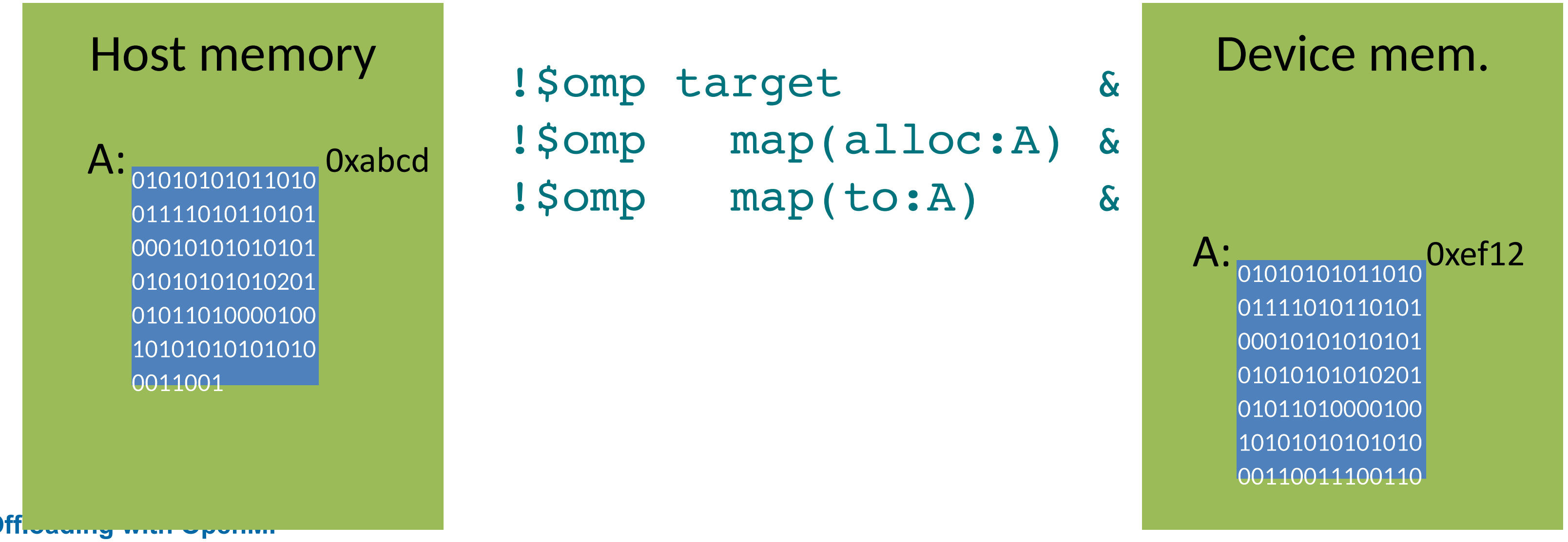



```
!$omp target          &
!$omp   map(alloc:A)  &
```



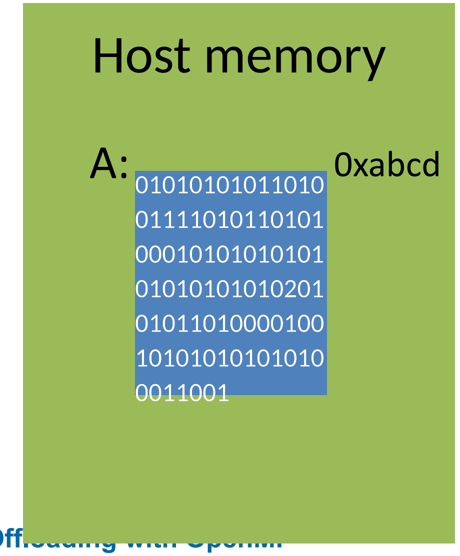
OpenMP Execution Model for Devices

- Offload region and its data environment are bound to the lexical scope of the construct
  - →Data environment is created at the opening curly brace
  - →Data environment is automatically destroyed at the closing curly brace
  - →Data transfers (if needed) are done at the curly braces, too:
    - →Upload data from the host to the target device at the opening curly brace.
    - →Download data from the target device at the closing curly brace.

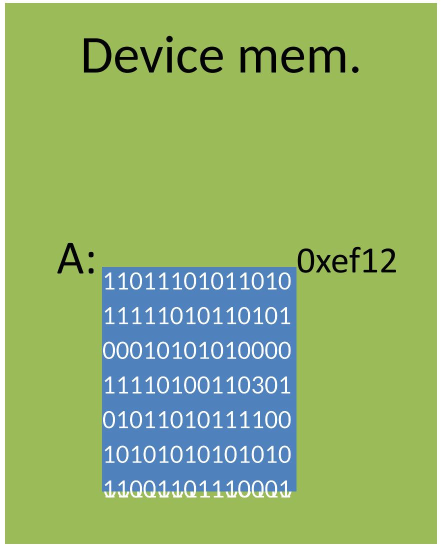


OpenMP Execution Model for Devices

- Offload region and its data environment are bound to the lexical scope of the construct
  - →Data environment is created at the opening curly brace
  - →Data environment is automatically destroyed at the closing curly brace
  - →Data transfers (if needed) are done at the curly braces, too:
    - →Upload data from the host to the target device at the opening curly brace.
    - →Download data from the target device at the closing curly brace.

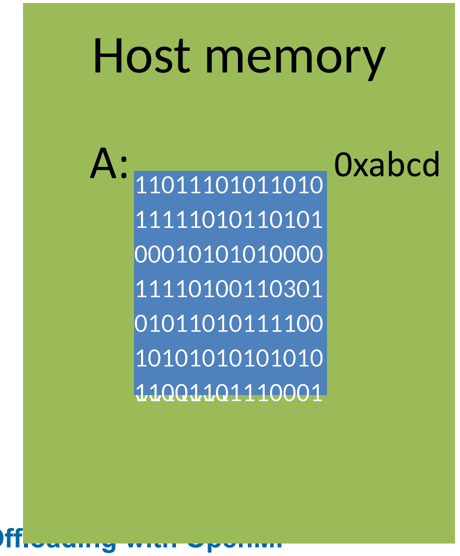



```
!$omp target           &
!$omp   map(alloc:A)   &
!$omp   map(to:A)      &
!$omp   map(from:A)    &
```

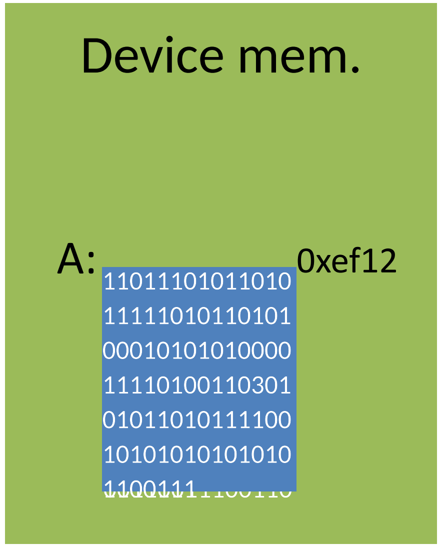


OpenMP Execution Model for Devices

- Offload region and its data environment are bound to the lexical scope of the construct
  - → Data environment is created at the opening curly brace
  - → Data environment is automatically destroyed at the closing curly brace
  - → Data transfers (if needed) are done at the curly braces, too:
    - → Upload data from the host to the target device at the opening curly brace.
    - → Download data from the target device at the closing curly brace.

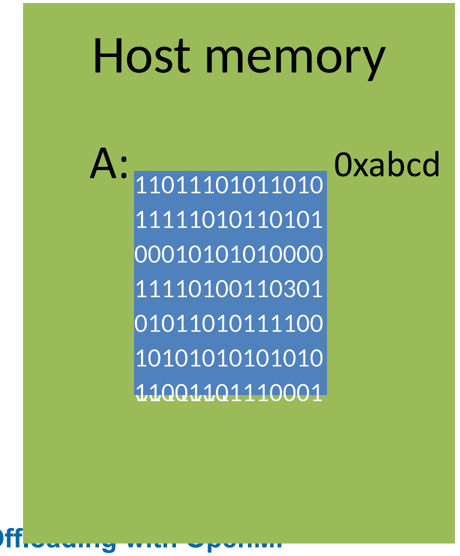



```
!$omp target              &
!$omp   map(alloc:A)  &
!$omp   map(to:A)      &
!$omp   map(from:A)   &
   call compute(A)
```

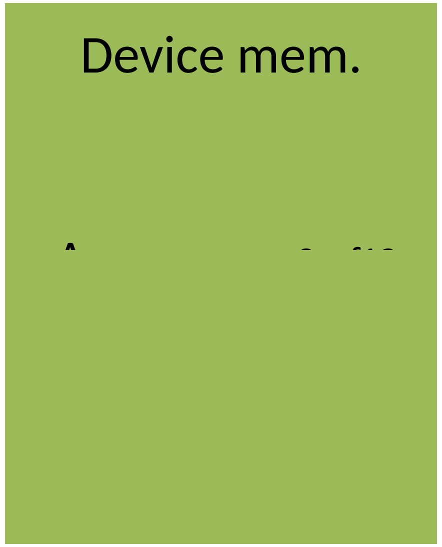


OpenMP Execution Model for Devices

- Offload region and its data environment are bound to the lexical scope of the construct
  - →Data environment is created at the opening curly brace
  - →Data environment is automatically destroyed at the closing curly brace
  - →Data transfers (if needed) are done at the curly braces, too:
    - →Upload data from the host to the target device at the opening curly brace.
    - →Download data from the target device at the closing curly brace.

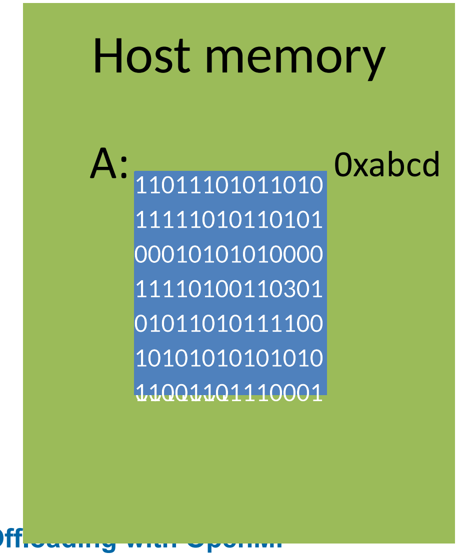



```
!$omp target              &
!$omp   map(alloc:A)  &
!$omp   map(to:A)        &
!$omp   map(from:A)    &
    call compute(A)
```

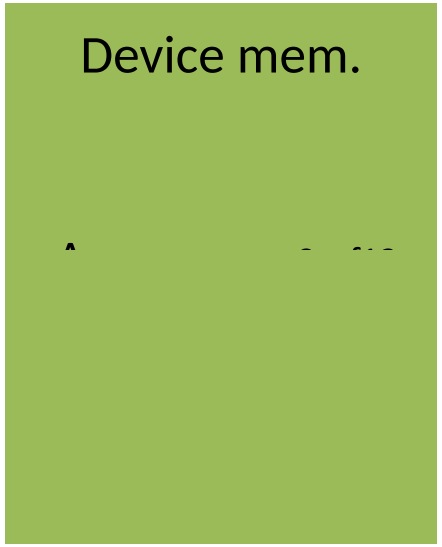


OpenMP Execution Model for Devices

- Offload region and its data environment are bound to the lexical scope of the construct
  - →Data environment is created at the opening curly brace
  - →Data environment is automatically destroyed at the closing curly brace
  - →Data transfers (if needed) are done at the curly braces, too:
    - →Upload data from the host to the target device at the opening curly brace.
    - →Download data from the target device at the closing curly brace.




```
!$omp target            &
!$omp   map(alloc:A)    &
!$omp   map(to:A)       &
!$omp   map(from:A)     &
   call compute(A)
!$omp end target
```



OpenMP for Devices - Constructs

- Transfer control and data from the host to the device
- Syntax (C/C++)

```
#pragma omp target [clause[[,] clause],…]
structured-block
```

- Syntax (Fortran)

```
!$omp target [clause[[,] clause],…]
structured-block
!$omp end target
```

- Clauses

```
device(scalar-integer-expression)
map([{alloc | to | from | tofrom}:] list)
if(scalar-expr)
```

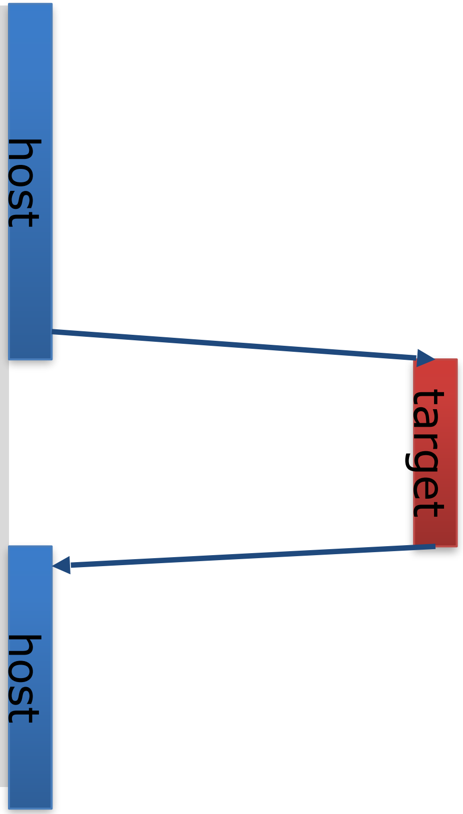
Example: saxpy

```
void saxpy() {
    float a, x[SZ], y[SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

```
clang -fopenmp --offload-arch=gfx90a ...
```

Example: saxpy

```
void saxpy() {
    float a, x[SZ], y[SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

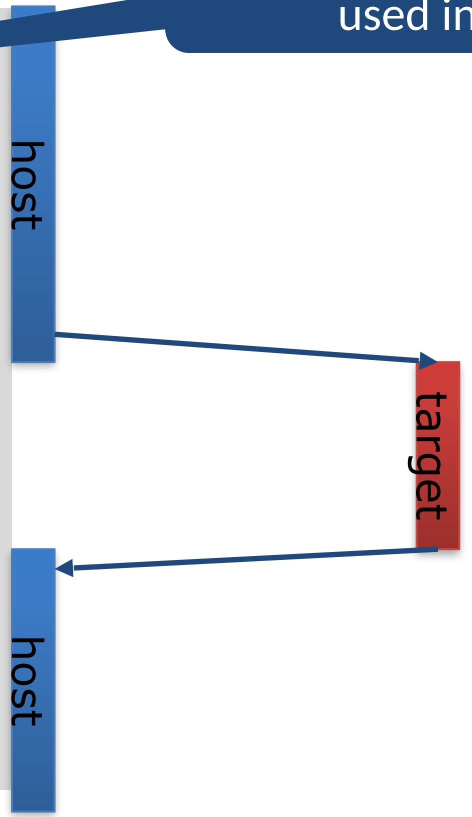



```
clang -fopenmp --offload-arch=gfx90a ...
```

Example: saxpy

The compiler identifies variables that are used in the `target` region.

```
void saxpy() {
    float a, x[SZ], y[SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

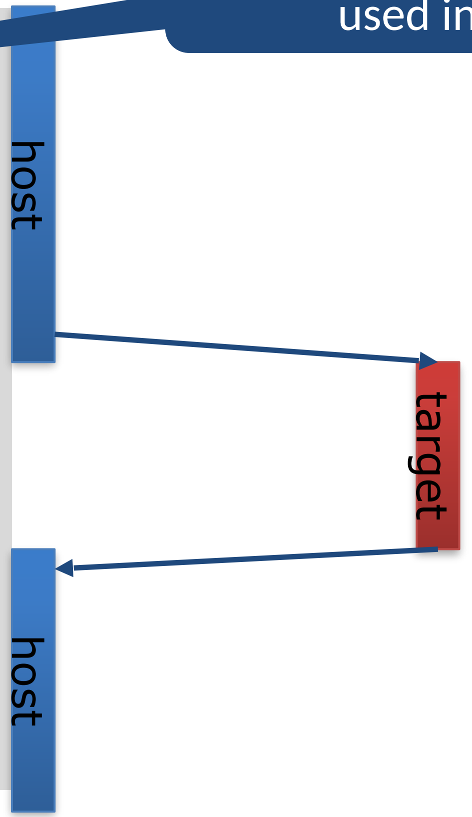



```
clang -fopenmp --offload-arch=gfx90a ...
```

Example: saxpy

The compiler identifies variables that are used in the `target` region.

```
void saxpy() {
    float a, x[SZ], y[SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target “map(tofrom:y[0:SZ])”
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```



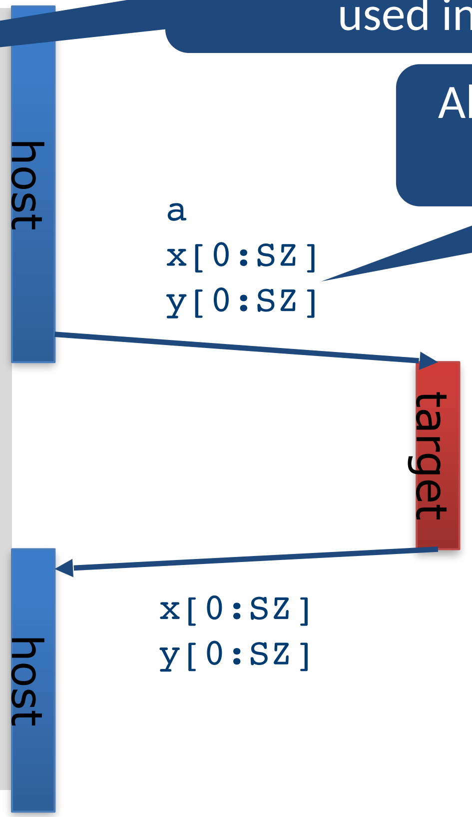

```
clang -fopenmp --offload-arch=gfx90a ...
```

Example: saxpy

```
void saxpy() {
    float a, x[SZ], y[SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target"map(tofrom:y[0:SZ])"
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

The compiler identifies variables that are used in the `target` region.

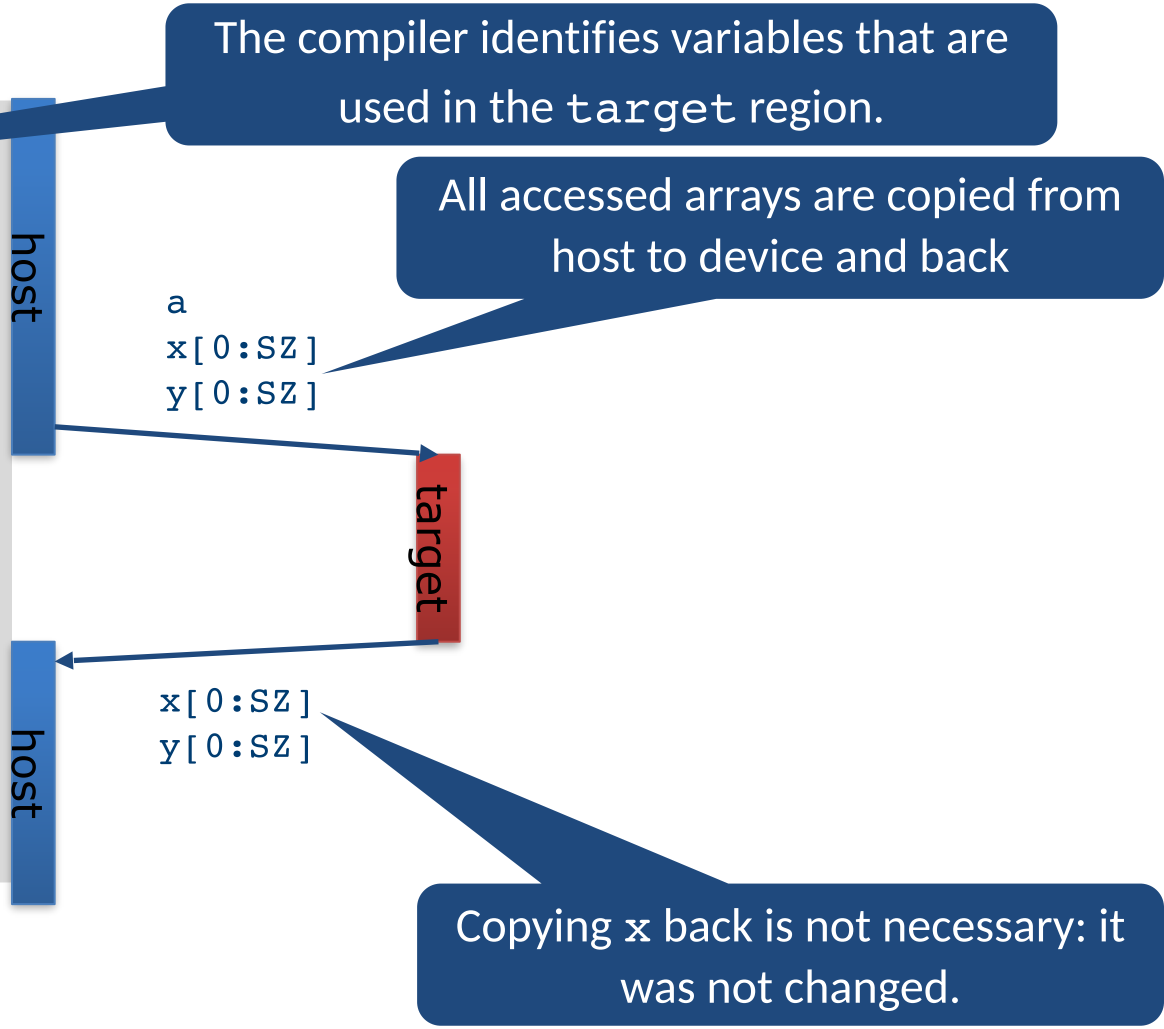
All accessed arrays are copied from host to device and back




```
clang -fopenmp --offload-arch=gfx90a ...
```

Example: saxpy

```
void saxpy() {
    float a, x[SZ], y[SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target "map(tofrom:y[0:SZ])"
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```




```
clang -fopenmp --offload-arch=gfx90a ...
```

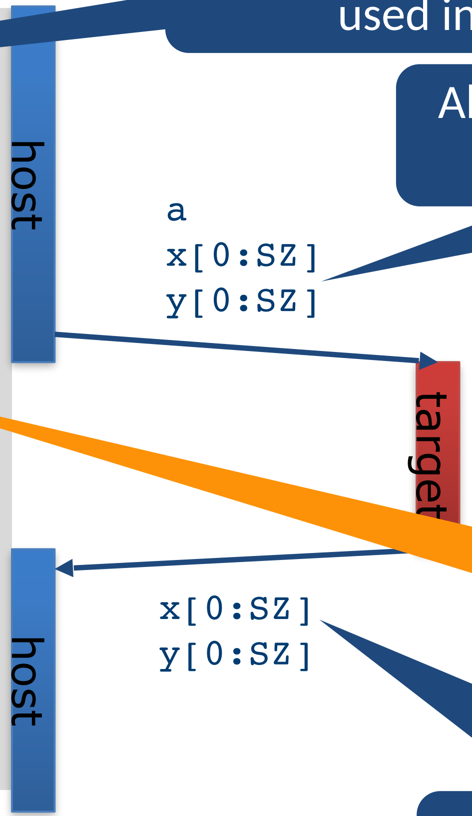
Example: saxpy

OpenMP

```
void saxpy() {
    float a, x[SZ], y[SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target “map(tofrom:y[0:SZ])”
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

The compiler identifies variables that are used in the `target` region.

All accessed arrays are copied from host to device and back



Presence check: only transfer if not yet allocated on the device.

Copying `x` back is not necessary: it was not changed.

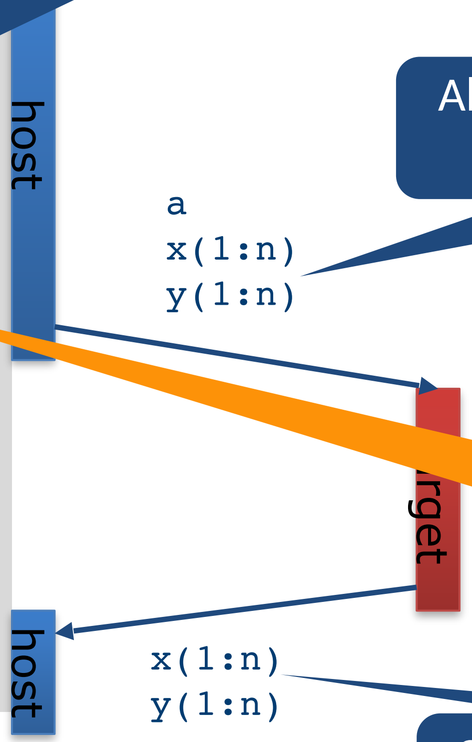
```
clang -fopenmp --offload-arch=gfx90a ...
```

Example: saxpy

The compiler identifies variables that are used in the `target` region.

```
subroutine saxpy(a, x, y, n)
    use iso_fortran_env
    integer :: n, i
    real(kind=real32) :: a
    real(kind=real32), dimension(n) :: x
    real(kind=real32), dimension(n) :: y

!$omp target "map(tofrom:y(1:n))"
    do i=1,n
        y(i) = a * x(i) + y(i)
    end do
!$omp end target
end subroutine
```



All accessed arrays are copied from host to device and back

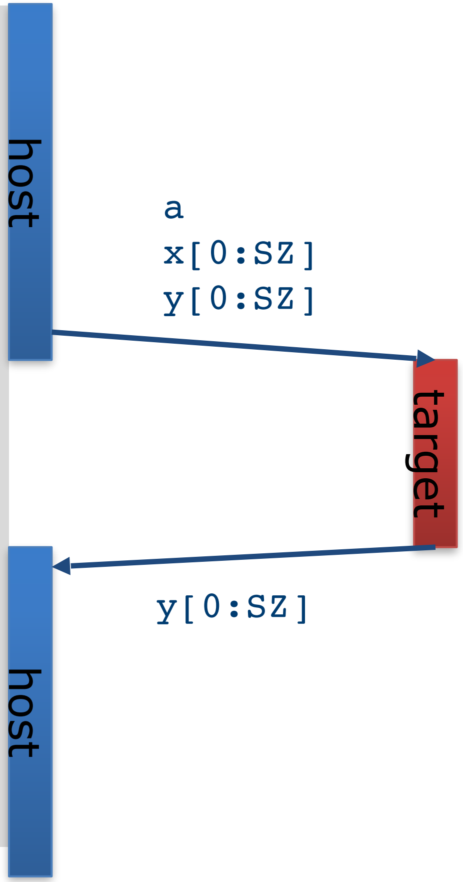
Presence check: only transfer if not yet allocated on the device.

Copying `x` back is not necessary: it was not changed.

```
flang -fopenmp --offload-arch=gfx90a ...
```

Example: saxpy

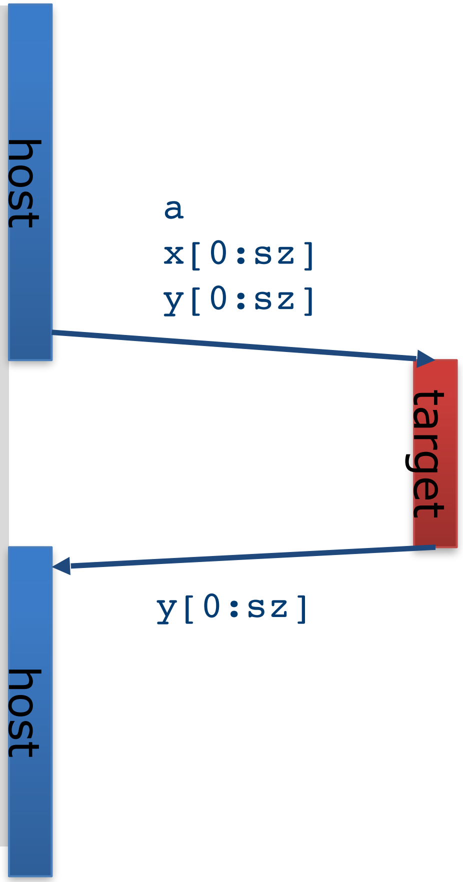
```
void saxpy() {
    double a, x[SZ], y[SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target map(to:x[0:SZ]) \
                   map(tofrom:y[0:SZ])
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```




```
clang -fopenmp --offload-arch=gfx90a ...
```

Example: saxpy

```
void saxpy(float a, float* x, float* y,
           int sz) {
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target map(to:x[0:sz]) \
                   map(tofrom:y[0:sz])
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

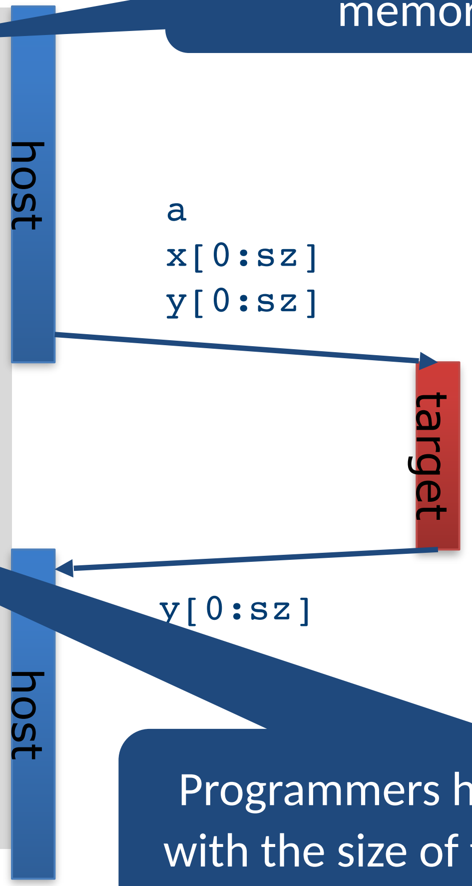



```
clang -fopenmp --offload-arch=gfx90a
```

Example: saxpy

```
void saxpy(float a, float* x, float* y,
           int sz) {
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target map(to:x[0:sz]) \
                   map(tofrom:y[0:sz])
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

The compiler cannot determine the size of memory behind the pointer.



Programmers have to help the compiler with the size of the data transfer needed.

```
clang -fopenmp --offload-arch=gfx90a
```

Exploiting (Multilevel) Parallelism

Creating Parallelism on the Target Device

- The `target` construct transfers the control flow to the target device
  - →Transfer of control is sequential and synchronous
  - →This is intentional!

- OpenMP separates offload and parallelism
  - →Programmers need to explicitly create parallel regions on the target device
  - →In theory, this can be combined with any OpenMP construct
  - →In practice, there is only a useful subset of OpenMP features for a target device such as a GPU, e.g., no I/O, limited use of base language features.

Example: saxpy

```
void saxpy(float a, float* x, float* y,
           int sz) {
#pragma omp target map(to:x[0:sz]) \
                   map(tofrom(y[0:sz])
#pragma omp parallel for simd
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```


```
clang -fopenmp --offload-arch=gfx90a
```

Example: saxpy

```
void saxpy(float a, float* x, float* y,
           int sz) {
#pragma omp target map(to:x[0:sz]) \
                   map(tofrom(y[0:sz])
#pragma omp parallel for simd
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

host

target

host

Create a team of threads to execute the loop in parallel using SIMD instructions.

```
clang -fopenmp --offload-arch=gfx90a
```

Example: saxpy

```
void saxpy(float a, float* x, float* y,
           int sz) {
#pragma omp target map(to:x[0:sz]) \
                   map(tofrom(y[0:sz])
#pragma omp parallel for simd
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

host

target

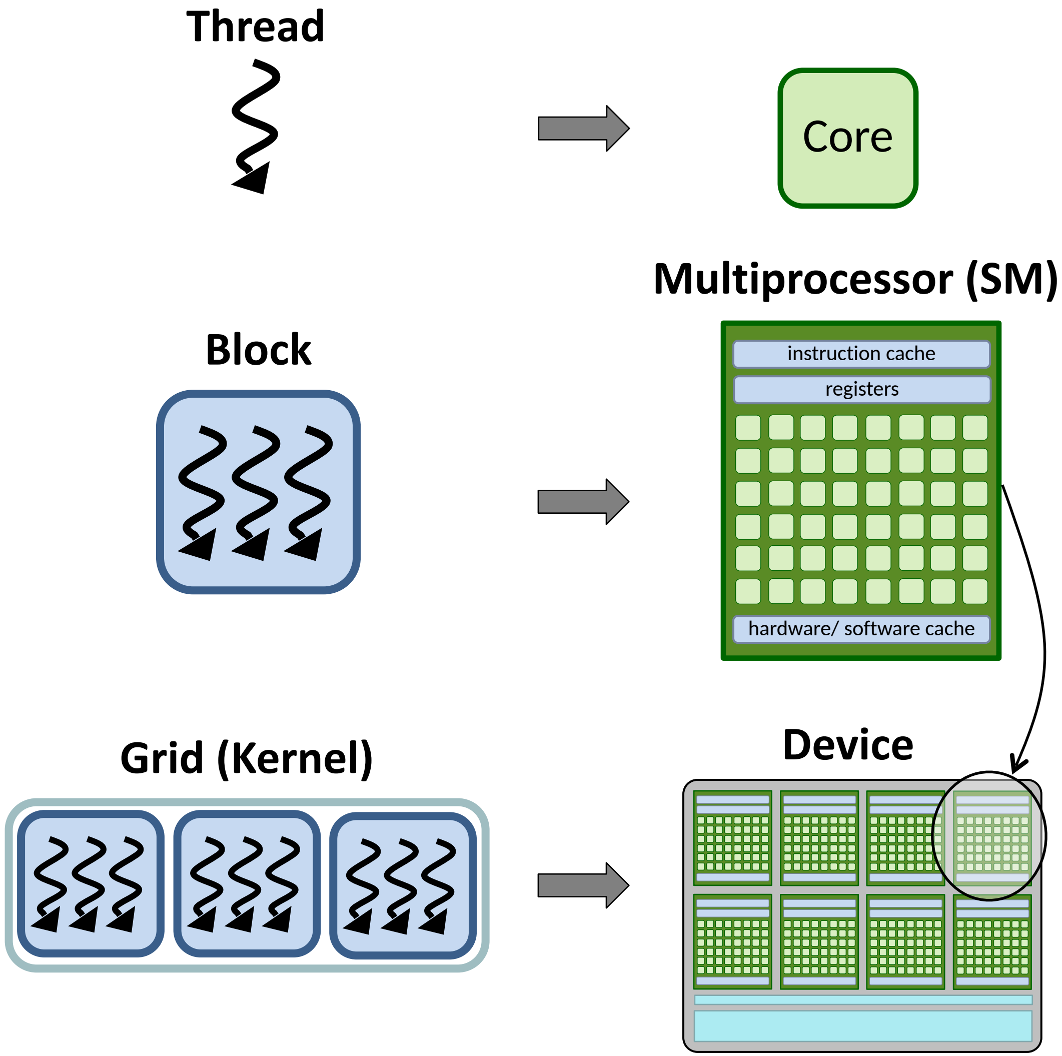
host

GPUs are multi-level devices: SIMD, threads, thread blocks

Create a team of threads to execute the loop in parallel using SIMD instructions.

```
clang -fopenmp --offload-arch=gfx90a
```

Mapping to Hardware



- Each thread is executed by a core
- Each block is executed on a SM
- Several concurrent blocks can reside on a SM depending on shared resources
- Each kernel is executed on a device

teams Construct

- Support multi-level parallel devices
- Syntax (C/C++):

```
#pragma omp teams [clause[[,] clause],…]
structured-block
```

- Syntax (Fortran):

```
!$omp teams [clause[[,] clause],…]
structured-block
```

- Clauses

```
num_teams(integer-expression), thread_limit(integer-expression)
default(shared | firstprivate | private none)
private(list), firstprivate(list), shared(list), reduction(operator:list)
```

Multi-level Parallel saxpy

- Manual code transformation
  - → Tile the loop into an outer loop and an inner loop.
  - → Assign the outer loop to “teams”.
  - → Assign the inner loop to the “threads”.
  - → (Assign the inner loop to SIMD units.)

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

Multi-level Parallel saxpy

- Manual code transformation
  → Tile the loop into an outer loop and an inner loop.
  → Assign the outer loop to "teams".
  → Assign the inner loop to the "threads".
  → (Assign the inner loop to SIMD units.)

```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target teams map(to:x[0:sz]) map(tofrom:y[0:sz])
num_teams(nteams)
    {
        int bs = n / omp_get_num_teams();    // n assumed to be multiple of
#teams
        #pragma omp distribute
        for (int i = 0; i < sz; i += bs) {
            #pragma omp parallel for simd firstprivate(i,bs)
            for (int ii = i; ii < i + bs; ii++) {
                y[ii] = a * x[ii] + y[ii];
}   }   }   }
```

Multi-level Parallel saxpy

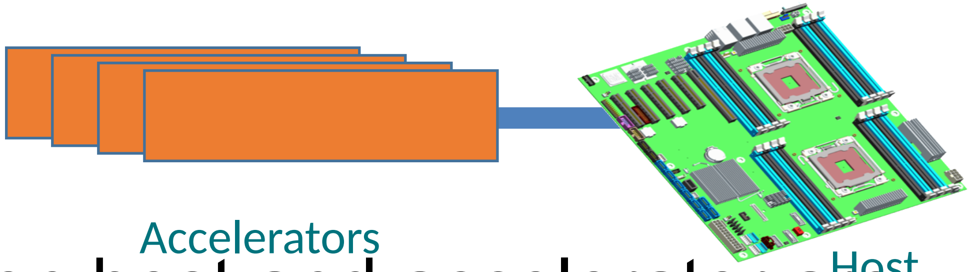
- For convenience, OpenMP defines composite constructs to implement the required code transformations

```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target teams distribute parallel for simd \
            num_teams(num_blocks) map(to:x[0:sz]) map(tofrom:y[0:sz])
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

```
subroutine saxpy(a, x, y, n)
    ! Declarations omitted
!$omp omp target teams distribute parallel do simd &
!$omp&            num_teams(num_blocks) map(to:x) map(tofrom:y)
    do i=1,n
        y(i) = a * x(i) + y(i)
    end do
!$omp end target teams distribute parallel do simd
end subroutine
```

Optimizing Data Transfers

Optimizing Data Transfers is Key to Performance



- Connections between host and accelerator are typically lower-bandwidth, higher-latency interconnects
  - →Bandwidth host memory: hundreds of GB/sec
  - →Bandwidth accelerator memory: TB/sec
  - →PCIe Gen 4 bandwidth (16x): tens of GB/sec

- Unnecessary data transfers must be avoided, by
  - →only transferring what is actually needed for the computation, and
  - →making the lifetime of the data on the target device as long as possible.

Role of the Presence Check

- If `map` clauses are not added to `target` constructs, presence checks determine if data is already available in the device data environment:

```
subroutine saxpy(a, x, y, n)
    use iso_fortran_env
    integer :: n, i
    real(kind=real32) :: a
    real(kind=real32), dimension(n) ::
x
    real(kind=real32), dimension(n) ::
y

!$omp target
    do i=1,n
        y(i) = a * x(i) + y(i)
    end do
!$omp end target
end subroutine
```

– OpenMP maintains a mapping table that records what memory pointers have been mapped.

– That table also maintains the translation between host memory and device memory.

– Constructs with no `map` clause for a data item then determine if data has been mapped and if not, a `map(tofrom:…)` is added for that data item.

Role of the Presence Check

- If `map` clauses are not added to `target` constructs, presence checks determine if data is already available in the device data environment:

```
subroutine saxpy(a, x, y, n)
    use iso_fortran_env
    integer :: n, i
    real(kind=real32) :: a
    real(kind=real32), dimension(n) ::
x
    real(kind=real32), dimension(n) ::
y

!$omp target“present?(y)” “present?(x)”
    do i=1,n
        y(i) = a * x(i) + y(i)
    end do
!$omp end target
end subroutine
```

- OpenMP maintains a mapping table that records what memory pointers have been mapped.
- That table also maintains the translation between host memory and device memory.
- Constructs with no `map` clause for a data item then determine if data has been mapped and if not, a `map(tofrom:…)` is added for that data item.

Optimize Data Transfers

- Reduce the amount of time spent transferring data:
  - →Use `map` clauses to enforce direction of data transfer.
  - →Use `target data`, `target enter data`, `target exit data` constructs to keep data environment on the target device.

```
subroutine caller
    ! Declarations omitted

!$omp target data map(to:x) &
                  map(tofrom:y)
    call saxpy(a, x, y, n)
!$omp end target
end subroutine
```

```
subroutine saxpy(a, x, y, n)
    ! Declarations omitted

!$omp target
    do i=1,n
        y(i) = a * x(i) + y(i)
    end do
!$omp end target
end subroutine
```

Optimize Data Transfers

- Reduce the amount of time spent transferring data:
  - →Use `map` clauses to enforce direction of data transfer.
  - →Use `target data`, `target enter data`, `target exit data` constructs to keep data environment on the target device.

```
subroutine caller
    ! Declarations omitted

!$omp target data map(to:x) &
                  map(tofrom:y)
    call saxpy(a, x, y, n)
!$omp end target
end subroutine
```

```
subroutine saxpy(a, x, y, n)
    ! Declarations omitted

!$omp target "present?(y)" "present?(x)"
    do i=1,n
        y(i) = a * x(i) + y(i)
    end do
!$omp end target
end subroutine
```

Optimize Data Transfers

- Reduce the amount of time spent transferring data:
  - → Use `map` clauses to enforce direction of data transfer.
  - → Use `target data`, `target enter data`, `target exit data` constructs to keep data environment on the target device.

```
void example() {
    float tmp[N], data_in[N], float
data_out[N];
#pragma omp target data map(alloc:tmp[:N]) \
                        map(to:a[:N],b[:N]) \
                        map(tofrom:c[:N])
    {
        zeros(tmp, N);
        compute_kernel_1(tmp, a, N); // uses
target
        saxpy(2.0f, tmp, b, N);
        compute_kernel_2(tmp, b, N); // uses
target
        saxpy(2.0f, c, tmp, N);
}   }
```

```
void zeros(float* a, int n) {
#pragma omp target teams distribute parallel
for
    for (int i = 0; i < n; i++)
        a[i] = 0.0f;
}
```

```
void saxpy(float a, float* y, float* x, int
n) {
#pragma omp target teams distribute parallel
for
    for (int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}
```

`target data` Construct Syntax

- Create scoped data environment and transfer data from the host to the device and back
- Syntax (C/C++)

```
#pragma omp target data [clause[[,] clause],…]
structured-block
```

- Syntax (Fortran)

```
!$omp target data [clause[[,] clause],…]
structured-block
!$omp end target data
```

- Clauses

```
device(scalar-integer-expression)
map([{alloc | to | from | tofrom | release | delete}:] list)
if(scalar-expr)
```

`target update` Construct Syntax

- Issue data transfers to or from existing data device environment
- Syntax (C/C++)

```
#pragma omp target update [clause[[,] clause],…]
```

- Syntax (Fortran)

```
!$omp target update [clause[[,] clause],…]
```

- Clauses

```
device(scalar-integer-expression)
to(list)
from(list)
if(scalar-expr)
```

Example: target data and target update

```
#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);

    update_input_array_on_the_host(input);

#pragma omp target update device(0) to(input[:N])

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

Example: target data and target update

```
#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);

    update_input_array_on_the_host(input);

#pragma omp target update device(0) to(input[:N])

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

Programming OpenMP

Hands-on Exercises: Jacobi on GPU

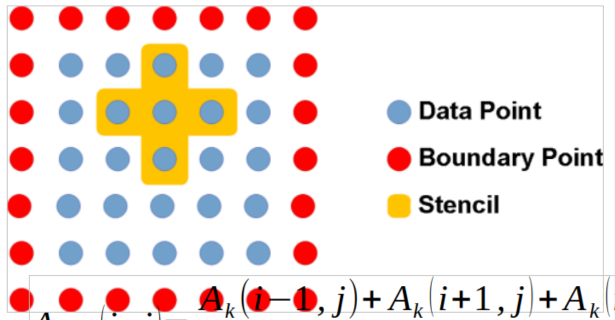
Christian Terboven
Michael Klemm

Jacobi on GPU / 1

During the following exercises, you will port a Jacobi solver to OpenMP. This **Jacobi** example solves a finite difference discretization (5-point-stencil) of the Laplace equation (2D):

$$\nabla^2 A(x,y) = 0$$

using the Jacobi iterative method. To this end, the Jacobi method starts with an approximation of the objective function *f(x,y)* and reuses formerly-computed matrix elements to solve the current one. It iterates only about the inner elements of the 2D-grid so that the boundary elements are only used within the stencil. The solving process is aborted if either a certain number of iterations is achieved (see `iter_max`) or the computed approximation is probably close to the solution. In this code, the latter is evaluated by checking whether the biggest change on any matrix element (see array `err` and variable `err`) is smaller than a given tolerance value, in the current iteration.



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k}{4}$$

Figure 1: 5-point stencil

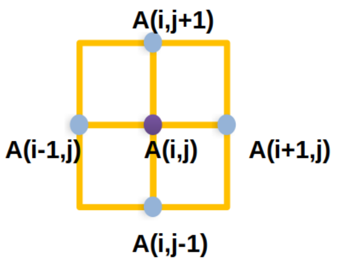


Figure 2: Computation of matrix element A(i,j)

Jacobi on GPU / 2

- Task 0: You might want to acquire reference measurements on the host (wo/ GPU)...
- Task 1: Get it to the GPU: Parallelize only the one most compute-intensive loop
- Task 2: Improve the data management and the amount of parallelism on the GPU
- Task 3: Optimize that scheduling of iterations for the GPU
- Understand the performance of the host and the GPU
- Future tasks: use multiple GPUs, use the host and a GPU, …