Programming OpenMP

Christian Terboven

Michael Klemm

Agenda (in total 5 webinars)

- Webinar 1: OpenMP Introduction
- Webinar 2: Tasking
- Webinar 3: Optimization for NUMA and SIMD
- Webinar 4: Introduction to Offloading with OpenMP
- **Webinar 5: Advanced Offloading Topics**
  - →Review of webinar 4 / homework assignments
  - →Unstructured Data Movement
  - →Reducing Data Transfers
  - →HALO Exchange
  - →Asynchronous Offloading
  - →Real-World Application Case Study: NWChem
  - →Integration of GPU-Kernels (i.e., HIP)
  - →Homework assignments ☺

Programming OpenMP

Hands-on Exercises: Stream and Jacobi

Christian Terboven
Michael Klemm

My Setup on CLAIX

```
=> nvidia-smi
Thu May 20 17:21:08 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:61:00.0 Off |                    0 |
| N/A   39C    P0    53W / 300W |      0MiB / 16160MiB |      0%   E. Process |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  Off  | 00000000:62:00.0 Off |                    0 |
| N/A   37C    P0    53W / 300W |      0MiB / 16160MiB |      0%   E. Process |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

- clang 12.0.0 with gcc 4.8.5 from CentOS 7.9.2009

Jacobi on GPU / 1

- Task 0: You might want to acquire reference measurements on the host (wo/ GPU)...
  - Skipped…
- Task 1: Get it to the GPU: Parallelize only the one most compute-intensive loop

```
Jacobi relaxation Calculation: 16384 x 16384 mesh with 1 threads and at most 100 iterations. 0 rows out of 16384 on CPU.
    0, 0.250000
   10, 0.250000
   20, 0.250000
   30, 0.250000
   40, 0.250000
   50, 0.250000
   60, 0.250000
   70, 0.250000
   80, 0.250000
   90, 0.250000
 total: 144.992748 s
```

Jacobi on GPU / 2

- Task 2: Improve the data management and the amount of parallelism on the GPU

```
=> ./jacobi.sol.gpu-v100
Jacobi relaxation Calculation: 16384 x 16384 mesh with 1 threads. 0 rows out of 16384 on CPU.
    0, 0.250000
   10, 0.021563
   20, 0.011489
   30, 0.007826
   40, 0.005857
   50, 0.004751
   60, 0.003945
   70, 0.003412
   80, 0.002980
   90, 0.002658
 total: 7.872561 s
```

- Task 3: Optimize that scheduling of iterations for the GPU

```
=> ./jacobi.sol.gpu-v100
Jacobi relaxation Calculation: 16384 x 16384 mesh with 1 threads. 0 rows out of 16384 on CPU.
    0, 0.250000
   10, 0.021563
   20, 0.011489
   30, 0.007826
   40, 0.005857
   50, 0.004751
   60, 0.003945
   70, 0.003412
   80, 0.002980
   90, 0.002658
 total: 5.519289 s
```

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)
}
```

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

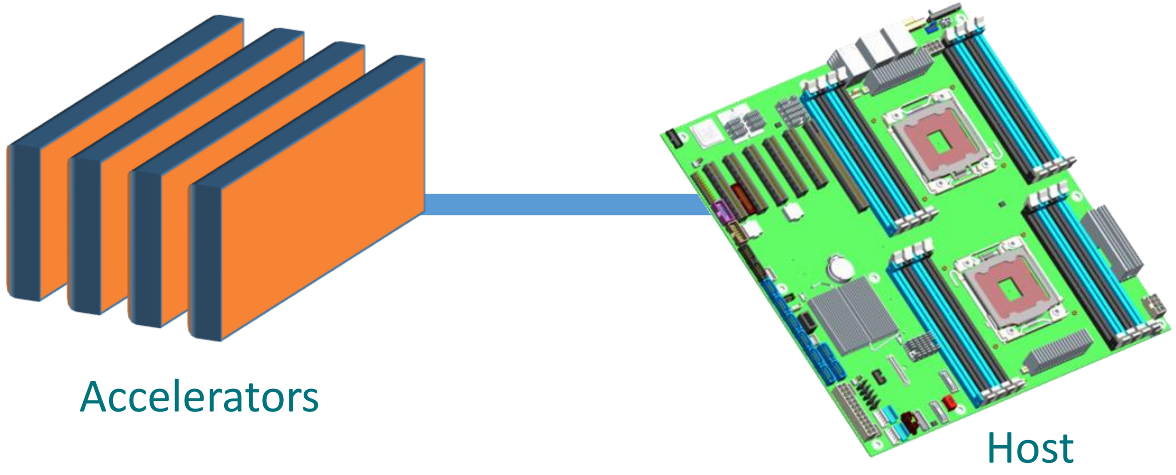
```
void init(float *v1, float *v2, int N) {
  for (int i=0; i<N; i++)
    v1[i] = v2[i] = ...;
#pragma omp target enter data map(alloc: p[:N])
}

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

```
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);
}
```

- 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.

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: 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

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

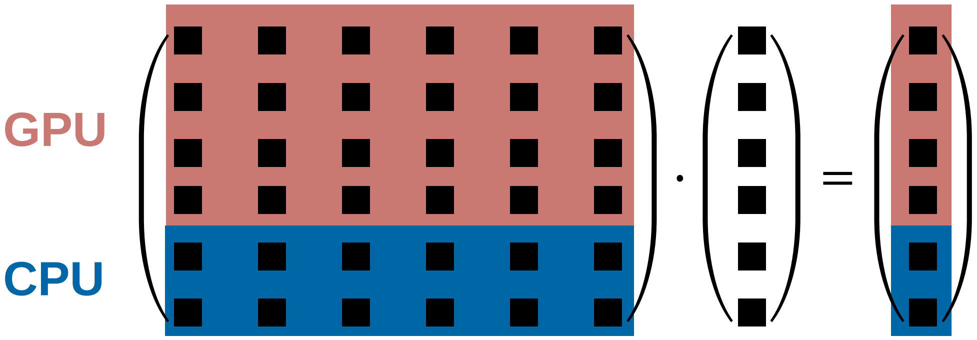
Heterogeneous Computing

- Heterogeneous Computing
  - CPU & GPU are (fully) utilized

- Challenge: load balancing

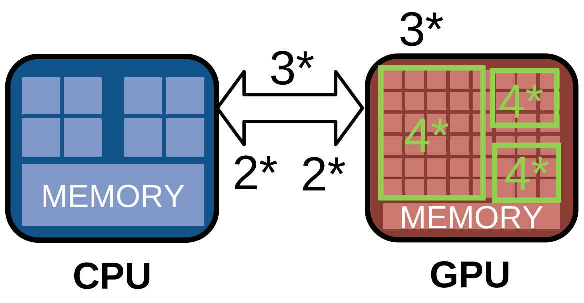
- Domain decomposition
  - If load is known beforehand, static decomposition
  - Exchange data if needed (e.g. halos)

matrix vector multiplication

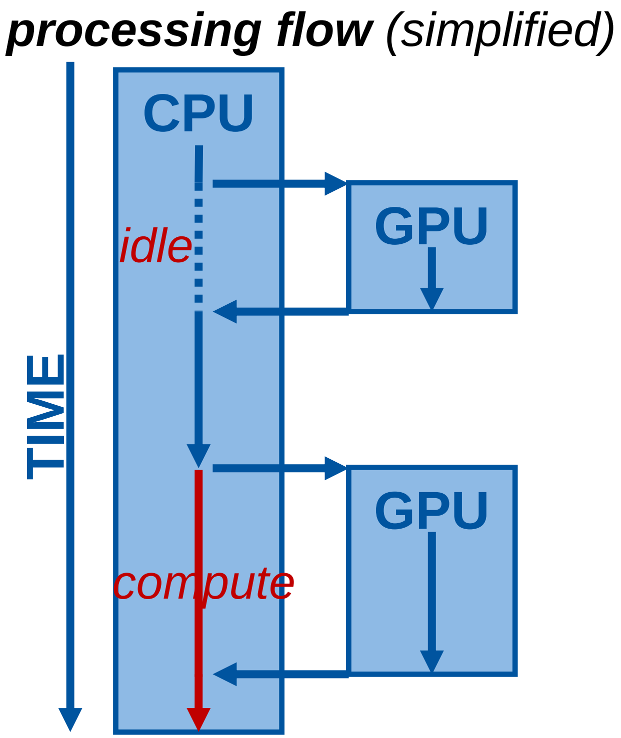


Asynchronous Operations

- 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



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.

Hybrid Programming

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);
    }
}
```

```
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()

- Assume a HIP version of the SAXPY kernel:

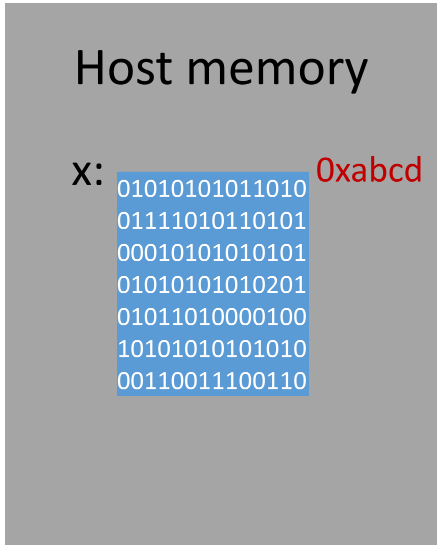
```
__global__ void saxpy_kernel(size_t n, float a, float * x, float * y) {
    size_t i = threadIdx.x + blockIdx.x * blockDim.x;
    y[i] = a * x[i] + y[i];
}

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!

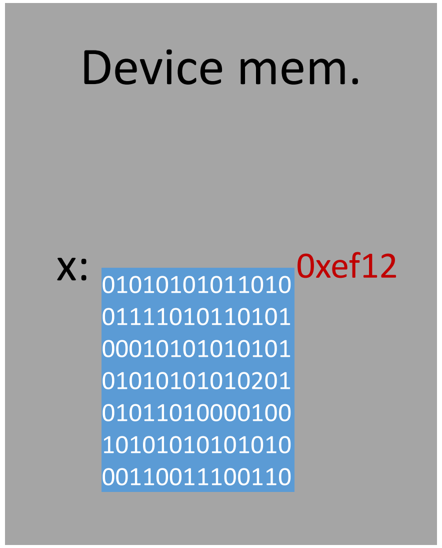
- We need a way to translate the host pointer that was mapped by OpenMP directives and retrieve the associated device pointer.

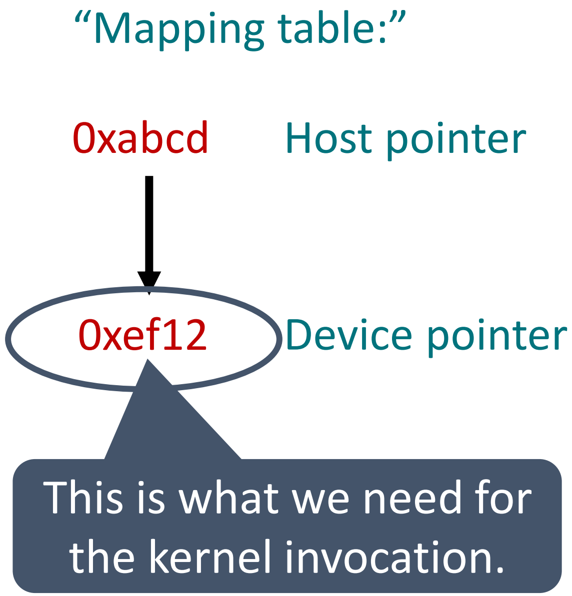
Pointer Translation /1

- 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.
- This mapping is established through the data-mapping directives and their clauses.




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





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 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)
- 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);
    }
}
```

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.
  - 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

- 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();
    } ①

    #pragma omp taskwait ② ④
}
```

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_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 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_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)      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

OpenMP®

Enabling HPC since 1997

Visit www.openmp.org for more information

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
```

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,p…4d,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.

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,p5 4d,h7,triplesx
          end if
c         same for sd_t_d1_2 until sd_t_d1_9
!$omp target end data
        end do
        do ...
c         Similar structure for sd_t_d2_1 until sd_t_d2_9, inc
        end do
        call sum_energy(energy, triplesx)
      end do
!$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
       triplesx(h2h
     1   - t2sub(h7
      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.

OpenMP®

Enabling HPC since 1997

Visit www.openmp.org for more information

Programming OpenMP

Hands-on Exercises: Stream and Jacobi

Christian Terboven
Michael Klemm

Jacobi on GPU

- Task 0: You might want to acquire reference measurements on the host (wo/ GPU)...
- Task 1: Get it to the GPU: Parallelize only the one most compute-intensive loop
- Task 2: Improve the data management and the amount of parallelism on the GPU
- Task 3: Optimize that scheduling of iterations for the GPU
- Task 4: Make the code as fast as you can :-)