INAF
ISTITUTO NAZIONALE
DI ASTROFISICA

GPU Programming

Lecture 3
UniTS Advance HPC Course 2024/2025

Agenda

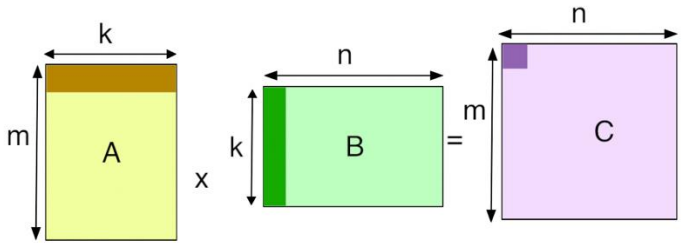
- OpenMP for GPUs
- Async Programming
- TASK workload
- Signal callback
- Summary

Multi-level Parallelism with OpenMP

```
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];
    }
}
```

Parallelism on the Target Device

DGEMM $C = \alpha A \times B$




```
void __attribute__ ((noinline)) mm_mul(TYPE *MA, TYPE *MB, TYPE *MC, TYPE *MCPU, TYPE alpha, int Ndim, int Mdim, int Kdim){
    for (int i = 0; i < Mdim; i++) {
      for (int j = 0; j < Ndim; j++) {
        for (int kk = 0; kk < Kdim; kk++) {
          MC[i * Ndim + j] =  MA [i * Kdim + kk] * MB [kk * Ndim + j] + MC[i * Ndim + j];
        }
      MC[i * Ndim + j] *= alpha ;
    }
  }
```

OpenMP: Modular Programming

If you have multiple accelerators available, you can select the one on which you run the kernels with the `device` clause of the `target` construct. It includes both target data constructs and `target teams/parallel` constructs.

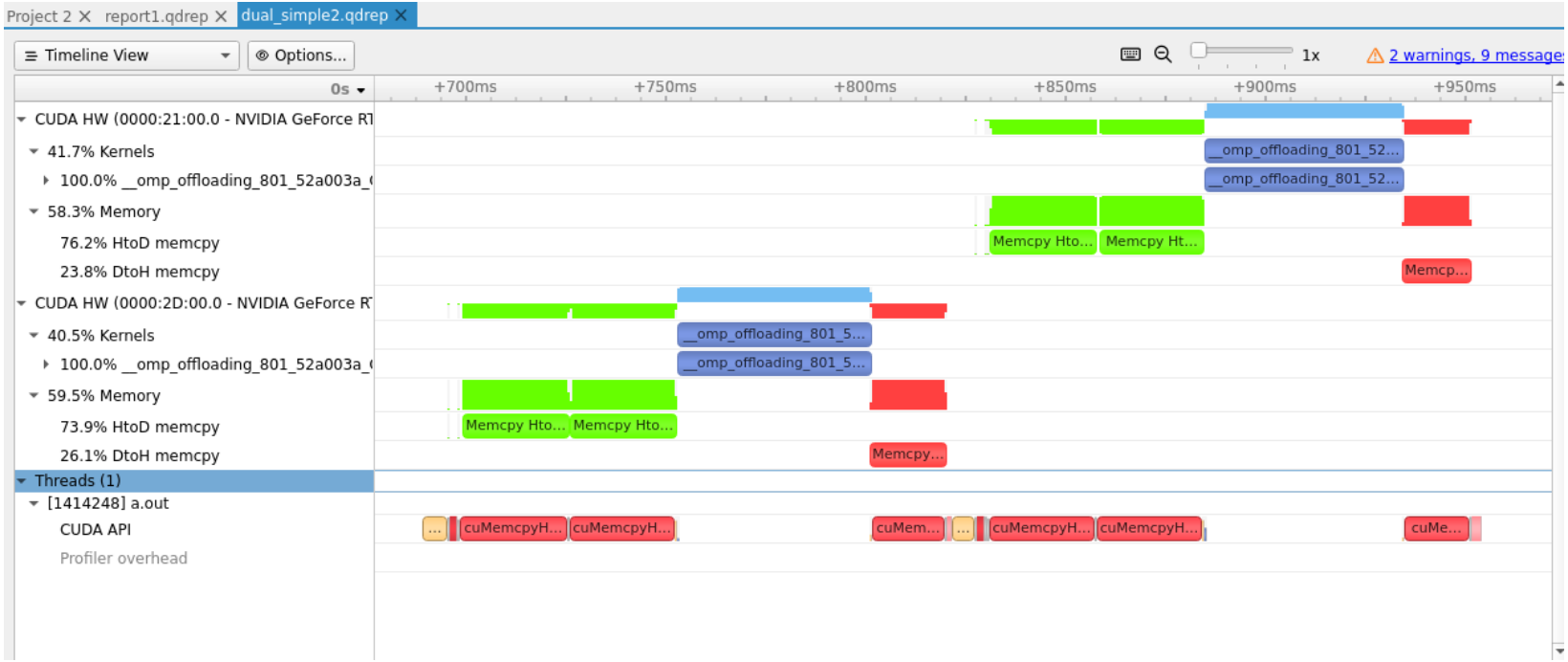
```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int num_gpus = omp_get_num_devices();
int my_gpu = my_rank%num_gpus
#pragma omp target data map(...) device(my_gpu)
{
    ...
}
```

`device` number is 0, 1, 2, …
On `target data` construct you can partition data between GPUs

OpenMP: Modular Programming

```
#pragma omp target data  device(1) map(to:a, n, x[0:n/2]) map(tofrom:y[0:n/2])
#pragma omp target  device(1)
{
printf("Running on device: %d\n", omp_get_device_num());
}
#pragma omp target teams distribute parallel for simd device(1)  map(to:a, n, x[0:n/2]) map(tofrom:y[0:n/2])
for (int i = 0; i < n/2; ++i) {
  y[i] = a * x[i] + y[i];
}

printf("second GPU\n");

#pragma omp target data  device(0) map(to:a, n, x[n/2+1:n/2]) map(tofrom:y[n/2+1:n/2])
{
#pragma omp target teams distribute parallel for simd  device(0)  map(to:a, n, x[n/2+1:n/2]) map(tofrom:y[n/2+1:n/2])
for (int i = n/2+1; i < n; ++i) {
  y[i] = a * x[i] + y[i];
}
}
```

OpenMP: Modular Programming



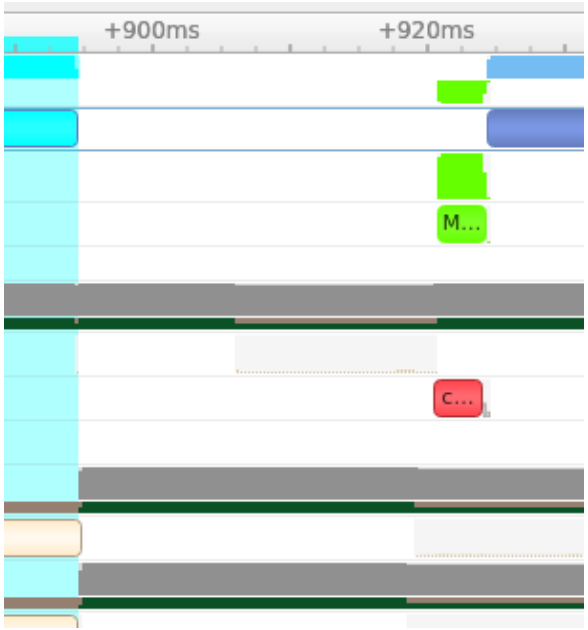
OpenMP: Async Computing

GPU and CPU are running the code serially

Can I process in parallel GPU and CPU code?

Can I process GPUs workload in parallel?

Can I disentangle data movement and computation?

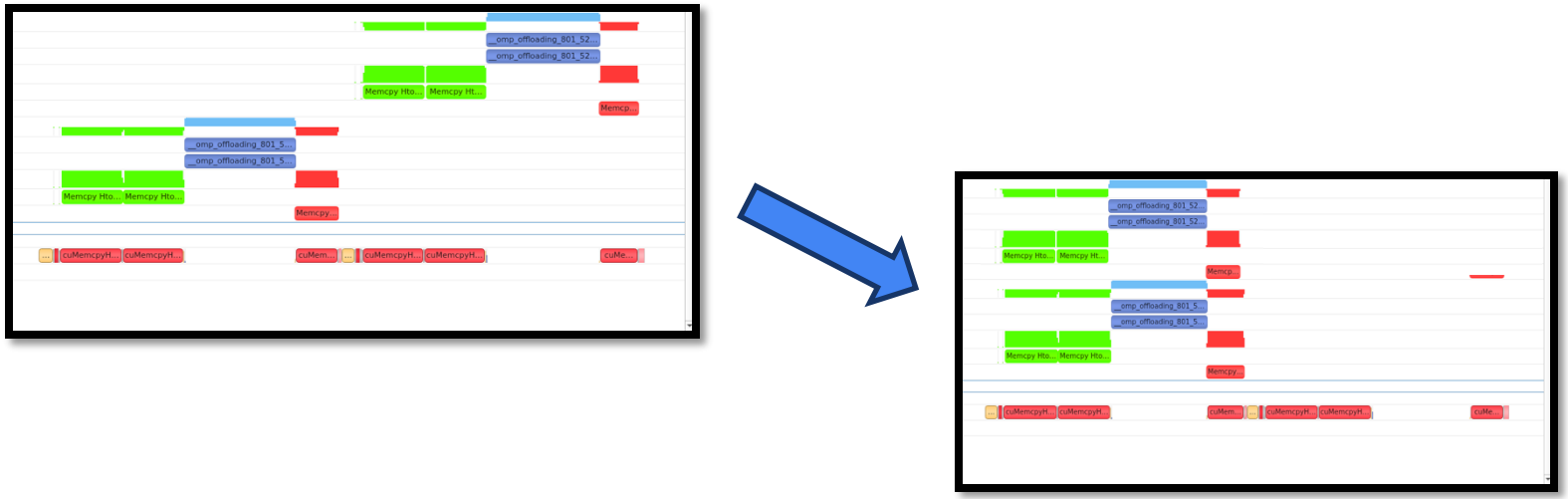


OpenMP: asynchronous operations

Using the CPU and GPU resources simultaneously

Couple data transfer with computing

Using multiple GPUs simultaneously



OpenMP: Concurrency & Streams

Concurrency: the ability to perform multiple CUDA operations simultaneously
(beyond multi-threaded parallelism)

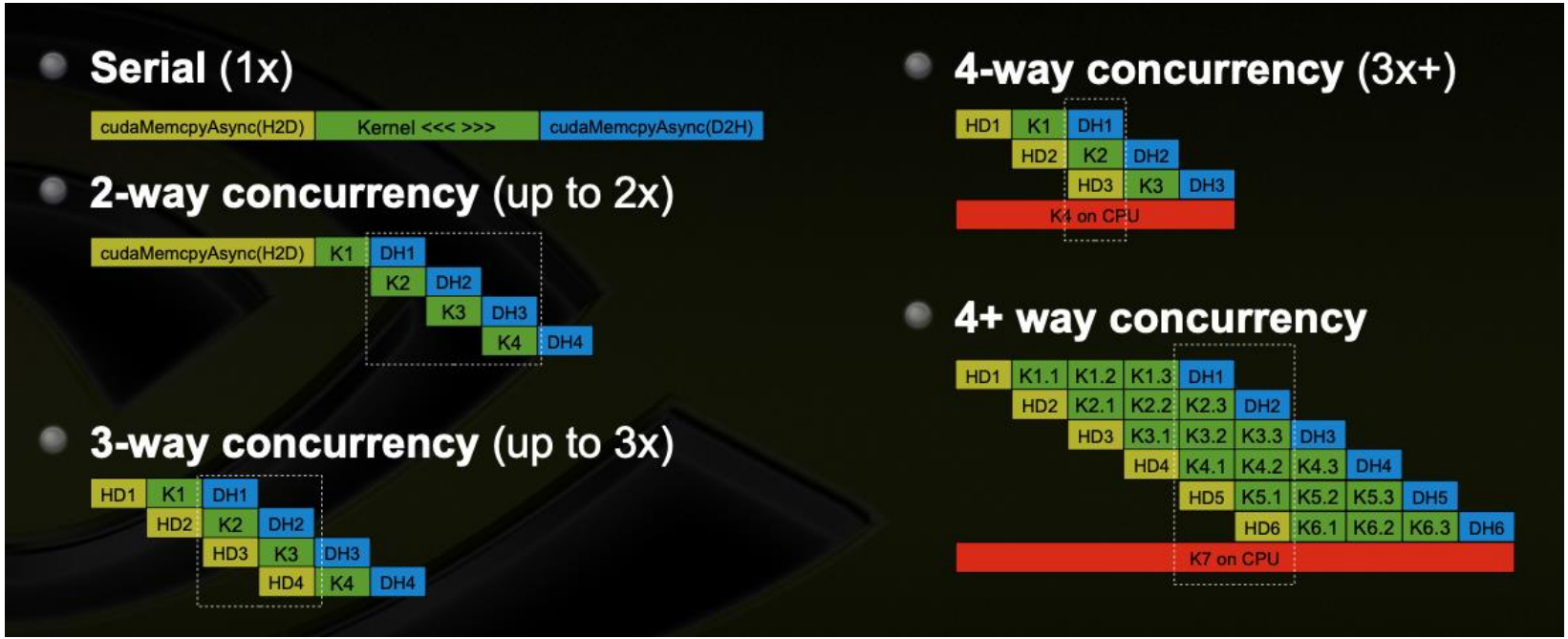
- CUDA Kernel <<<>>>
- `cudaMemcpyAsync` (HostToDevice)
- `cudaMemcpyAsync` (DeviceToHost)
- Operations on the CPU

Stream: a sequence of operations that execute in issue-order on the GPU

Programming model used to effect concurrency

- CUDA operations in different streams may run concurrently
- CUDA operations from different streams may be interleaved

OpenMP: Concurrency



GPU Streams primer

Default Stream (aka Stream 0)

- Stream used when no stream is specified
- Completely **synchronous** w.r.t. host and device
  - As if `cudaDeviceSynchronize()` inserted before and after every CUDA operation

- Exceptions – asynchronous w.r.t. host
  - Kernel launches in the default stream
  - cudaMemcpy*Async
  - cudaMemset*Async
  - cudaMemcpy within the same device
  - H2D cudaMemcpy of 64kB or less

Requirements for Concurrency

Default Stream (aka Stream 0)

- CUDA operations must be in different, non-0, streams
- `cudaMemcpyAsync` with host from 'pinned' memory
  - Page-locked memory
  - Allocated using cudaMallocHost() or cudaHostAlloc()
- Sufficient resources must be available
  - cudaMemcpyAsyncs in different directions
  - Device resources (SMEM, registers, blocks, etc.)

Concurrency with CUDA

….is relatively simple!

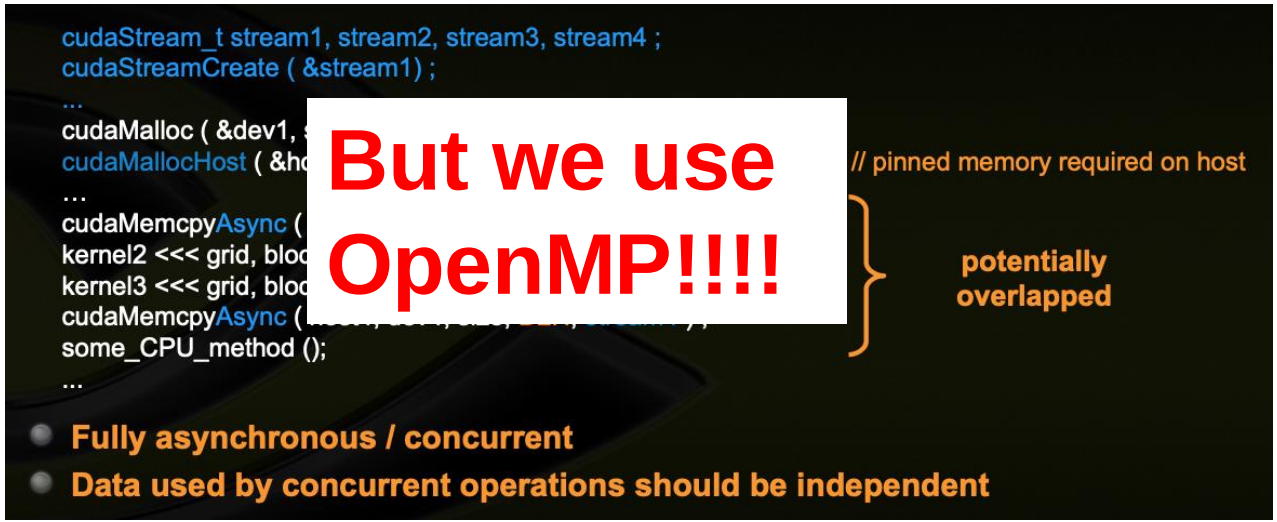
```
cudaMalloc ( &dev1, size ) ;
double* host1 = (double*) malloc ( &host1, size ) ;
…

cudaMemcpy ( dev1, host1, size, H2D ) ;
kernel2 <<< grid, block >>> ( …, dev2, … ) ;
some_CPU_method ();
kernel3 <<< grid, block >>> ( …, dev3, … ) ;
cudaMemcpy ( host4, dev4, size, D2H ) ;
...
```

potentially overlapped

- GPU kernels are asynchronous with host by default

Concurrency with CUDA

….is relatively simple!



But we use OpenMP!!!!

Concurrency with OpenMP

SAXPY example: dual GPU

Two Threads from HOST
Threads execute in parallel
Data partitioned between GPUs

```
// Use OpenMP to launch computations on two GPUs concurrently
#pragma omp parallel num_threads(2)
{
    int thread_id = omp_get_thread_num(); // 0 or 1
    int device = thread_id; // Assign GPU device based on thread ID

    // Calculate offsets for each GPU
    const int offset = thread_id * nhalf;
    const float *x_local = x + offset;
    float *y_local = y + offset;

    // Each GPU works on its portion of the data
    computeLoop(nhalf, a, x_local, y_local, device);
}
```

```
int computeLoop(const int n,
                const float a,
                const float *x,
                      float *y,
                const int device)
{
    #pragma omp target data device(device) map(to:a, n, x[0:n]) map(tofrom:y[0:n])
    {
        #pragma omp target teams distribute parallel for simd device(device)
        for (int i = 0; i < n; ++i) {
            y[i] = a * x[i] + y[i];
        }
    }
    return device;
```

Concurrency with OpenMP

SAXPY example: dual GPU

```
#pragma omp declare target
int computeLoop(const int n,
                const float a,
                const float *x,
                      float *y,
                const int device)
{
    #pragma omp target data device(device) map(to:a, n, x[0:n]) map(tofrom:y[0:n])
    {
        #pragma omp target teams distribute parallel for simd device(device)
        for (int i = 0; i < n; ++i) {
            y[i] = a * x[i] + y[i];
        }
    }
    return device;
}
```

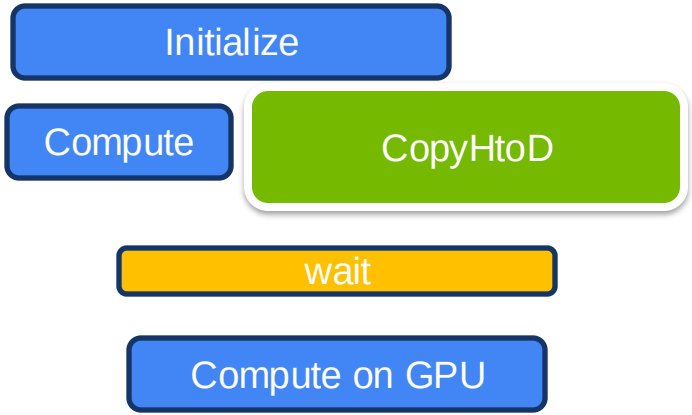
Synchronization

```
void GPUsaxpy_tasks(const int n,
                    const float a,
                    const float *x,
                          float *y)
{
    int num_devices = omp_get_num_devices();
    if (num_devices < 2) {
        printf("Error: At least two devices are required.\n");
        exit(EXIT_FAILURE);
    }

    int nhalf = n / 2; // Split workload into two halves

    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int device = 0; device < 2; ++device) {
                int offset = device * nhalf;
                const float *x_local = x + offset;
                float *y_local = y + offset;

                // Create a task for each GPU
                #pragma omp task shared(a, x_local, y_local, nhalf) firstprivate(device) nowait
                {
                    computeLoop(nhalf, a, x_local, y_local, device);
                }
            }
        } // End of single region

        // Wait for all tasks to complete
        #pragma omp taskwait
    } // End of parallel region
```

OpenMP: asynchronous copy

Overlap computing and data transfer

```
#pragma omp parallel for
for (int iw …)
{
   int* array = all_arrays[iw].data();
   #pragma omp target \
     map(always, tofrom: array[:100])
   for(int i ...)
   { // operations on array }
}
```



Concurrency with OpenMP

SAXPY example: dual GPU

Two Threads from HOST
Threads execute in parallel
Data partitioned between GPUs

```
// Use OpenMP to launch computations on two GPUs concurrently
#pragma omp parallel num_threads(2)
{
    int thread_id = omp_get_thread_num(); // 0 or 1
    int device = thread_id; // Assign GPU device based on thread ID

    // Calculate offsets for each GPU
    const int offset = thread_id * nhalf;
    const float *x_local = x + offset;
    float *y_local = y + offset;

    // Each GPU works on its portion of the data
    computeLoop(nhalf, a, x_local, y_local, device);
}
```

```
int computeLoop(const int n,
                const float a,
                const float *x,
                      float *y,
                const int device)
{
    #pragma omp target data device(device) map(to:a, n, x[0:n]) map(tofrom:y[0:n])
    {
        #pragma omp target teams distribute parallel for simd device(device)
        for (int i = 0; i < n; ++i) {
            y[i] = a * x[i] + y[i];
        }
    }
    return device;
```

OpenMP: asynchronous tasking

- Ideal scenario. Only need the master thread to have full asynchronous kernel execution.
- LLVM 12 uses helper threads.

LIBOMP_USE_HIDDEN_HELPER_TASK=TRUE

LIBOMP_NUM_HIDDEN_HELPER_THREADS=8

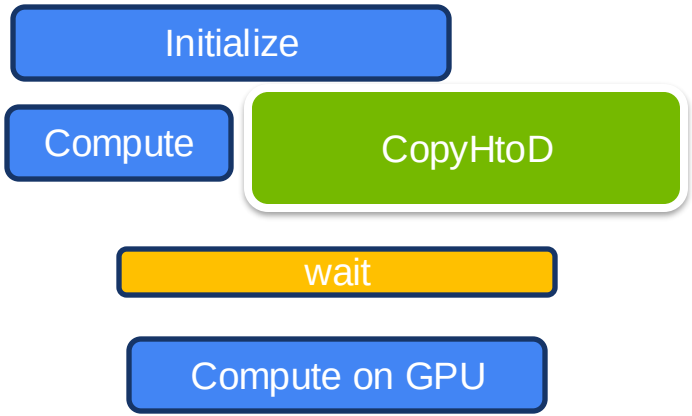
- Pros:
  - No need of parallel region
  - Fast turnaround
- Cons:
  - Helper threads are actively waiting
  - They can be “noisy”

```
for (int iw …) {
   int* array = all_arrays[iw].data();
   // target task
   #pragma omp target nowait \
      map(always, tofrom: array[:100])
   for(int i ...) { // operations on array }
}
#pragma omp taskwait
```

OpenMP: asynchronous copy

```
int main() {
    float A[M * K], B[N * K], C[N * M]

    // Initialize arrays A and B
    for (int i = 0; i < M * K; ++i) {
        A[i] = i * 0.1f;
    }
    for (int i = 0; i < N * K; ++i) {
        B[i] = i * 0.2f;
    }

    // Asynchronous data transfer to the device
    #pragma omp target nowait enter data map(to:A[0:M*K], B[0:N*K])
    // Initialize arrays C
    for (int i = 0; i < N * M; ++i) {
        B[i] = i * 0.2f;
    }
    // Ensure data transfer is complete before starting computation
    #pragma omp taskwait

    // Start kernel execution
    #pragma omp target data map(fro   '0:M*K])
    {
```

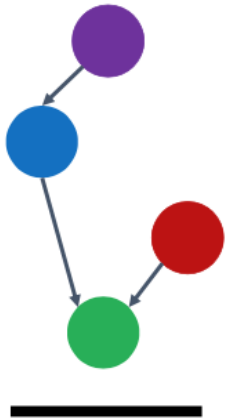
Synchronization



OpenMP: asynchronous operations

- 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
```



OpenMP: asynchronous operations

```
// Asynchronously transfer data to the device
#pragma omp target nowait enter data map(to:A[0:M*K], B[0:K*N], C[0:M*N])

// Perform other tasks on the host while data is being transferred
printf("Host is performing initialization while data transfer occurs...\n");

// Wait for the data transfer to complete before starting computation
#pragma omp taskwait

// Task 1: Matrix multiplication
#pragma omp target map(to:A[0:M*K], B[0:K*N]) map(from:C[0:M*N]) \
    nowait depend(out:C) depend(in:A) depend(in:B)
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        C[i * N + j] = 0.0;
        for (int k = 0; k < K; k++) {
            C[i * N + j] += A[i * K + k] * B[k * N + j];
        }
        C[i * N + j] *= alpha;
    }
}
// Task 2: Another operation depending on the result of Task 1
#pragma omp target map(to:C[0:M*N]) map(from:B[0:K*N]) \
    nowait depend(in:C) depend(out:B)
for (int i = 0; i < K; i++) {
    for (int j = 0; j < N; j++) {
        B[i * N + j] += C[i * N + j % M]; // Some dependent operation
    }
}

// Wait for all tasks to complete
#pragma omp taskwait
```

Hybrid programming…let's mix everything together

- 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

Hybrid programming…let's mix everything together

```
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);
    }
}
```

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

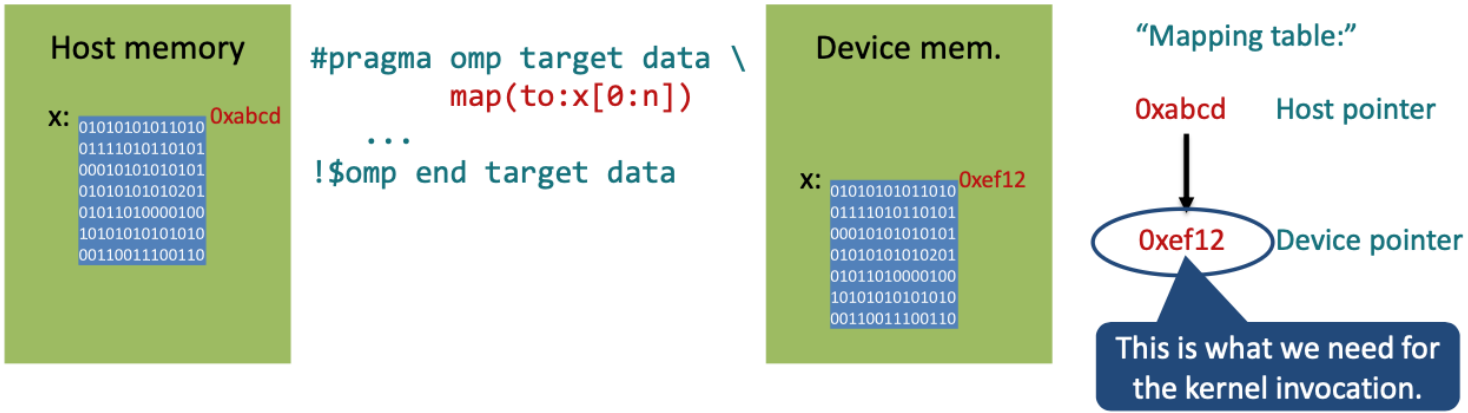
```
void saxpy(size_t n, float a,
           float * x, float * y) {
#pragma omp target teams distribute \
                    parallel for simd
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Hybrid programming…let's mix everything together

```
#include <cuda_runtime.h>
#include <iostream


__global__ void addVectors(float *A, float *B, float *C, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        C[idx] = A[idx] + B[idx];
    }
}
// Copy vectors to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

addVectors<<<numBlocks, blockSize>>>(d_A, d_B, d_C, n);

// Copy result back to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

Hybrid programming…pointer translation

- 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.



Hybrid programming…pointer translation

- 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.

Hybrid programming…pointer translation

```
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])
        {
```

Advanced Async Computing

- 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)
```

Advanced Async Computing

- 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)`

Advanced Async Computing

```
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

Advanced Async Computing: callback functions

```
void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
 ③omp_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 taskwait②④
#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

Advanced Async Computing: do I need task wait?

```
void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
  ②omp_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)         // 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

INAF
ISTITUTO NAZIONALE
DI ASTROFISICA

THANK YOU.

Now, it’s your time
QUESTIONS?