Intro to GPU Programming with the OpenMP API

Dr.-Ing. Michael Klemm

Chief Executive Officer
OpenMP Architecture Review Board

Principal Member of Technical Staff
HPC Center of Excellence
AMD

OpenMP Architecture Review Board

The mission of the OpenMP ARB (Architecture Review Board) is to standardize directive-based multi-language high-level parallelism that is performant, productive and portable.

The OpenMP API moves common approaches into an industry standard to simplify a developers' life.

OpenMP

Agenda

- OpenMP device and execution model
- Offload basics
- Exploit parallelism
- Asynchronous offloading
- Summary

OpenMP

Introduction to OpenMP Offload Features

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 ☺)

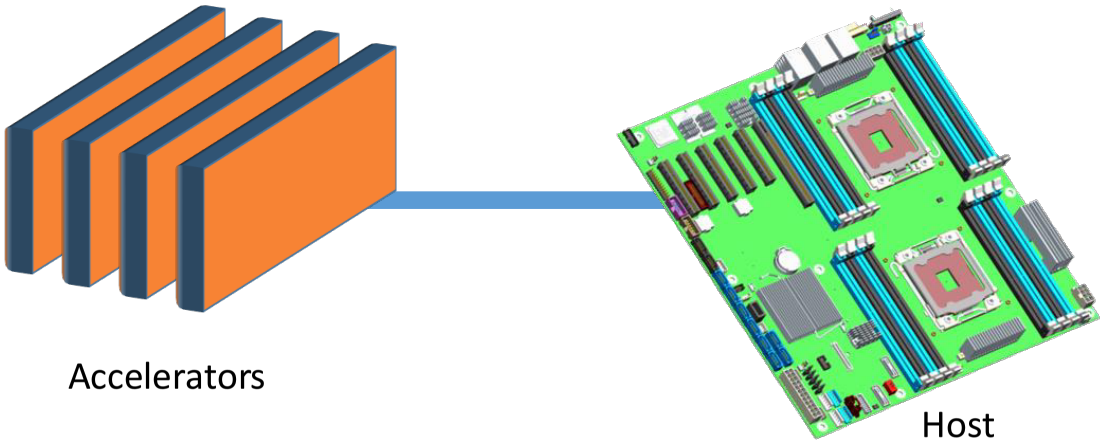
This is the code we want to execute on a target device (i.e., GPU)

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!

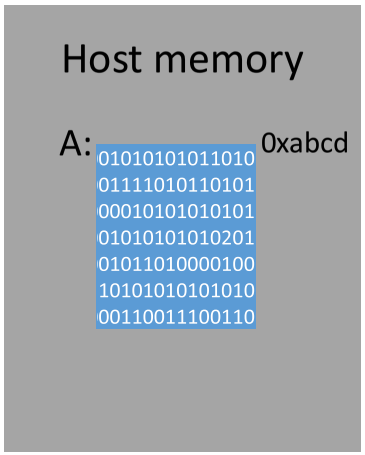
Device Model

- 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.




```
!$omp target          &
!$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)

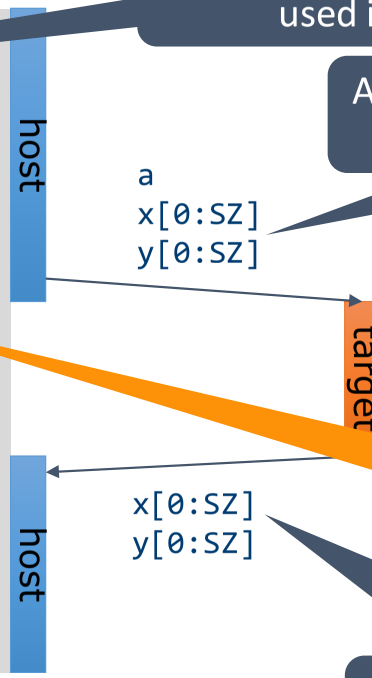
```
!$omp target [clause[[,] clause],…]
structured-block
!$omp end target
```

- 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 "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

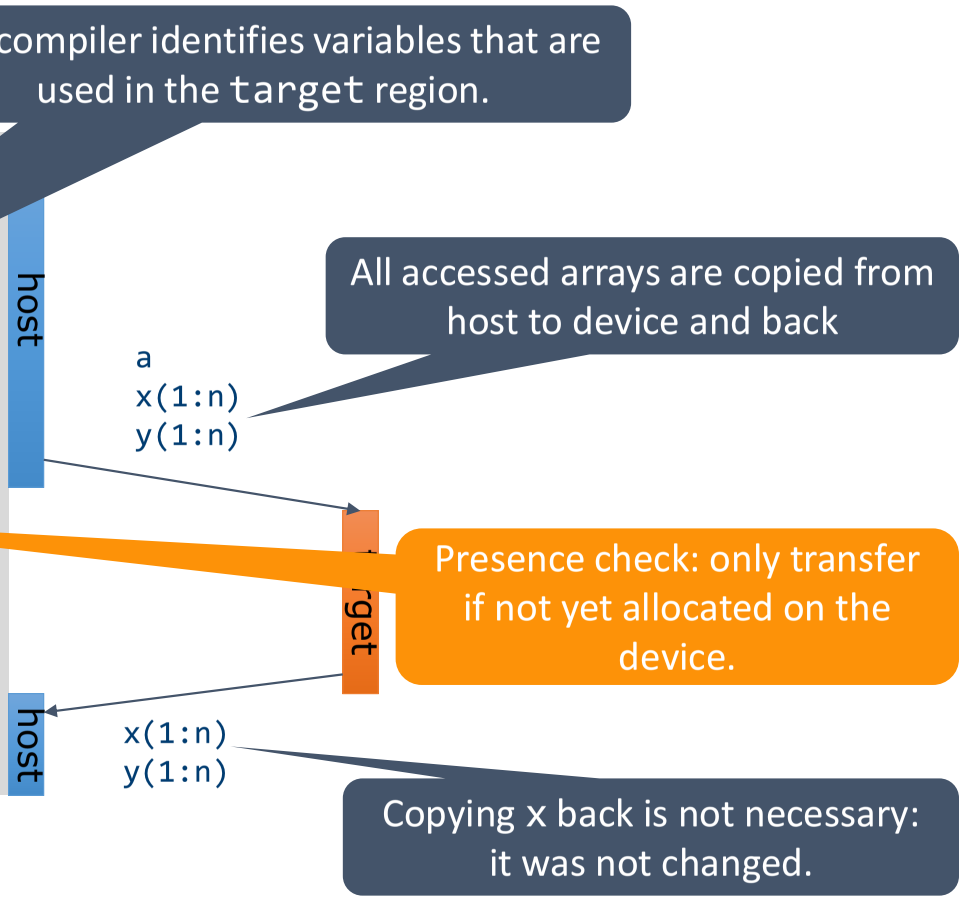
Presence check: only transfer if not yet allocated on the device.

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

clang/LLVM:	`clang -fopenmp -fopenmp-targets=<target triple>`
GNU:	`gcc -fopenmp`
AMD ROCm:	`clang -fopenmp -offload-arch=gfx908`
NVIDIA:	`nvcc –mp=gpu -gpu=cc80`
Intel:	`icx -fiopenmp -fopenmp-targets=spir64`
IBM XL:	`xlc –qsmp –qoffload –qtgtarch=sm_70`

Example: saxpy

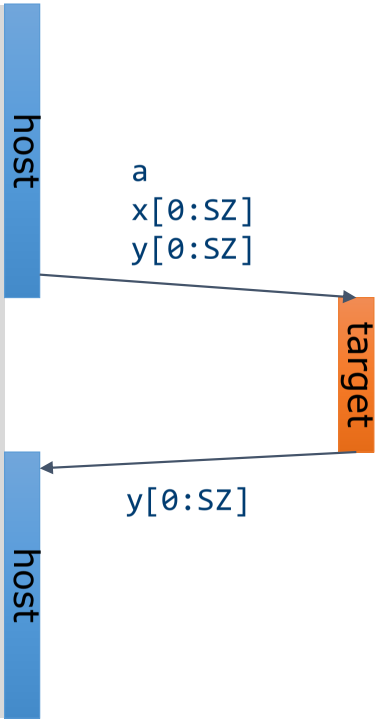
```
subroutine saxpy(a, x, y, n)
    use iso_fortran_env
    integer :: n, i
    real(kind=real32) :: a
    real(kind=real32), dimension(n) :: x
    real(kind=real32), dimension(n) :: y

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



clang/LLVM:	`flang -fopenmp –fopenmp-targets=<target triple>`
GNU:	`gfortran -fopenmp`
AMD ROCm:	`flang -fopenmp –offload-arch=gfx908`
NVIDIA:	`nvfortran –mp=gpu -gpu=cc80`
Intel:	`ifx –fiopenmp -fopenmp-targets=spir64`
IBM XL:	`xlf –qsmp –qoffload –qtgtarch=sm_70`

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);
}
```



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.

Observation when running this: the loop is a sequential loop, and the capabilities of the GPU are not really used! ☹

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

OpenMP

Commercial Break… Community Interaction

Check out openmp.org/news/events-calendar/

OpenMP

Exploiting (Multilevel) Parallelism

OpenMP

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];
    }
}
```

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

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

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
```

`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
```

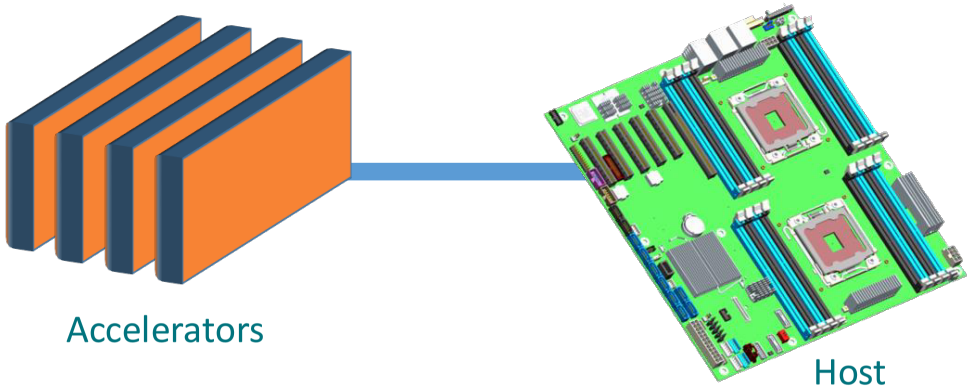
- Clauses

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

OpenMP

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

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

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

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)
  }
```

`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],…]
```

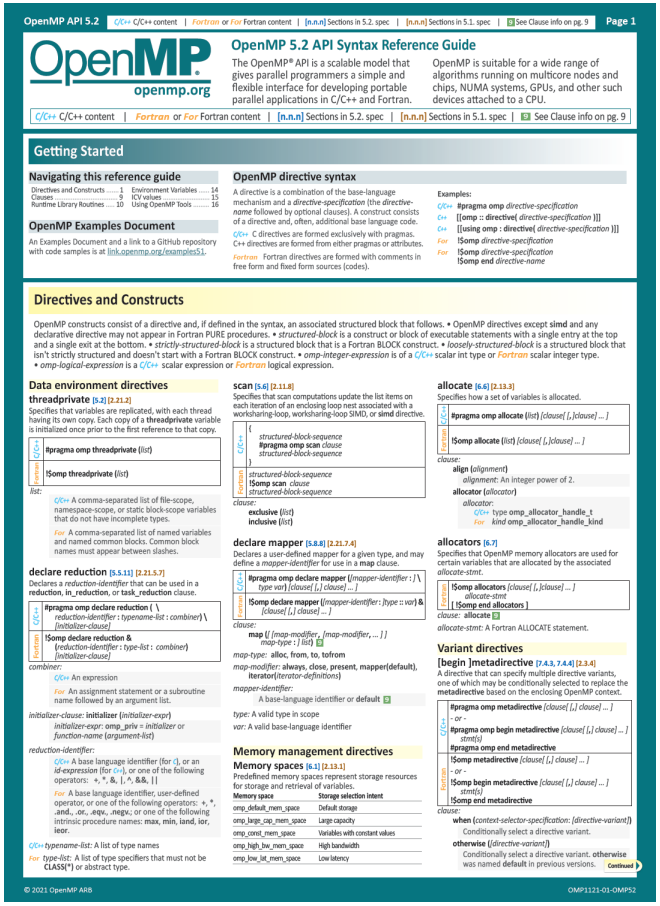
- Syntax (Fortran)

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

- Clauses

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

Commerical Break…



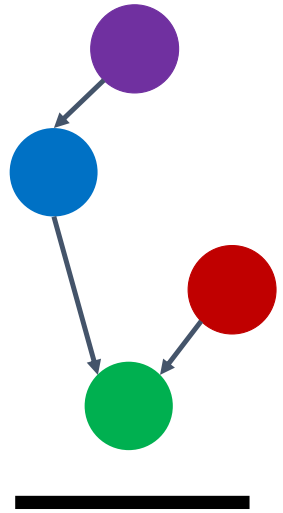
OpenMP

Asynchronous Offloading

Asynchronous Offloads

- OpenMP `target` constructs are synchronous by default
  - The encountering host thread awaits the end of the `target` region before continuing
  - The `nowait` clause makes the target constructs asynchronous (in OpenMP speak: they become an OpenMP task)

```
#pragma omp task                                                depend(out:a)
    init_data(a);

#pragma omp target map(to:a[:N]) map(from:x[:N])  nowait      depend(in:a) depend(out:x)
    compute_1(a, x, N);

#pragma omp target map(to:b[:N]) map(from:z[:N])  nowait      depend(out:y)
    compute_3(b, z, N);

#pragma omp target map(to:y[:N]) map(to:z[:N])    nowait      depend(in:x) depend(in:y)
    compute_4(z, x, y, N);

#pragma omp taskwait
```



Summary

- OpenMP API is ready to use GPUs for offloading computations
  - Mature offload model w/ support for asynchronous offload/transfer
  - Tightly integrates with OpenMP multi-threading on the host

- More, advanced features (not covered here)
  - Memory management API
  - Interoperability with native data management
  - Interoperability with native streaming interfaces
  - Unified shared memory support

Visit www.openmp.org for more information