

Programming OpenMP for GPUs

Christian Terboven

Michael Klemm



• Agenda (in total 4 days)

- Day 1: OpenMP Introduction
- Day 2: Tasking & Optimizations for NUMA
- Day 3: Introduction to Offloading with OpenMP
- **Day 3: Advanced Offloading Topics**
 - Welcome
 - Optimizing Data Transfer
 - Unstructured Data Transfers
 - Asynchronous Offloading
 - Mixing OpenMP and HIP
 - **Hands-On**

- Material

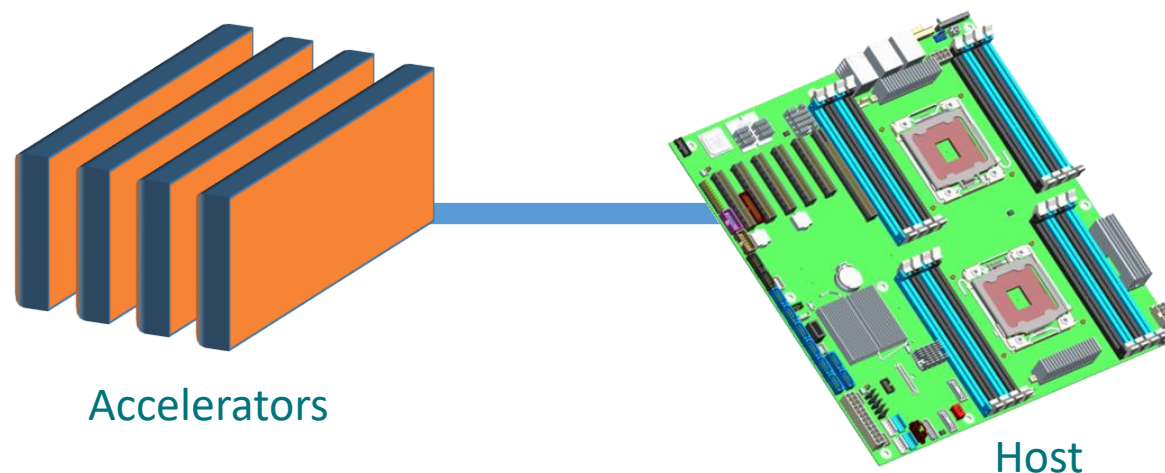


<https://github.com/cterboven/OpenMP-tutorial-CSC>



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];  
}
```


Programming OpenMP

GPU: unstructured data movement

Christian Terboven

Michael Klemm



Map variables across multiple target regions

- Optimize sharing data between host and device.
- The `target data`, `target enter data`, and `target exit data` constructs map variables but do not offload code.
- Corresponding variables remain in the device data environment for the extent of the target data region.
- Useful to map variables across multiple target regions.
- The `target update` synchronizes an original variable with its corresponding variable.

target data Construct

- Map variables to a device data environment for the extent of the region.

- 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(integer-expression)
map([map-type-modifier[ ,] [map-type-modifier[,]...] map-
type:] locator-list)
if([ target data :]scalar-expression)
use_device_ptr(ptr-list)
use_device_addr(list)
```

target enter/exit data Constructs

- Map variables to a device data environment.

- Syntax (C/C++)

```
#pragma omp target enter data clause[[[,] clause]...]
#pragma omp target exit data clause[[[,] clause]...]
```

- Syntax (Fortran)

```
!$omp target enter data clause[[[,] clause]...]
!$omp target exit data clause[[[,] clause]...]
```

- Clauses

```
if([ target enter data :] scalar-expression) OR
    if([ target exit data :] scalar-expression)
device(integer-expression)
map([ [map-type-modifier[ ,] [map-type-modifier[,]...] map-
type:] locator-list)
depend([depend-modifier,] dependence-type: locator-list)
nowait
```

Map variables to a device data environment

- The host thread executes the data region
- Be careful when using the device clause

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

host target host target host

Synchronize mapped variables

- Synchronize the value of an original variable in a host data environment with a corresponding variable in a device data environment

```
#pragma omp target data map(alloc:tmp[:N]) map(to:input[:N]) map(tofrom:res)
{
    #pragma omp target
    #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 to(input[:N])

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

host

target

host

target

host

Code Examples

target data Construct

```
void vec_mult(float* p, float* v1, float* v2, int N)
{
    int i;
    init(v1, v2, N);

    #pragma omp target data map(from: p[0:N])
    {
        #pragma omp target map(to: v1[:N], v2[:N])
        #pragma omp parallel for
        for (i=0; i<N; i++)
            p[i] = v1[i] * v2[i];

        init_again(v1, v2, N);

        #pragma omp target map(to: v1[:N], v2[:N])
        #pragma omp parallel for
        for (i=0; i<N; i++)
            p[i] = p[i] + (v1[i] * v2[i]);

        output(p, N);
    }
}
```

- The **target data** construct maps variables to the *device data environment*.
 - structured mapping – the device data environment is created for the block of code enclosed by the construct
- v1 and v2 are mapped at each **target** construct.
- p is mapped once by the **target data** construct.

target enter/exit data Construct

```
void vec_mult(float* p, float* v1, float* v2, int N)
{
    int i;
    init(v1, v2, N);

    #pragma omp target map(to: v1[:N], v2[:N])
    #pragma omp parallel for
        for (i=0; i<N; i++)
            p[i] = v1[i] * v2[i];

    init_again(v1, v2, N);

    #pragma omp target map(to: v1[:N], v2[:N])
    #pragma omp parallel for
        for (i=0; i<N; i++)
            p[i] = p[i] + (v1[i] * v2[i]);

    output(p, N);
}
```

```
void init(float *v1, float *v2, int N) {
    for (int i=0; i<N; i++)
        v1[i] = v2[i] = ...;
    #pragma omp target enter data map(alloc: p[:N])
}

void output(float *p, int N) {
    ...
    #pragma omp target exit map(from: p[:N])
}
```

- The **target enter/exit data** construct maps variables to/from the *device data environment*.
 - unstructured mapping – the device data environment can span more than one function
- v1 and v2 are mapped at each **target** construct.
- p is allocated and remains undefined in the device data environment by the **target enter data map(alloc:...)** construct.
- The value of p in the *device data environment* is assigned to the original variable on the host by the **target exit data map(from:...)** construct.

Programming OpenMP

GPU: asynchronous offloading

Christian Terboven

Michael Klemm



Synchronization

- OpenMP target default: synchronous operations
 - CPU thread waits until OpenMP kernel/ movement is completed

- Remember:

- Use `target` construct to
 - Transfer control from the host to the target device
- Use `map` clause to
 - Map variables between the host and target device

- Host thread waits until offloaded region completed
 - Use the `nowait` clause for asynchronous execution

```
count = 500;
#pragma omp target map(to:b,c,d) map(from:a)
{
    #pragma omp parallel for
    for (i=0; i<count; i++) {
        a[i] = b[i] * c + d;
    }
}
a0 = a[0];
```

host

target

host

- Remember: GPUs only allow for synchronization within a streaming multiprocessor
 - Synchronization or memory fences across SMs not supported due to limited control logic
 - Barriers, critical regions, locks, atomics only apply to the threads within a team
 - No cache coherence between L1 caches

Remember: What is a Task in OpenMP?

- Tasks are work units whose execution
 - may be deferred or...
 - ... can be executed immediately
- Tasks are composed of
 - **code** to execute, a **data** environment (initialized at creation time), internal **control** variables (ICVs)
- Tasks are created...
 - ... when reaching a parallel region → implicit tasks are created (per thread)
 - ... when encountering a task construct → explicit task is created
 - ... when encountering a taskloop construct → explicit tasks per chunk are created
 - ... when encountering a target construct → target task is created

Remember: The task construct

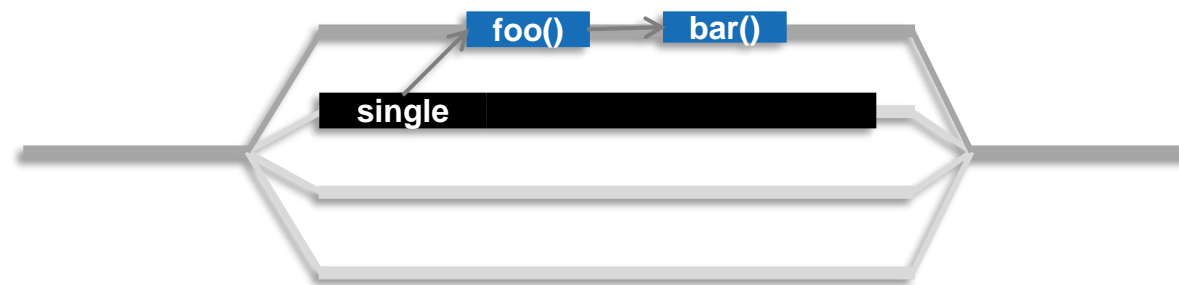
- Deferring (or not) a unit of work (executable for any member of the team)

```
#pragma omp task [clause[[,] clause]...]  
{structured-block}
```

```
!$omp task [clause[[,] clause]...]  
...structured-block...  
!$omp end task
```

- Task scheduling points (and the taskyield directive)

- tasks can be suspended/resumed at TSPs → some additional constraints to avoid deadlock problems
- implicit scheduling points (creation, synchronization, ...)



Asynchronous Offloading

- A host task is generated that encloses the target region.
- The **nowait** clause specifies that the encountering thread does not wait for the target region to complete.
- The **depend** clause can be used for ensuring the order of execution with respect to other tasks.

target task

A mergeable and untied task that is generated by a **target**, **target enter data**, **target exit data** or **target update** construct.

```
subroutine vec_mult(p, v1, v2, N)
  real, dimension(*) :: p, v1, v2
  integer :: N, i
  call init(v1, v2, N)

  !$omp target data map(tofrom:v1(1:N), v2(1:N), p(1:N))
  !$omp target nowait
  !$omp parallel do
    do i=1, N/2
      p(i) = v1(i) * v2(i)
    end do
  !$omp end target

  !$omp target nowait
  !$omp parallel do
    do i=N/2+1, N
      p(i) = v1(i) * v2(i)
    end do
  !$omp end target
  !$omp end target data

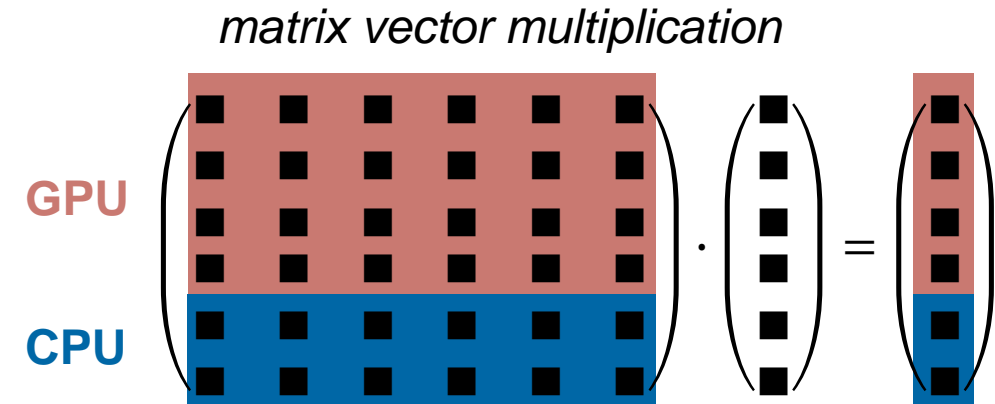
  call output(p, N)
end subroutine
```

Remark on Heterogeneous Computing

Slides are taken from the lecture High-Performance Computing at RWTH Aachen University
Authors include: Sandra Wienke, Julian Miller

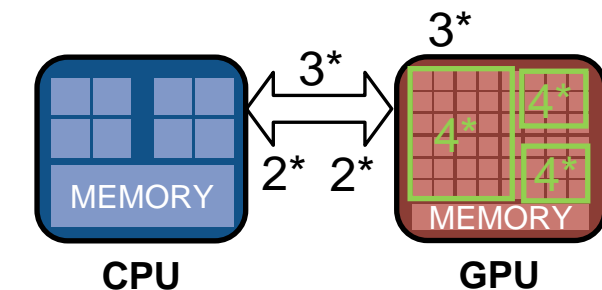
Heterogeneous Computing

- Heterogeneous Computing
 - CPU & GPU are (fully) utilized
- Challenge: load balancing
- Domain decomposition
 - If load is known beforehand, static decomposition
 - Exchange data if needed (e.g. halos)



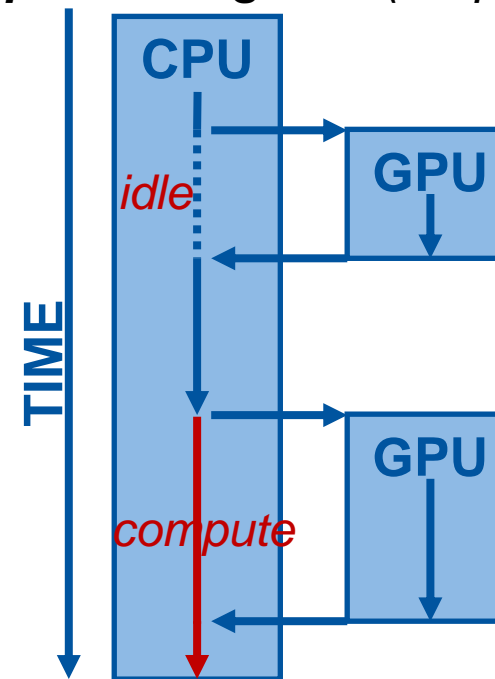
Asynchronous Operations

- Definition
 - Synchronous: Control does not return until accelerator action is complete
 - Asynchronous: Control returns immediately
- Asynchronicity allows, e.g.,
 1. Heterogeneous computing (CPU + GPU)
 2. Overlap of PCIe transfers in both directions
 3. Overlap of data transfers and computation
 4. Simultaneous execution of several kernels (if resources are available)



<num>* Can be executed simultaneously

processing flow (simplified)



Asynchronous Operations

- Default: synchronous operations
- Asynchronous operations with tasks
 - Execute asynchronously with dependency: `task depend`
 - Synchronize tasks: `taskwait`
- Synchronize async operations → `taskwait` directive
 - Wait for completion of an asynchronous activity

```
#pragma omp target map(...) nowait depend(out:gpu_data)
// do work on device
#pragma omp task depend(out:cpu_data)
// do work on host
#pragma omp task depend(in:cpu_data) depend(in:gpu_data)
// combine work on host
#pragma omp taskwait
// wait for all tasks
```

Code Examples

Tasks and Target Example / 1

```
void vec_mult_async(float* p, float* v1, float* v2, int N)
{
    #pragma omp target enter data map(alloc: v1[:N], v2[:N])

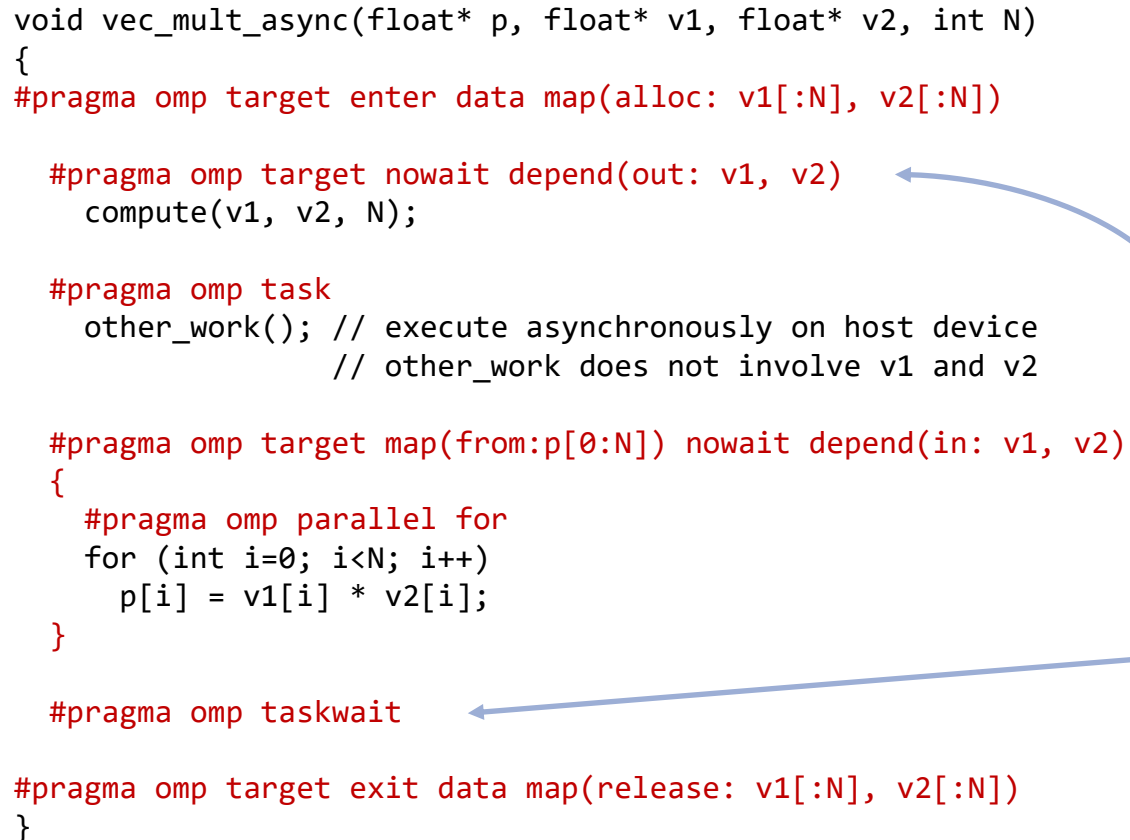
    #pragma omp target nowait depend(out: v1, v2)
        compute(v1, v2, N);

    #pragma omp task
        other_work(); // execute asynchronously on host device
                        // other_work does not involve v1 and v2

    #pragma omp target map(from:p[0:N]) nowait depend(in: v1, v2)
    {
        #pragma omp parallel for
        for (int i=0; i<N; i++)
            p[i] = v1[i] * v2[i];
    }

    #pragma omp taskwait

    #pragma omp target exit data map(release: v1[:N], v2[:N])
}
```



- If `other_work()` does not involve `v1` and `v2`, the encountering thread on the host will execute the task asynchronously.
- The dependency requirement between the two target tasks must be satisfied before the second target task starts execution.
- The **taskwait** directive ensures all sibling tasks complete before proceeding to the next statement.

Tasks and Target Example / 2

```
void vec_mult_async(float* p, float* v1, float* v2, int N)
{
    #pragma omp target enter data map(alloc: v1[:N], v2[:N])

    #pragma omp target nowait depend(out: v1, v2)
        compute(v1, v2, N);

    #pragma omp target update from(v1[:N], v2[:N]) depend(inout: v1, v2)

    #pragma omp task depend(inout: v1, v2)
        compute_on_host(v1, v2); // execute asynchronously on host device
                                // other_work involves v1, v2

    #pragma omp target update to(v1[:N], v2[:N]) depend(inout: v1, v2)

    #pragma omp target map(from:p[0:N]) nowait depend(in: v1, v2)
    {
        #pragma omp parallel for
        for (int i=0; i<N; i++)
            p[i] = v1[i] * v2[i];
    }

    #pragma omp taskwait

    #pragma omp target exit data map(release: v1[:N], v2[:N])
}
```

- If `compute_on_host()` updates `v1` and `v2`, the **depend** clause must be specified to ensure the execution of the target task and the explicit task respects the dependency.
- Since we update `v1` and `v2` on the host in `compute_on_host()`, we need to update the data results from `compute()` on the device to the host.
- After completion of `compute_on_host()`, the data in the target device is updated with the result.
- The **update** clause is required before and after the explicit task.



OpenMP Offload Programming: HIP & OpenMP Hybrid Programming

Dr.-Ing. Michael Klemm

Principal Member of Technical Staff
HPC Center of Excellence

Hybrid Programming

- Hybrid programming here stands for the interaction of OpenMP with a lower-level programming model, e.g.,
 - OpenCL
 - HIP
- OpenMP supports these interactions
 - Calling low-level HIP kernels from OpenMP application code
 - Calling OpenMP kernels from low-level application code
 - Interaction with the underlying asynchronous stream mechanism

HIP Buffer Management

```
void example() {  
    HIPCALL(hipSetDevice(0));  
  
    compute_1(n, x);  
    compute_2(n, y);  
  
    HIPCALL(hipMalloc(&x_dev, sizeof(*x_dev) * count));  
    HIPCALL(hipMalloc(&y_dev, sizeof(*y_dev) * count));  
    HIPCALL(hipMemcpy(x_dev, x, sizeof(*x) * count, hipMemcpyHostToDevice));  
    HIPCALL(hipMemcpy(y_dev, y, sizeof(*y) * count, hipMemcpyHostToDevice));  
  
    saxpy_omp(count, a, x_dev, y_dev);  
  
    HIPCALL(hipMemcpy(y, y_dev, sizeof(*y) * count, 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) * count));
    HIPCALL(hipMalloc(&y_dev, sizeof(*y_dev) * count));
    HIPCALL(hipMemcpy(x_dev, x, sizeof(*x) * count));
    HIPCALL(hipMemcpy(y_dev, y, sizeof(*y) * count));

    saxpy_omp(count, a, x_dev, y_dev);

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

    compute_3(n, y);
}

```

```

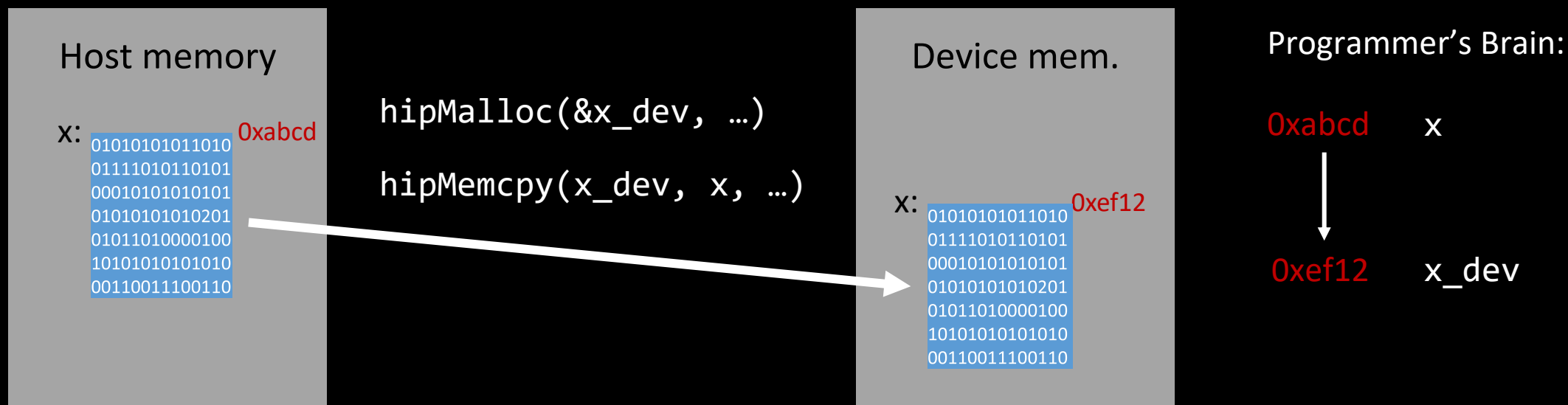
void saxpy_omp(size_t n, float a,
               float * x, float * y) {
    #pragma omp target teams distribute \
                    parallel for simd
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}

```

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

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(size_t n, float a, float * x, float * y) {  
    #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;
```

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

```
#pragma omp target data map(to:x[0:count]) map(tofrom:y[0:count])
```

```
{
```

```
    compute_1(n, x);
```

```
    compute_2(n, y);
```

```
    #pragma omp target update to(x[0:count])
```

```
    saxpy(n, a, x, y)
```

```
    compute_3(n, y);
```

```
}
```

```
}
```

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

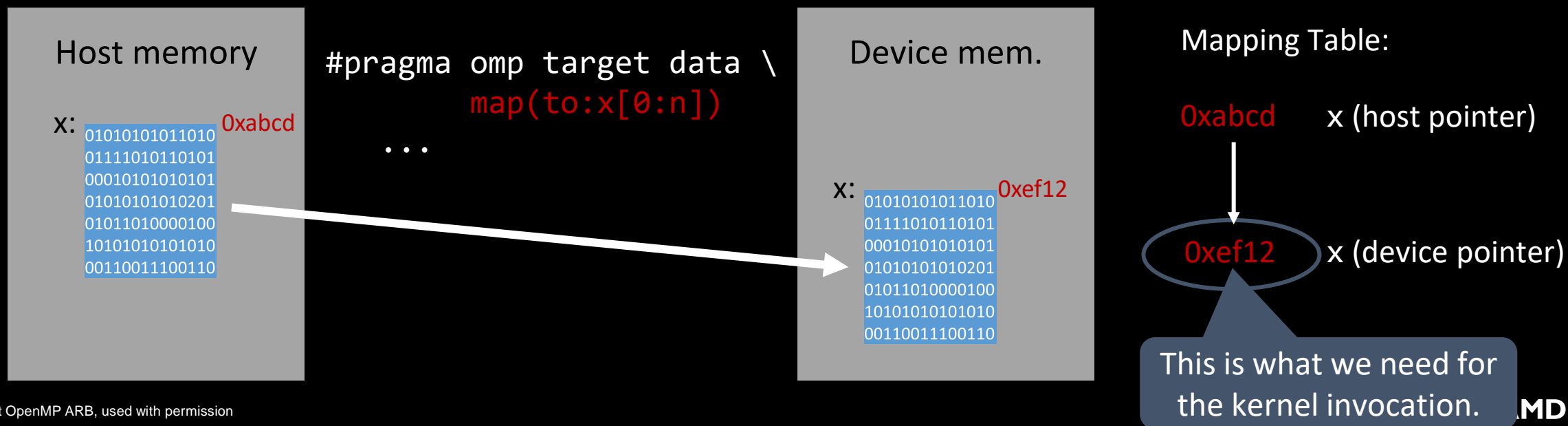
These are device pointers!

```
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);  
}
```

- 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_ptr` 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_ptr(x)  
{  
    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 = ...;

    // allocate the device memory
    #pragma omp target data map(to:x[0:count]) map(tofrom:y[0:count])
    {
        compute_1(n, x); // mapping table: x:[0xabcd,0xef12], x = 0xabcd
        compute_2(n, y);
        #pragma omp target update to(x[0:count]) to(y[0:count]) // update x and y on the target
        #pragma omp target data use_device_ptr(x,y)
        {
            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



Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED 'AS IS.' AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

© 2021 Advanced Micro Devices, Inc and OpenMP® Architecture Review Board. All rights reserved.

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

