OpenMP Offload Programming: Introduction

Dr.-Ing. Michael Klemm

Principal Member of Technical Staff
Compilers, Languages, Runtimes & Tools
Machine Learning & Software Engineering

Foundations

Running Example for this Presentation: saxpy

```
void saxpy() {
    float a, x[N], y[N];
    // 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 < N; 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 ☺)

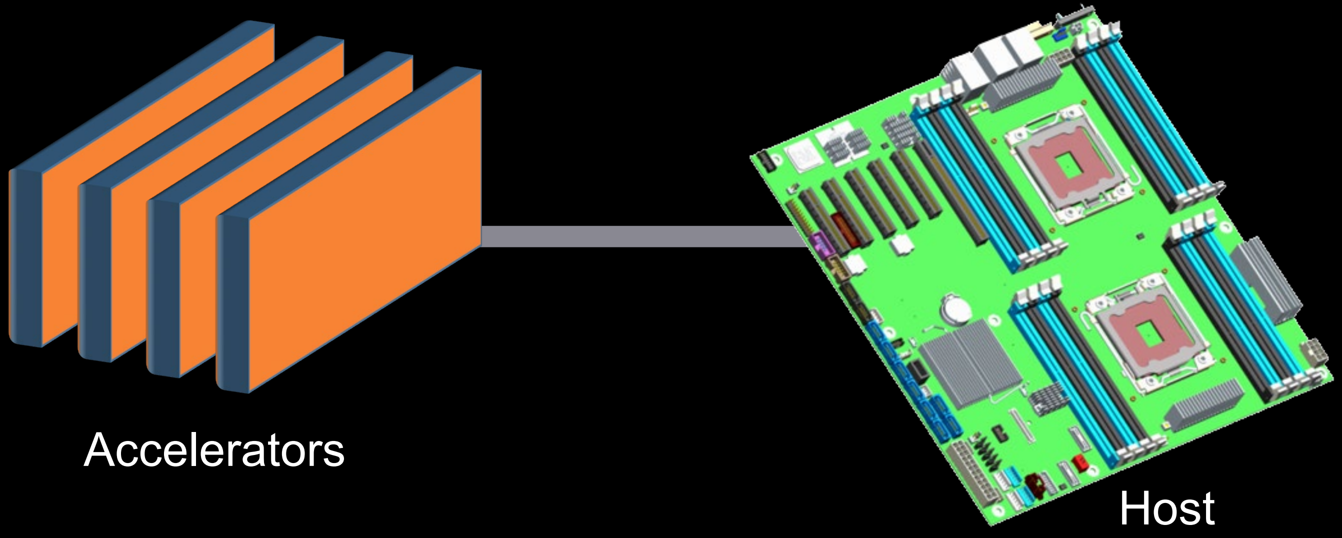
This is the code we want to execute on a target device (i.e., GPU)

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!

OpenMP Device Model

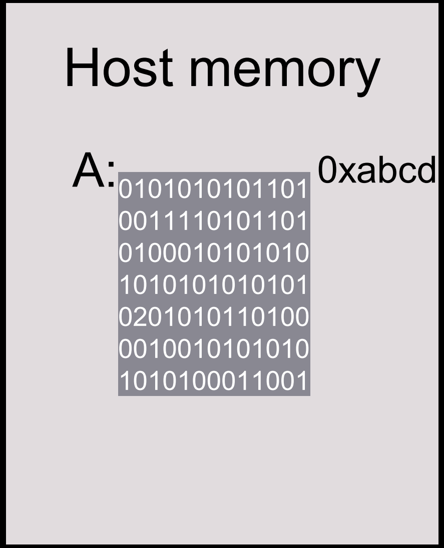
- As of version 4.0 the OpenMP API supports accelerators/coprocessors.
- Device model:
  - One host for “traditional” multi-threading
  - Multiple accelerators/coprocessors of the same kind for offloading
  - Devices are accessible though a device ID (from 0 to *n*-1 for *n* devices)
- OpenMP device model is agnostic of actual technology. In theory, devices only need to
  - be able to receive data from the host and send data back and
  - perform computation upon request.



AMD

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
```

AMD

Offload Basics

OpenMP Device Constructs

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

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

- Syntax (Fortran)

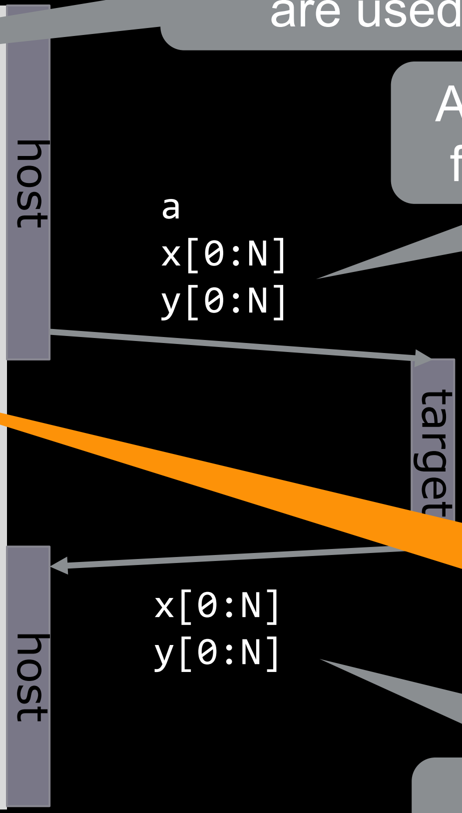
```
!$omp target [clause[[,] clause],…]
structured-block
!$omp end target
```

- Clauses

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


Example: saxpy

```
void saxpy() {
    float a, x[N], y[N];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
    #pragma omp target "map(tofrom:y[0:N])"
    for (int i = 0; i < N; 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.

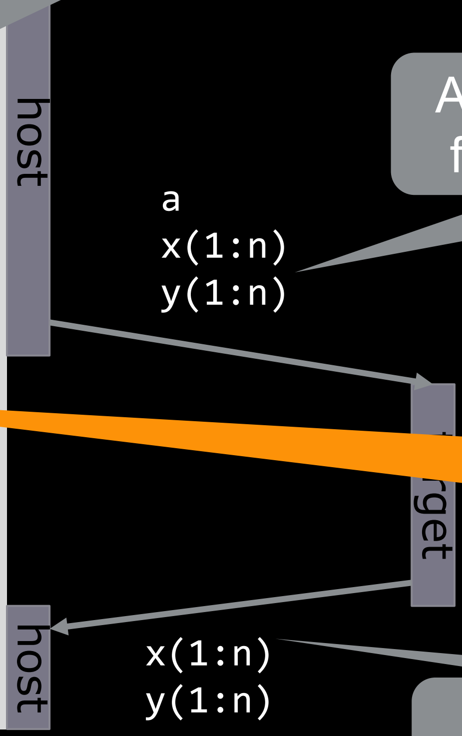
Copying x back is not necessary. It was not changed.

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


Example: saxpy

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

```fortran
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.

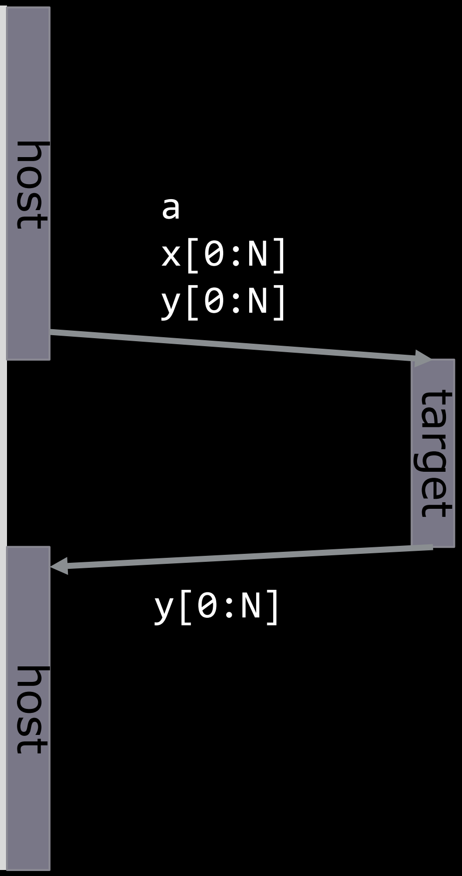
Copying x back is not necessary: it was not changed.

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


AMD

Example: saxpy

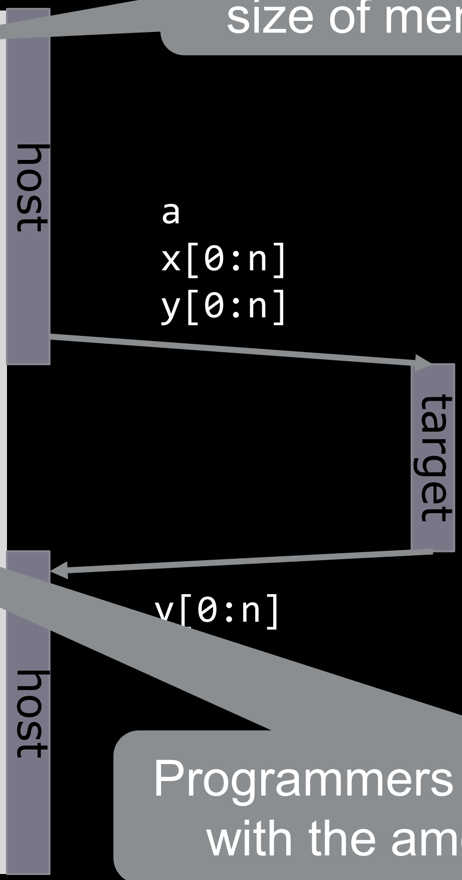
```
void saxpy() {
    double a, x[N], y[N];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
    #pragma omp target map(to:x[0:N]) \
                       map(tofrom:y[0:N])
    for (int i = 0; i < N; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te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




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


Example: saxpy

```
void saxpy(float a, float* x, float* y,
           int n) {
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
    #pragma omp target map(to:x[0:n]) \
                       map(tofrom:y[0:n])
    for (int i = 0; i < n; 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 amount of data to transfer.

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


AMD

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 n) {
#pragma omp target map(to:x[0:n]) \
                   map(tofrom(y[0:n])
#pragma omp parallel for simd
    for (int i = 0; i < n; 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.

 |

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” (OpenCL: work groups; HIP: blocks).
  - Assign the inner loop to the “threads” (OpenCL: work items; HIP: threads).
  - (Assign the inner loop to SIMD units.)

```
void saxpy(float a, float* x, float* y, int n) {
    #pragma omp target teams map(to:x[0:n]) map(tofrom:y[0:n]) num_teams(nteams)
    {
        int bs = n / omp_get_num_teams(); // could also use nteams
        #pragma omp distribute
        for (int i = 0; i < n; 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];
            }
        }
    }
}
```

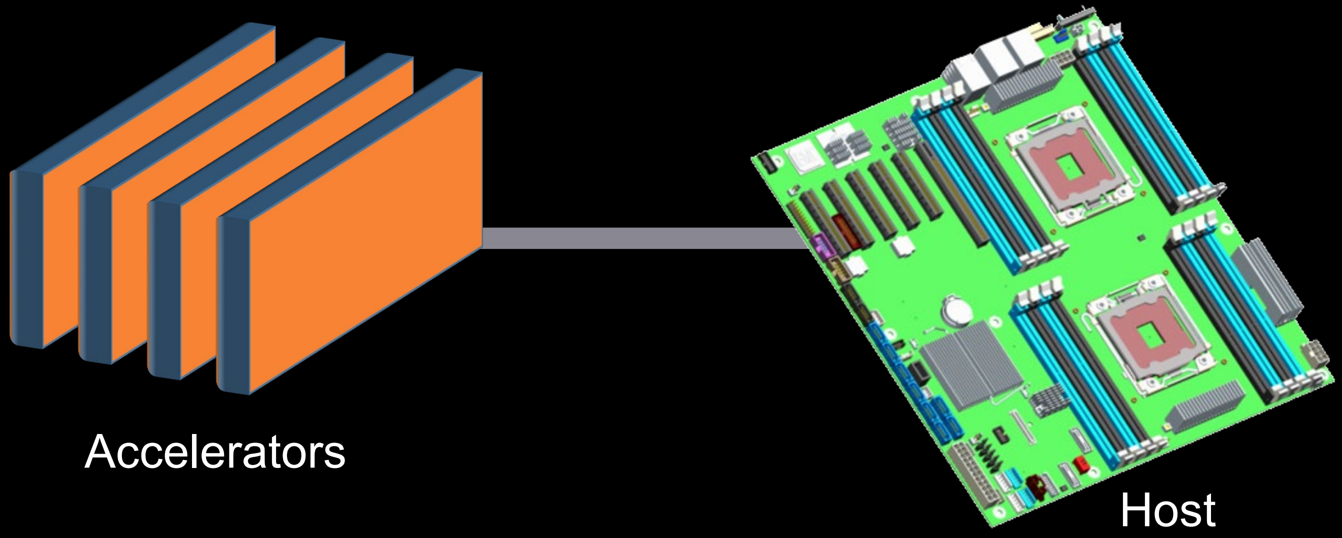

Multi-level Parallel saxpy

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

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

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.

No `map` clauses! Presence checks will find data via the pointer.

Create data environment.

```
void example() {
    float tmp[N], a[N], b[N], c[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 teams distribute parallel for simd
    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 teams parallel for simd reduction(+:res)
    for (i=0; i<N; i++)
      res += final_computation(input[i], tmp[i], i)
  }
```

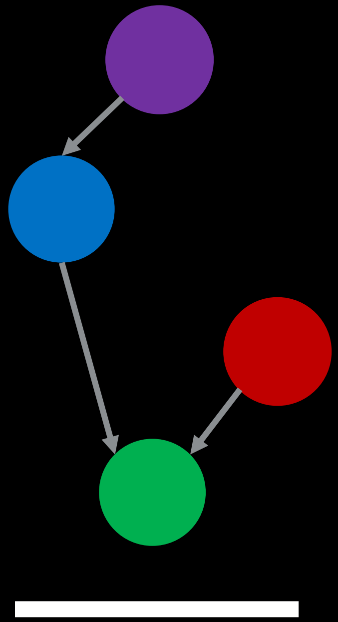
host
target
host
target
host

Asynchronous Offloading & Unified Shared Memory

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 lingo: 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:y[:N])        nowait   depend(in:b) depend(out:y)
    compute_2(b, y, N);

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

#pragma omp taskwait
```



Using Unified Shared Memory

CPU CODE

```
double* in = (double*)malloc(Msize);
double* out = (double*)malloc(Msize);

 for (int i=0; i<M; i++)
   in[i] = …;

 for (int i=0; i<M; i++)
   out[i] = ... in[i] ...;

for (int i=0; i<M; i++)
  … = out[i];
```

W/O UNIFIED MEMORY

```
double* in = (double*)malloc(Msize);
double* out = (double*)malloc(Msize);

for (int i=0; i<M; i++)
   in[i] = …;

#pragma omp target data \
            map(to:in[0:Msize]) \
            map(from:out[0:Msize])
{
#pragma omp target teams distribute \
            parallel for
  for (int i=0; i<M; i++)
    out[i] = ... in[i] ...;
}

for (int i=0; i<M; i++)
  … = out[i];
```

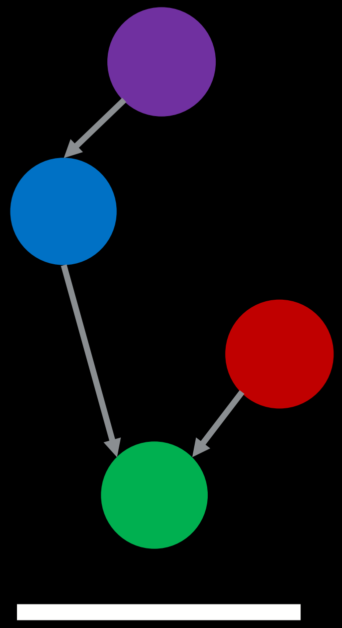
UNIFIED MEMORY

```
#pragma omp require unified_shared_memory

double* in = (double*)malloc(Msize);
double* out = (double*)malloc(Msize);

for (int i=0; i<M; i++)
   in[i] = …; //writes GPU mem directly

#pragma omp target teams distribute \
            parallel for
  for (int i=0; i<M; i++)
    out[i] = ... in[i] ...;

for (int i=0; i<M; i++)
  … = out[i]; //reads GPU mem directly
```

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 lingo: 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:y[:N])         nowait   depend(in:b) depend(out:y)
    compute_2(b, y, N);

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

#pragma omp taskwait
```



Calling OpenMP Kernels with HIP-managed Buffers

HIP Buffer Management

```
void example() {
    HIPCALL(hipSetDevice(0));

    compute_1(n, x);
    compute_2(n, y);

    HIPCALL(hipMalloc(&x_dev, sizeof(*x_dev) * n));
    HIPCALL(hipMalloc(&y_dev, sizeof(*y_dev) * n));
    HIPCALL(hipMemcpy(x_dev, x, sizeof(*x) * n, hipMemcpyHostToDevice));
    HIPCALL(hipMemcpy(y_dev, y, sizeof(*y) * n, hipMemcpyHostToDevice));

    saxpy_omp(a, x_dev, y_dev, n);

    HIPCALL(hipMemcpy(y, y_dev, sizeof(*y) * n, hipMemcpyDeviceToHost));
    HIPCALL(hipFree(x_dev));
    HIPCALL(hipFree(y_dev));

    compute_3(n, y);
}
```

Allocate buffers to hold data on the target GPU.

Copy the data from the host memory to the GPU buffer space.

Copy result data back from GPU.

Deallocate the buffers on the target GPU.

HIP Buffer Management

```
void example() {
    HIPCALL(hipSetDevice(0));

    compute_1(n, x);
    compute_2(n, y);

    HIPCALL(hipMalloc(&x_dev, sizeof(*x_dev) * n));
    HIPCALL(hipMalloc(&y_dev, sizeof(*y_dev) * n));
    HIPCALL(hipMemcpy(x_dev, x, sizeof(*x) * n
    HIPCALL(hipMemcpy(y_dev, y, sizeof(*y) * n

    saxpy_omp(a, x_dev, y_dev, n);

    HIPCALL(hipMemcpy(y, y_dev, sizeof(*y) * n
    HIPCALL(hipFree(x_dev));
    HIPCALL(hipFree(y_dev));

    compute_3(n, y);
}
```

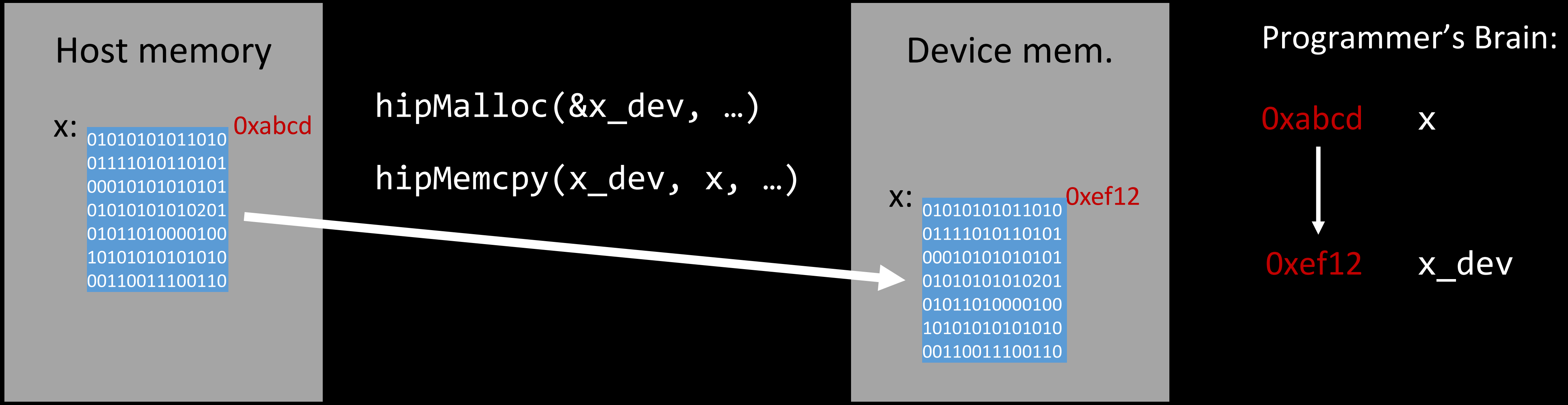
```
void saxpy_omp(float a, float * x,
               float * y, size_t n) {
#pragma omp target teams distribute \
                   parallel for simd
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

OpenMP region needs to access the existing device pointers, no pointer translation please!

AMD

HIP “Pointer Translation”

- In the HIP model, “pointer translation” is handled by the programmer!
  - Explicitly associate host pointer (“x”) with device pointer (“x_dev”).
  - Association is done via the `hipMemcpy()` API that requires both as arguments.



 |

Disabling OpenMP Presence Check (and Pointer Translation)

- The OpenMP `target` construct has the `is_device_ptr()` clause that
  - instructs the OpenMP implementation to not do a presence check for the listed entities, and
  - avoids pointer translation and passes the given pointer value into the kernel w/o further interpretation.

```
void saxpy_omp(float a, float * x, float * y, size_t n) {
    #pragma omp target teams distribute parallel for \
                schedule(nonmonotonic:static,1)      \
                is_device_ptr(x, y)
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

 |

Calling HIP from OpenMP Offload Regions

Example: Calling saxpy

```
void example() {
    float a = 2.0;
    float * x;
    float * y;

    compute_1(n, x);
    compute_2(n, y);
    #pragma omp target data map(to:x[0:count])
    {
        some_gpu_omp_kernel(x, y, n);
        saxpy(a, x, y, n)
        compute_3(n, y);
    }
}
```

Allocate device memory for x and y, and specify directions of data transfers

Let's assume that we want to implement the saxpy() function in a low-level language.

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


HIP Kernel for saxpy()

- Assume a HIP version of the SAXPY kernel:

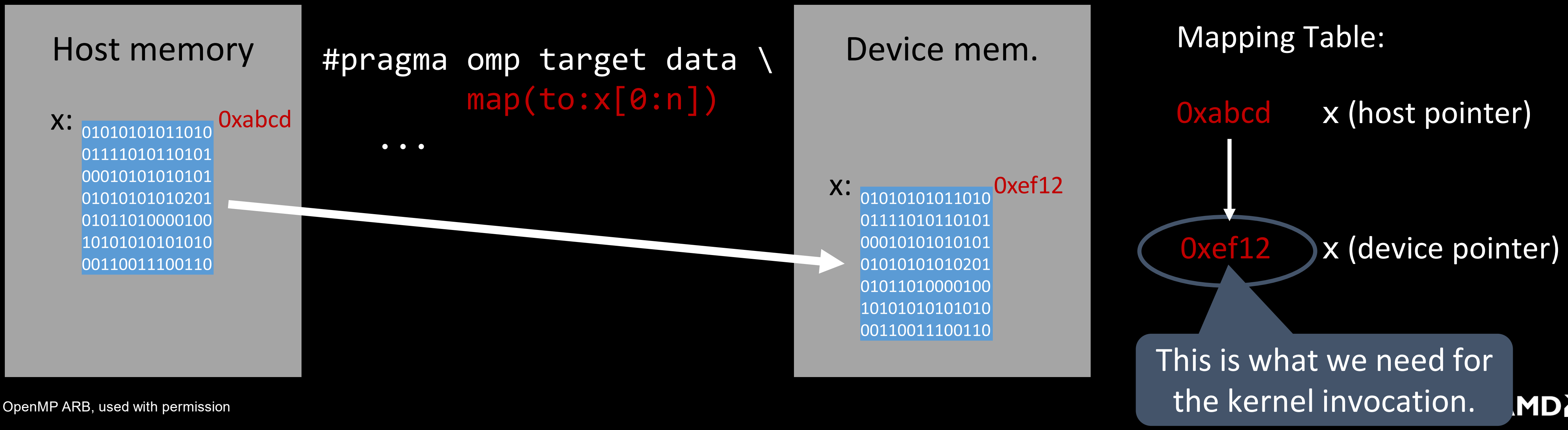
```
__global__ void saxpy_kernel(float a, float * x, float * y, size_t n, ) {
    size_t i = threadIdx.x + blockIdx.x * blockDim.x;
    y[i] = a * x[i] + y[i];
}

void saxpy_hip(size_t n, float a, float * x, float * y) {
    assert(n % 256 == 0);
    saxpy_kernel<<<n/256,256,0,NULL>>>(n, a, x, y);
}
```

These are device pointers!

- We need a way to translate the host pointer that was mapped by OpenMP directives and retrieve the associated device pointer.

Pointer Translation /1

- When creating the device data environment, OpenMP creates a mapping between
  - the (virtual) memory pointer on the host and
  - the (virtual) memory pointer on the target device.
- This mapping is established through the data-mapping directives and their clauses.



Pointer Translation /2

- The target data construct defines the `use_device_addr` clause to perform pointer translation.
  - The OpenMP implementation searches for the host pointer in its internal mapping tables.
  - The associated device pointer is then returned.

```
type * x = 0xabcd;
#pragma omp target data use_device_addr(x[:0])
{
    example_func(x);   // x == 0xef12
}
```

- Note: the pointer variable is “shadowed” within the `target data` construct for the translation.

 |

Putting it Together…

```
void example() {
    float a = 2.0;
    float * x = ...;   // assume: x = 0xabcd
    float * y = ...;

    compute_1(n, x);
    compute_2(n, y);
    // allocate the device memory
    #pragma omp target data map(to:x[0:count]) map(tofrom:y[0:count])
    {
        some_gpu_omp_kernel(x, y, n); // mapping table: x:[0xabcd,0xef12], x = 0xabcd
        #pragma omp target data use_device_addr(x[:0],y[:0])
        {
             saxpy_hip(n, a, x, y) // mapping table: x:[0xabcd,0xef12], x = 0xef12
        }
    }
    compute_3(n, y);
}
```

 |

AOMP Implementation Status

- Call HIP kernel with OpenMP-managed buffers (`use_device_ptr`) ✓
- Call OpenMP kernels with HIP-managed buffers (`is_device_ptr`) ✓
- HIP and OpenMP kernels co-existence in same translation unit ✗

Asynchronous API Interactions

Asynchronous API Interaction

- Some APIs are based on asynchronous operations
  - MPI asynchronous send and receive
  - Asynchronous I/O
  - HIP stream-based offloading
  - In general: any other API/model that executes asynchronously with OpenMP (tasks)
- Example: HIP asynchronous memory transfers

```
do_something();
hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
do_something_else();
hipStreamSynchronize(stream);
do_other_important_stuff(dst);
```

- Programmers need a mechanism to marry asynchronous APIs with the parallel task model of OpenMP
  - How to synchronize completions events with task execution?

Try 1: Use just OpenMP Tasks

```
void hip_example() {
#pragma omp task      // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
    }
    #pragma omp task // task B
    {
        do_something_else();
    }
    #pragma omp task // task C
    {
        hipStreamSynchronize(stream);
        do_other_important_stuff(dst);
    }
}
```

Race condition between the tasks A & C, task C may start execution before task A enqueues memory transfer.

- This solution does not work!

Try 2: Use just OpenMP Tasks Dependences

```
void hip_example() {
#pragma omp task depend(out:stream)    // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
    }
    #pragma omp task                   // task B
    {
        do_something_else();
    }
    #pragma omp task depend(in:stream) // task C
    {
        hipStreamSynchronize(stream);
        do_other_important_stuff(dst);
    }
}
```

Synchronize execution of tasks through dependence. May work, but task C will be blocked waiting for the data transfer to finish

- This solution may work, but
  - Takes a thread away from execution while the system is handling the data transfer and may be problematic if the called interface is not thread-safe!

OpenMP Detachable Tasks

- OpenMP 5.0 introduces the concept of a detachable task
  - Task can detach from executing thread without being “completed”
  - Regular task synchronization mechanisms can be applied to await completion of a detached task
  - Runtime API to complete a task

- Detached task events: `omp_event_handle_t` datatype
- Detached task clause: `detach(event)`
- Runtime API: `void omp_fulfill_event(omp_event_handle_t event)`

Detaching Tasks

```
omp_event_handle_t event;
void detach_example() {
#pragma omp task detach(event)
    {
        important_code();
    }①

    #pragma omp taskwait ②④
}
```

Some other thread/task:

```
omp_fulfill_event(event);③
```

1. Task detaches
2. `taskwait` construct cannot complete
3. Signal event for completion
4. Task completes and `taskwait` can continue

Putting It All Together

```
void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
  ③ omp_fulfill_event(* (omp_event_handle_t *) cb_data);
}
void hip_example() {
    omp_event_handle_t hip_event;
#pragma omp task detach(hip_event) // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
        hipStreamAddCallback(stream, callback, &hip_event, 0);
  ① }
#pragma omp task                      // task B
        do_something_else();

#pragma omp taskwait ② ④
#pragma omp task                      // task C
    {
        do_other_important_stuff(dst);
}   }
```

1. Task A detaches
2. taskwait does not continue
3. When memory transfer completes, callback is invoked to signal the event for task completion
4. taskwait continues, task C executes

Removing the taskwait Construct

```
void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
  ② omp_fulfill_event(* (omp_event_handle_t *) cb_data);
}
void hip_example() {
    omp_event_handle_t hip_event;
#pragma omp task depend(out:dst) detach(hip_event) // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
        hipStreamAddCallback(stream, callback, &hip_event, 0);
  ① }
#pragma omp task                          // task B
        do_something_else();

#pragma omp task depend(in:dst) ③         // task C
    {
        do_other_important_stuff(dst);
}   }
```

1. Task A detaches and task C will not execute because of its unfulfilled dependency on A
2. When memory transfer completes, callback is invoked to signal the event for task completion
3. Task A completes and C's dependency is fulfilled

Summary

- OpenMP API is ready to use AMD discrete GPUs for offloading compute
  - Mature offload model w/ support for asynchronous offload/transfer
  - Tightly integrates with OpenMP multi-threading on the host
  - Unified shared memory is directly supported by the OpenMP API
  - Memory copies can automatically be elided in USM mode

- More, advanced features (not covered here)
  - Memory management API
  - Interoperability with native streaming interfaces

Disclaimer

AMD