Hybrid Programming

Hybrid Programming

- Hybrid programming here stands for the interaction of OpenMP with a lower-level programming model, e.g.
  - OpenCL
  - CUDA
  - HIP

Hybrid Programming

- Hybrid programming here stands for the interaction of OpenMP with a lower-level programming model, e.g.
  - OpenCL
  - CUDA
  - HIP

- OpenMP supports these interactions
  - Calling low-level kernels from OpenMP application code
  - Calling OpenMP kernels from low-level application code

Example: Calling saxpy

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

  // allocate the device memory
  #pragma omp target data map(to:x[0:count]) map(tofrom:y[0:count])
  {
    compute_1(n, x);
    compute_2(n, y);
             saxpy(n, a, x, y)
    compute_3(n, y);
  }
}
```

Example: Calling saxpy

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

  // allocate the device memory
  #pragma omp target data map(to:x[0:count]) map(tofrom:y[0:count])
  {
    compute_1(n, x);
    compute_2(n, y);
              saxpy(n, a, x, y)
    compute_3(n, y);
  }
}
```

```
void saxpy(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];
  }
}
```

Example: Calling saxpy

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

  // allocate the device memory
  #pragma omp target data map(to:x[0:count]) map(tofrom:y[0:count])
  {
    compute_1(n, x);
    compute_2(n, y);
          saxpy(n, a, x, y)
    compute_3(n, y);
  }
}
```

```
void saxpy(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];
  }
}
```

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

HIP Kernel for saxpy()

```
__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];
}

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);
}
```

HIP Kernel for saxpy()

```
__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];
}

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!

HIP Kernel for saxpy()

- Assume a HIP version of the SAXPY kernel:

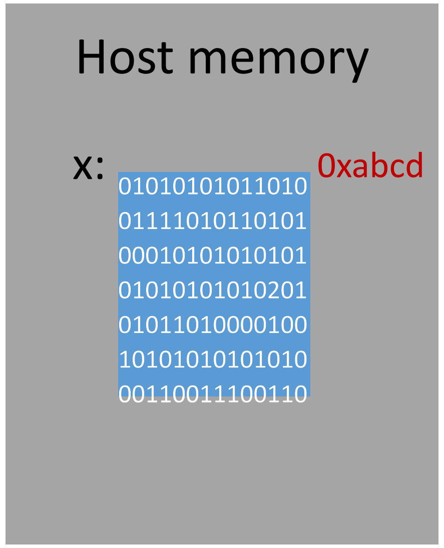
```
__global__ void saxpy_kernel(size_t n, float a, float * x, float * y) {
  size_t i = threadIdx.x + blockIdx.x * blockDim.x;
  y[i] = a * x[i] + y[i];
}

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.

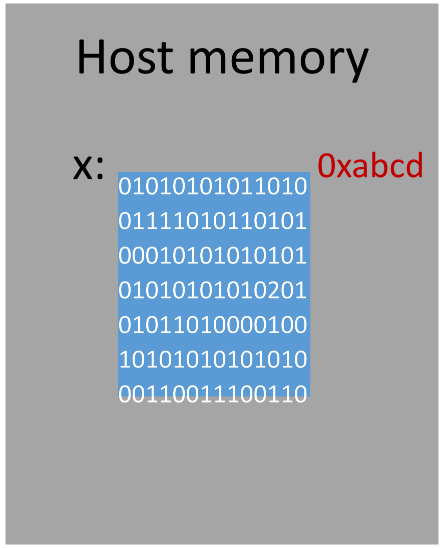
Pointer Translation /1

- 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.



Pointer Translation /1

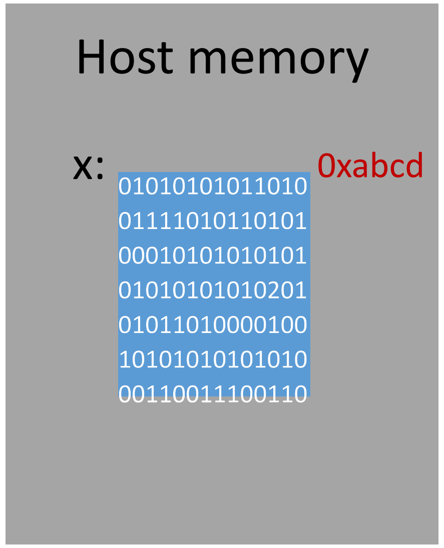
- When creating the device data environment, OpenMP creates a mapping between
  - the (virtual) memory pointer on the host and
  - the (virtual) memory pointer on the target device.
- This mapping is established through the data-mapping directives and their clauses.



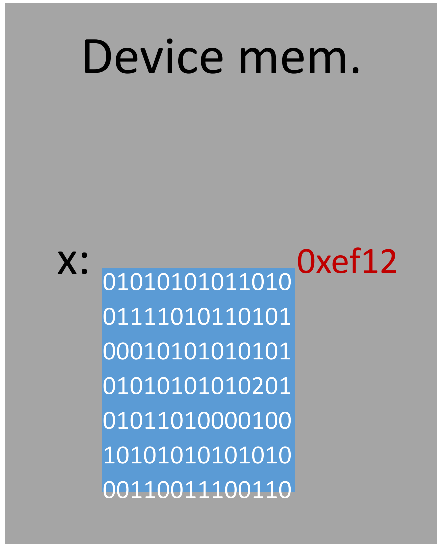
```
#pragma omp target data \
    map(to:x[0:n])
 ...
!$omp end target data
```

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.

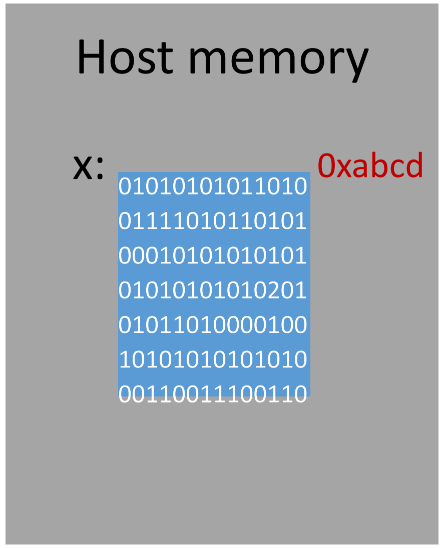


```
#pragma omp target data \
    map(to:x[0:n])
 ...
!$omp end target data
```

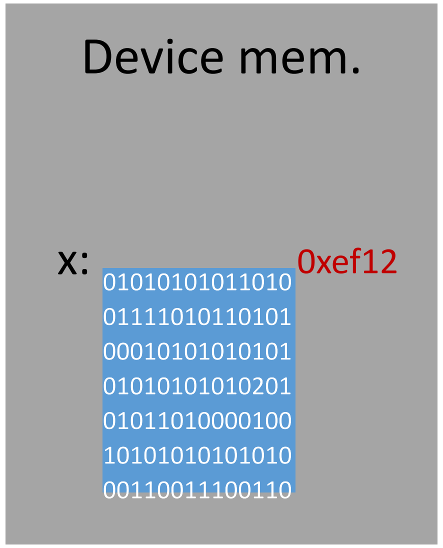


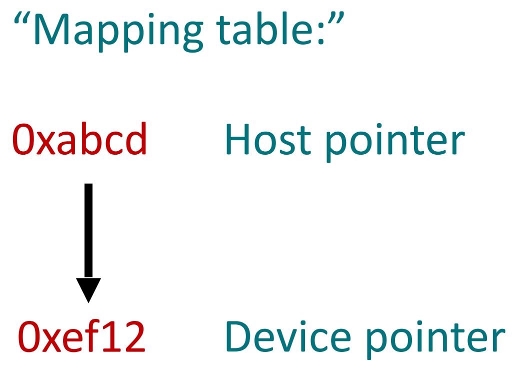
Pointer Translation /1

- 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.



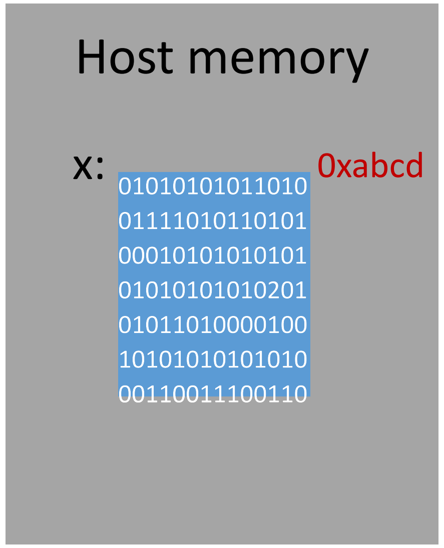
```
#pragma omp target data \
    map(to:x[0:n])
 ...
!$omp end target data
```



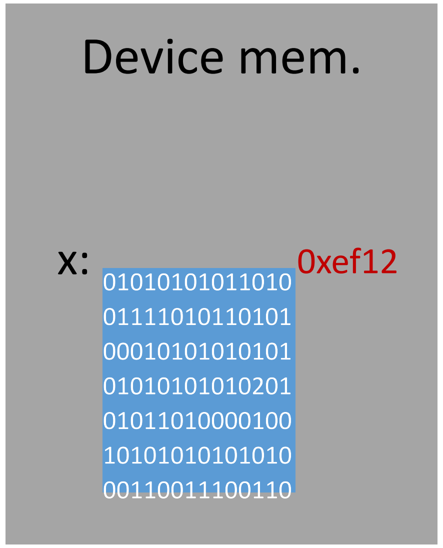


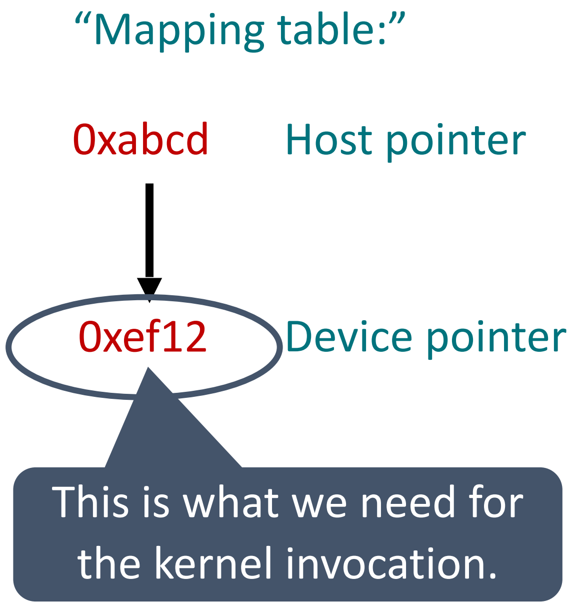
Pointer Translation /1

- 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.



```
#pragma omp target data \
    map(to:x[0:n])
 ...
!$omp end target data
```





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 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 data use_device_ptr(x,y)
            {
              saxpy_hip(n, a, x, y) // mapping table: x:[0xabcd,0xef12], x = 0xef12
    }
    compute_3(n, y);
  }
}
```

Advanced Task Synchronization

Asynchronous API Interaction

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

Asynchronous API Interaction

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

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

Asynchronous API Interaction

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

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.

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);
  }
}
```

Try 2: Use just OpenMP Tasks Dependences

```
void hip_example(){
#pragma omp task depend(out:stream)  // task A
  {
    do_something();
    hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHos, 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

Try 2: Use just OpenMP Tasks Dependences

```
void hip_example(){
#pragma omp task depend(out:stream)   // task A
  {
    do_something();
    hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHos , 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.
  - may be problematic if 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

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_t datatype

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_t datatype
- Detached task clause: detach(event)

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_t datatype
- Detached task clause: detach(event)
- Runtime API: void omp_fulfill_event(omp_event_t *event)

Detaching Tasks

```
omp_event_t *event;
void detach_example() {
#pragma omp task detach(event)
  {
    important_code();
  }

  #pragma omp taskwait
}
```

Detaching Tasks

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

  #pragma omp taskwait
}
```

1. Task detaches

Detaching Tasks

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

  #pragma omp taskwait          ②
}
```

1. Task detaches
2. taskwait construct cannot complete

Detaching Tasks

```
omp_event_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

Detaching Tasks

```
omp_event_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) {
    omp_fulfill_event((omp_event_t *) cb_data);
}
void hip_example() {
  omp_event_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);
  }
#pragma omp task          // task B
    do_something_else();

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

Putting It All Together

```
void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
    omp_fulfill_event((omp_event_t *) cb_data);
}
void hip_example() {
  omp_event_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
#pragma omp task          // task C
  {
    do_other_important_stuff(dst);
} }
```

1. Task A detaches

Putting It All Together

```
void callback(hipStream_t stream, hipError_t status, void *cb_dat){
    omp_fulfill_event((omp_event_t *) cb_data);
}
void hip_example(){
  omp_event_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
#pragma omp task          // task C
  {
    do_other_important_stuff(dst);
} }
```

1. Task A detaches
2. taskwait does not continue

Putting It All Together

```
void callback(hipStream_t stream, hipError_t status, void *cb_dat){
  ③ omp_fulfill_event((omp_event_t *) cb_data);
}
void hip_example(){
  omp_event_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
#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

Putting It All Together

```
void callback(hipStream_t stream, hipError_t status, void *cb_dat){
  ③omp_fulfill_event((omp_event_t *) cb_data);
}
void hip_example(){
  omp_event_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      ②④
#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) {
     omp_fulfill_event((omp_event_t *) cb_data);
}
void hip_example() {
  omp_event_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);
  }
#pragma omp task            // task B
    do_something_else();

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

Removing the taskwait Construct

```
void callback(hipStream_t stream, hipError_t status, void *cb_dat){
    omp_fulfill_event((omp_event_t *) cb_data);
}
void hip_example(){
  omp_event_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)  // task C
  {
    do_other_important_stuff(dst);
} }
```

1. Task A detaches and task C will not execute because of its unfulfilled dependency on A

Removing the taskwait Construct

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

#pragma omp task depend(in:dst)   // 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

Removing the taskwait Construct

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

#pragma omp task depend(in:dst)   // task C   ③
  {
    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

Case Study: NWChem TCE CCSD(T)

TCE:	Tensor Contraction Engine
CCSD(T):	Coupled-Cluster with Single, Double, and perturbative Triple replacements

NWChem

- Computational chemistry software package
  - Quantum chemistry
  - Molecular dynamics
- Designed for large-scale supercomputers
- Developed at the EMSL at PNNL
  - EMSL: Environmental Molecular Sciences Laboratory
  - PNNL: Pacific Northwest National Lab
- URL: http://www.nwchem-sw.org

Finding Offload Candidates

- Requirements for offload candidates
  - Compute-intensive code regions (kernels)
  - Highly parallel
  - Compute scaling stronger than data transfer, e.g., compute $O(n^3)$ vs. data size $O(n^2)$

Example Kernel (1 of 27 in total)

```
     subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
     1         h7d,triplesx,t2sub,v2sub)
c    Declarations omitted.
     double precision triplesx(h3d*h2d,h1d,p6d,p5d,p4d)
     double precision t2sub(h7d,p4d,p5d,h1d)
     double precision v2sub(h3d*h2d,p6d,h7d)
!$omp target „presence?(triplesx,t2sub,v2sub)"
!$omp teams distribute parallel do private(p4,p5,p6,h2,h3,h1,h7)
     do p4=1,p4d
     do p5=1,p5d
     do p6=1,p6d
     do h1=1,h1d
     do h7=1,h7d
     do h2h3=1,h3d*h2d
      triplesx(h2h3,h1,p6,p5,p4)=triplesx(h2h3,h1,p6,p5,p4)
     1 - t2sub(h7,p4,p5,h1)*v2sub(h2h3,p6,h7)
     end do
     end do
     end do
     end do
     end do
     end do
!$omp end teams distribute parallel do
!$omp end target
     end subroutine
```

- All kernels have the same structure
- 7 perfectly nested loops
- Some kernels contain inner product loop (then, 6 perfectly nested loops)
- Trip count per loop is equal to “tile size” (20-30 in production)

Example Kernel (1 of 27 in total)

```
      subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
     1         h7d,triplesx,t2sub,v2sub)
c     Declarations omitted.
      double precision triplesx(h3d*h2d,h1d,p6d,p5d,p4d)
      double precision t2sub(h7d,p4d,p5d,h1d)
      double precision v2sub(h3d*h2d,p6d,h7d)
!$omp target „presence?(triplesx,t2sub,v2sub)"
!$omp teams distribute parallel do private(p4,p5,p6,h2,h3,h1,h7)
      do p4=1,p4d
      do p5=1,p5d
      do p6=1,p6d
      do h1=1,h1d
      do h7=1,h7d
      do h2h3=1,h3d*h2d
       triplesx(h2h3,h1,p6,p5,p4)=triplesx(h2h3,h1,p6,p5,p4)
     1  – t2sub(h7,p4,p5,h1)*v2sub(h2h3,p6,h7)
      end do
      end do
      end do
      end do
      end do
      end do
!$omp end teams distribute parallel do
!$omp end target
      end subroutine
```

- All kernels have the same structure
- 7 perfectly nested loops
- Some kernels contain inner product loop (then, 6 perfectly nested loops)
- Trip count per loop is equal to “tile size” (20-30 in production)
- Naïve data allocation (tile size 24)
  - Per-array transfer for each target construct
  - triplesx: 1458 MB
  - t2sub, v2sub: 2.5 MB each

Example Kernel (1 of 27 in total)

```
      subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
     1         h7d,triplesx,t2sub,v2sub)
c   Declarations omitted.
      double precision triplesx(h3d*h2d,h1d,p6d,p5d,p4d)
      double precision t2sub(h7d,p4d,p5d,h1d)
      double precision v2sub(h3d*h2d,p6d,h7d)
!$omp target „presence?(triplesx,t2sub,v2sub)"
!$omp teams distribute parallel do private(p4,p5,p6,h2,h3,h1,h7)
      do p4=1,p4d
      do p5=1,p5d
      do p6=1,p6d
      do h1=1,h1d
      do h7=1,h7d
      do h2h3=1,h3d*h2d
       triplesx(h2h3,h1,p6,p5,p4)=triplesx(h2h3,h1,p6,p5,p4)
     1 - t2sub(h7,p4,p5,h1)*v2sub(h2h3,p6,h7)
      end do
      end do
      end do
      end do
      end do
      end do
!$omp end teams distribute parallel do
!$omp end target
      end subroutine
```

1.5GB data transferred
(host to device)

1.5GB data transferred
(device to host)

- All kernels have the same structure
- 7 perfectly nested loops
- Some kernels contain inner product loop (then, 6 perfectly nested loops)
- Trip count per loop is equal to "tile size" (20-30 in production)
- Naïve data allocation (tile size 24)
  - Per-array transfer for each target construct
  - triplesx: 1458 MB
  - t2sub, v2sub: 2.5 MB each

Invoking the Kernels / Data Management

- Simplified pseudo-code

```
!$omp target enter data map(alloc:triplesx(1:tr_size))
c   for all tiles
   do ...
    call zero_triplesx(triplesx)
    do ...
     call comm_and_sort(t2sub, v2sub)
!$omp target data map(to:t2sub(t2_size)) map(to:v2sub(v2_size))
     if (...)
      call sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,h7,triplesx,t2sub,v2sub)
     end if
c     same for sd_t_d1_2 until sd_t_d1_9
!$omp target end data
    end do
    do ...
c     Similar structure for sd_t_d2_1 until sd_t_d2_9, incl. target data
    end do
    call sum_energy(energy, triplesx)
   end do
!$omp target exit data map(release:triplesx(1:size))
```

- Reduced data transfers:

Invoking the Kernels / Data Management

- Simplified pseudo-code

```
!$omp target enter data map(alloc:triplesx(1:tr_size))
c   for all tiles
   do ...
    call zero_triplesx(triplesx)
    do ...
     call comm_and_sort(t2sub, v2sub)
!$omp target data map(to:t2sub(t2_size)) map(to:v2sub(v2_size))
     if (...)
      call sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,h7,triplesx,t2sub,v2sub)
     end if
c     same for sd_t_d1_2 until sd_t_d1_9
!$omp target end data
    end do
    do ...
c     Similar structure for sd_t_d2_1 until sd_t_d2_9, incl. target data
    end do
    call sum_energy(energy, triplesx)
   end do
!$omp target exit data map(release:triplesx(1:size))
```

Allocate 1.5GB data once, stays on device.

- Reduced data transfers:
  - triplesx:
    - allocated once
    - always kept on the target

Invoking the Kernels / Data Management

- Simplified pseudo-code

```
!$omp target enter data map(alloc:triplesx(1:tr_size))
c   for all tiles
   do ...
    call zero_triplesx(triplesx)
    do ...
     call comm_and_sort(t2sub, v2sub)
!$omp target data map(to:t2sub(t2_size)) map(to:v2sub(v2_size))
     if (...)
      call sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,h7,triplesx,t2su       ub)
     end if
c     same for sd_t_d1_2 until sd_t_d1_9
!$omp target end data
    end do
    do ...
c     Similar structure for sd_t_d2_1 until sd_t_d2_9, incl. target data
    end do
    call sum_energy(energy, triplesx)
   end do
!$omp target exit data map(release:triplesx(1:size))
```

Allocate 1.5GB data once, stays on device.

Update 2x2.5MB of data for (potentially) multiple kernels.

- Reduced data transfers:
  - triplesx:
    - allocated once
    - always kept on the target
  - t2sub, v2sub:
    - allocated after comm.
    - kept for (multiple) kernel invocations

Invoking the Kernels / Data Management

- Simplified pseudo-code

```
!$omp target enter data map(alloc:triplesx(1:tr_size))
c   for all tiles
   do ...
    call zero_triplesx(triplesx)
    do ...
     call comm_and_sort(t2sub, v2sub)
!$omp target data map(to:t2sub(t2_size)) map(to:v2sub(v2_size))
     if (...)
      call sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,h7,triplesx,t2sub,v2sub)
     end if
c     same for sd_t_d1_2 until sd_t_d1_9
!$omp target end data
    end do
    do ...
c     Similar structure for sd_t_d2_1 until sd_t_d2_9, incl. target data
    end do
    call sum_energy(energy, triplesx)
   end do
!$omp target exit data map(release:triplesx(1:size))
```

Allocate 1.5G
stays on

Update 2x2.5
(potentially) r

```
  subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
 1        h7d,triplesx,t2sub,v2sub)
c  Declarations omitted.
  double precision triplesx(h3d*h2d,h1d,p6d,p5d,p4d)
  double precision t2sub(h7d,p4d,p5d,h1d)
  double precision v2sub(h3d*h2d,p6d,h7d)
!$omp target „presence?(triplesx,t2sub,v2sub)"
!$omp teams distribute parallel do private(p4,p5,p6,h2,h3,h1,h7)
  do p4=1,p4d
  do p5=1,p5d
  do p6=1,p6d
  do h1=1,h1d
  do h7=1,h7d
  do h2h3=1,h3d*h2d
   triplesx(h2h3,h1,p6,p5,p4)=triplesx(h2h3,h1,p6,p5,p4)
 1 - t2sub(h7,p4,p5,h1)*v2sub(h2h3,p6,h7)
  end do
  end do
  end do
  end do
  end do
  end do
!$omp end teams distribute parallel do
!$omp end target
  end subroutine
```

Invoking the Kernels / Data Management

- Simplified pseudo-code

```
!$omp target enter data map(alloc:triplesx(1:tr_size))
c   for all tiles
   do ...
    call zero_triplesx(triplesx)
    do ...
     call comm_and_sort(t2sub, v2sub)
!$omp target data map(to:t2sub(t2_size)) map(to:v2sub(v2_size))
     if (...)
      call sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,h7,triplesx,t2su    ub)
     end if
c     same for sd_t_d1_2 until sd_t_d1_9
!$omp target end data
    end do
    do ...
c     Similar structure for sd_t_d2_1 until sd_t_d2_9, incl. target data
    end do
    call sum_energy(energy, triplesx)
   end do
!$omp target exit data map(release:triplesx(1:size))
```

Allocate 1.5G
stays on

Update 2x2.5
(potentially) r

```
  subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
1        h7d,triplesx,t2sub,v2sub)
c   Declarations omitted.
  double precision triplesx(h3d*h2d,h1d,p6d,p5d,p4d)
  double precision t2sub(h7d,p4d,p5d,h1d)
  double precision v2sub(h3d*h2d,p6d,h7d)
!$omp target „presence?(triplesx,t2sub,v2sub)"
!$omp teams distribute parallel do private(p4,p5,p6,h2,h3,h1,h7)
  do p4=1,p4d
  do p5=1,p5d
  do p6=1,p6d
  do h1=1,h1d
  do h7=1,h7d
  do h2h3=1,h3d*h2d
   triplesx(h2h3,h1,p6,p5,p
1  – t2sub(h7,p4,p5,h1)*v2
  end do
  end do
  end do
  end do
  end do
  end do
!$omp end teams distribute parallel do
!$omp end target
  end subroutine
```

Presence check determines that arrays have been allocated in the device data environment already.