Asynchronous GPU Programming in OpenMP

Christian Terboven, RWTH Aachen University
Michael Klemm, OpenMP ARB

HORIZON-EUROHPC-JU-2023-COE

EuroHPC Joint Undertaking

1 January 2024– 31 December 2026

Grant Agreement No 101143931

Agenda

- Who are Michael and Christian?
- Review: OpenMP device and execution model
- Review: Offloading in OpenMP
- Optimizing data transfers and asynchronous offloading
- Hybrid OpenMP and HIP (or CUDA)
- Advanced Task Synchronization

Who are Michael and Christian?

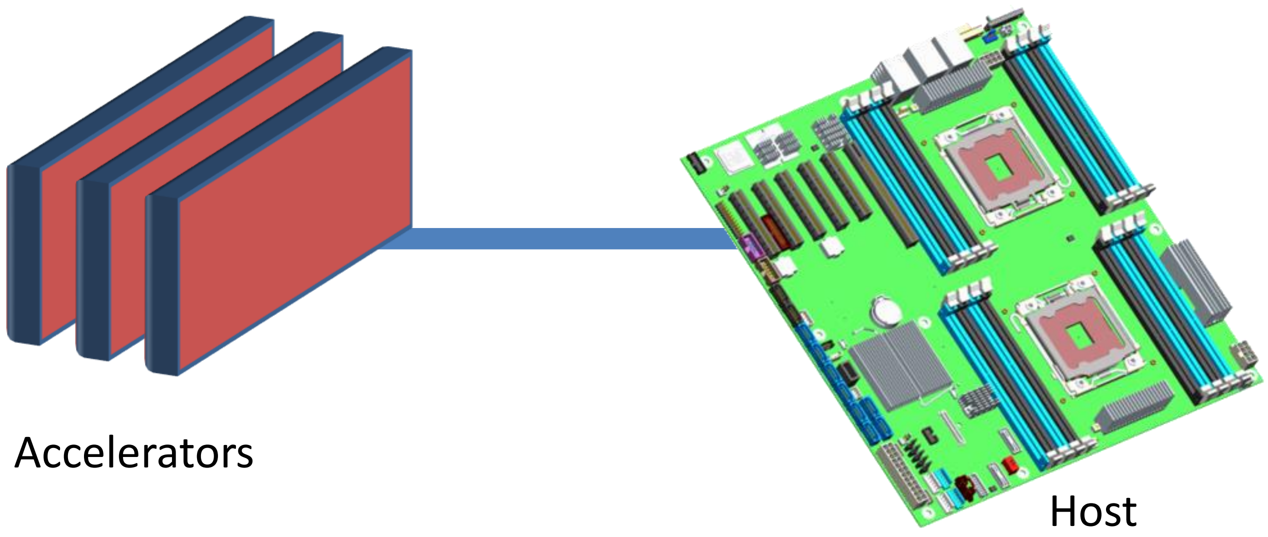
Michael and Christian

- Michael …
  - → Principal Member of Technical Staff at AMD
  - → Works in HPC since 2003
  - → Works on the Fortran OpenMP offload compiler for AMD Instinct™ Accelerators
  - → Is a member of the OpenMP language committee since 2009
  - → Chief Executive Officer of the OpenMP ARB since April 2016
- Christian …
  - → … is a senior scientist at RWTH Aachen University and leads the HPC group
  - → … does research on Parallel Programming and Performance
  - → … is a member of the OpenMP language committee since 2008 and co-chair of the Affinity subcom.
  - → is co-author of the book "Using OpenMP - The Next Step", published by MIT Press

Review: OpenMP device and execution model

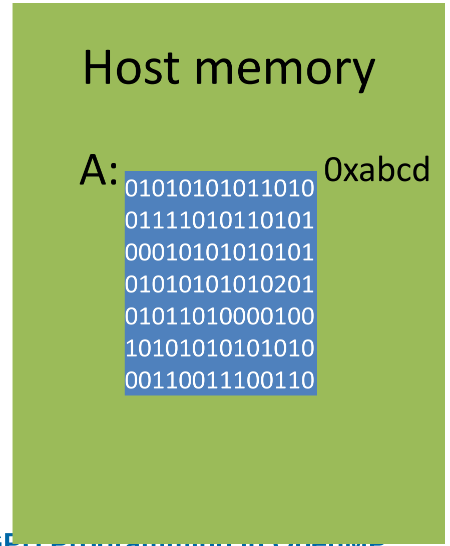
Device Model

- As of version 4.0 the OpenMP API supports accelerators/coprocessors
- Device model:
  - →One host for “traditional” multi-threading
  - →Multiple accelerators/coprocessors of the same kind for offloading



OpenMP Execution Model for Devices

- Offload region and its data environment are bound to the lexical scope of the construct
  - → Data environment is created at the opening curly brace / begin of target
  - → Data environment is automatically destroyed at the closing curly brace / end target
  - → Data transfers (if needed) are done at the curly braces / begin or end, 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
```

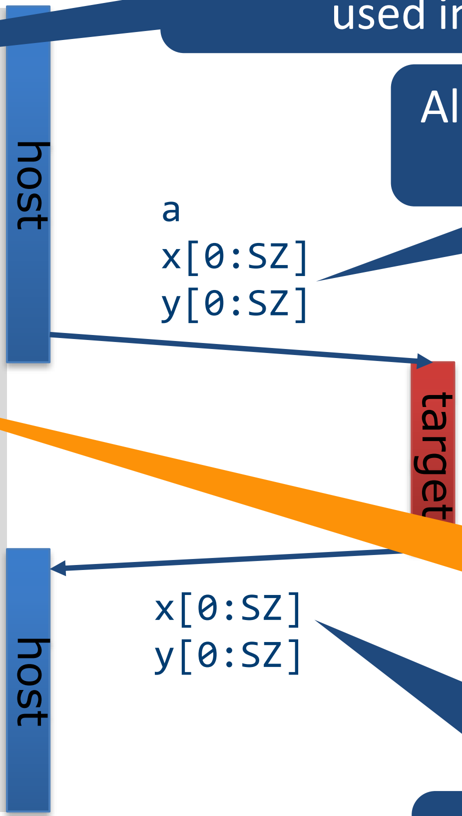
Review: Offloading in OpenMP

Example: saxpy

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

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

All accessed arrays are copied from host to device and back



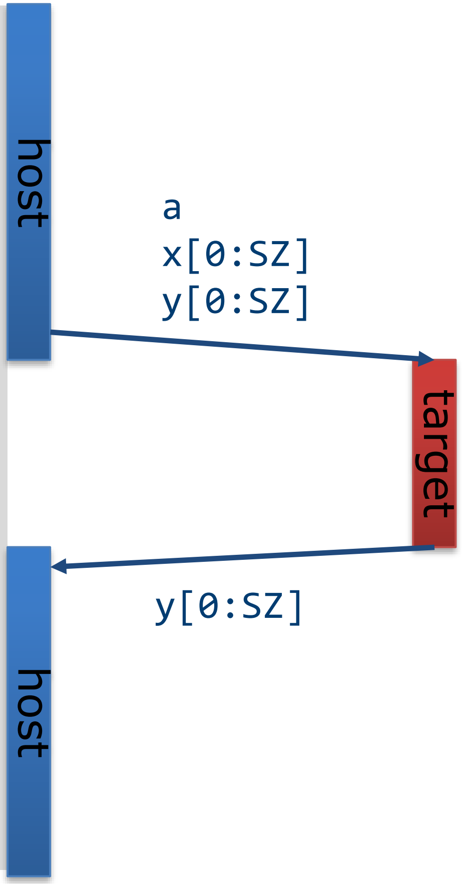
Presence check: only transfer if not yet allocated on the device.

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

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

Example: saxpy

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

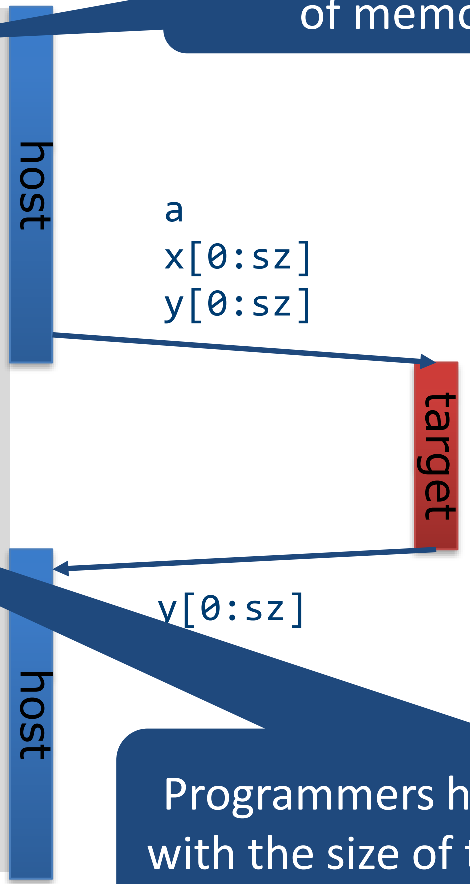



```
clang -fopenmp --offload-arch=gfx90a ...
```

Example: saxpy

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

The compiler cannot determine the size of memory behind the pointer.



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

```
clang -fopenmp --offload-arch=gfx90a
```

Creating Parallelism on the Target Device

- The `target` construct transfers the control flow to the target device
  - →Transfer of control is sequential and synchronous
  - →This is intentional!

- OpenMP separates offload and parallelism
  - →Programmers need to explicitly create parallel regions on the target device
  - →In theory, this can be combined with any OpenMP construct
  - →In practice, there is only a useful subset of OpenMP features for a target device such as a GPU, e.g., no I/O, limited use of base language features.

Example: saxpy

```
void saxpy(float a, float* x, float* y,
           int sz) {
#pragma omp target map(to:x[0:sz]) \
                   map(tofrom(y[0:sz])
#pragma omp parallel for simd
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

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.

```
clang -fopenmp --offload-arch=gfx90a
```

Multi-level Parallel saxpy

- Manual code transformation
  - → Tile the loop into an outer loop and an inner loop.
  - → Assign the outer loop to “teams”.
  - → Assign the inner loop to the “threads”.
  - → (Assign the inner loop to SIMD units.)

```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target teams map(to:x[0:sz]) map(tofrom:y[0:sz]) num_teams(nteams)
    {
        int bs = n / omp_get_num_teams();   // n assumed to be multiple of #teams
        #pragma omp distribute
        for (int i = 0; i < sz; i += bs) {
            #pragma omp parallel for simd firstprivate(i,bs)
            for (int ii = i; ii < i + bs; ii++) {
                y[ii] = a * x[ii] + y[ii];
}   }   }   }
```

Multi-level Parallel saxpy

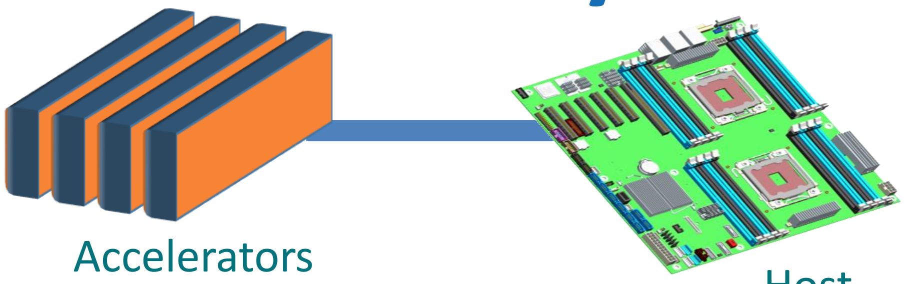
- For convenience, OpenMP defines composite constructs to implement the required code transformations

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

```
subroutine saxpy(a, x, y, n)
    ! Declarations omitted
!$omp omp target teams distribute parallel do simd &
!$omp&          num_teams(num_blocks) map(to:x) map(tofrom:y)
    do i=1,n
        y(i) = a * x(i) + y(i)
    end do
!$omp end target teams distribute parallel do simd
end subroutine
```

Optimizing data transfers and asynchronous offloading

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.

```
void example() {
    float tmp[N], data_in[N], float data_out[N];
#pragma omp target data map(alloc:tmp[:N]) \
                        map(to:a[:N],b[:N]) \
                        map(tofrom:c[:N])
    {
        zeros(tmp, N);
        compute_kernel_1(tmp, a, N); // uses target
        saxpy(2.0f, tmp, b, N);
        compute_kernel_2(tmp, b, N); // uses target
        saxpy(2.0f, c, tmp, N);
}   }
```

```
void zeros(float* a, int n) {
#pragma omp target teams distribute parallel for
    for (int i = 0; i < n; i++)
        a[i] = 0.0f;
}
```

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

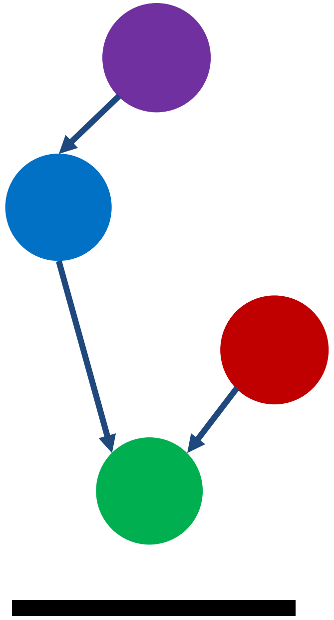
Example: target data and target update

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

    update_input_array_on_the_host(input);

#pragma omp target update device(0) to(input[:N])

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

Asynchronous Offloads

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

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

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

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

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

#pragma omp taskwait
```



Hybrid OpenMP and HIP (or CUDA)

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:

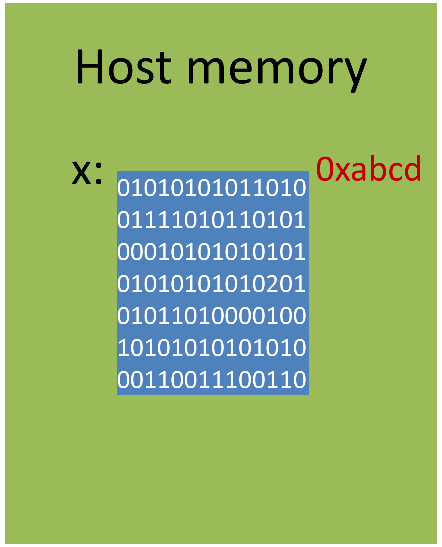
```
__global__ void saxpy_kernel(size_t n, float a, float * x, float * y) {
    size_t i = threadIdx.x + blockIdx.x * blockDim.x;
    y[i] = a * x[i] + y[i];
}

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!

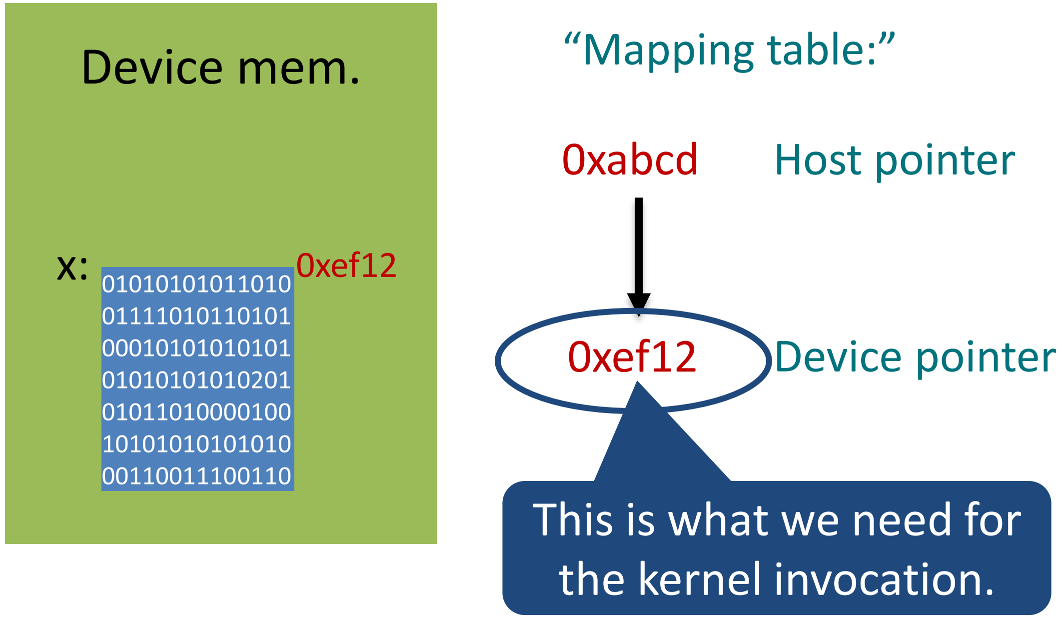
- We need a way to translate the host pointer that was mapped by OpenMP directives and retrieve the associated device pointer.

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



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 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_addr(x[:0],y[:0])
        {
            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

Detachable

- 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 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_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);
  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

Performance Optimisation and Productivity 3

A Centre of Excellence in HPC

Contact:

https://www.pop-coe.eu

pop@bsc.es

@POP_HPC

youtube.com/POPHPC

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101143931. The JU receives support from the European Union's Horizon Europe research and innovation programme and Spain, Germany, France, Portugal and the Czech Republic.