OPENMP USERS MONTHLY TELECON

Argonne NATIONAL LABORATORY

LEVERAGING IMPLICIT CUDA STREAMS AND ASYNCHRONOUS OPENMP OFFLOAD FEATURES IN LLVM

YE LUO
Computational Science Division,
Leadership Computing Facility
Argonne National Laboratory

March 26th

OUTLINE

- Using implicit CUDA streams
  - Maximize asynchronous operations on 1 stream
  - Using concurrent offload regions from multiple threads
  - Overlap computation with data transfer for free

- Asynchronous tasking
  - Using helper threads (LLVM12)
  - Leverage additional OpenMP threads
  - Coordinate tasks with dependency

USING IMPLICIT CUDA STREAMS

EXPLICIT VS IMPLICIT

Pros and cons

EXPLICIT

- Close to metal programming
  - Calls to native runtime directly
  - Full control but lengthy code
- Asynchronous execution and synchronization managed by the developer. In-order queue/streams and events.

IMPLICIT

- Programming OpenMP
  - The OpenMP runtime handles calls to the native runtime
  - Less control but more portable
  - Performance depends
- Asynchronous tasking with dependency handled by the OpenMP runtime

OPENMP OFFLOAD RUNTIME MOTIONS

Poor performance if every step is synchronous

- Allocate array on device (very slow)
- Transfer H2D (slow)
- Launch the kernel
- Transfer D2H (slow)
- Deallocate array on device (very slow)

```
// simple case
#pragma omp target \
   map(array[:100])
for(int i ...)
{ // operations on array }
```

PRE-ARRANGE MEMORY ALLOCATION

Move beyond textbook example

- Accelerator memory resource allocation/deallocation is orders of magnitude slower than that on the host.
- These operations may also block asynchronous execution.
- Allocate array on host pinned memory to make the transfer asynchronous.

https://github.com/ye-luo/miniqmc/blob/OMP_offload/src/Platforms/OMPTarget/OMPallocator.hpp

```
// optimized case
// pre-arrange allocation
#pragma omp target enter data \
  map(alloc: array[:100])
…
// use always to enforce transfer
#pragma omp target map(always, array[:100])
for(int i ...) { // operations on array }

// free memory
#pragma omp target exit data \
  map(delete: array[:100])
```

IMPLICIT ASYNCHRONOUS DISPATCH

Using queues/streams

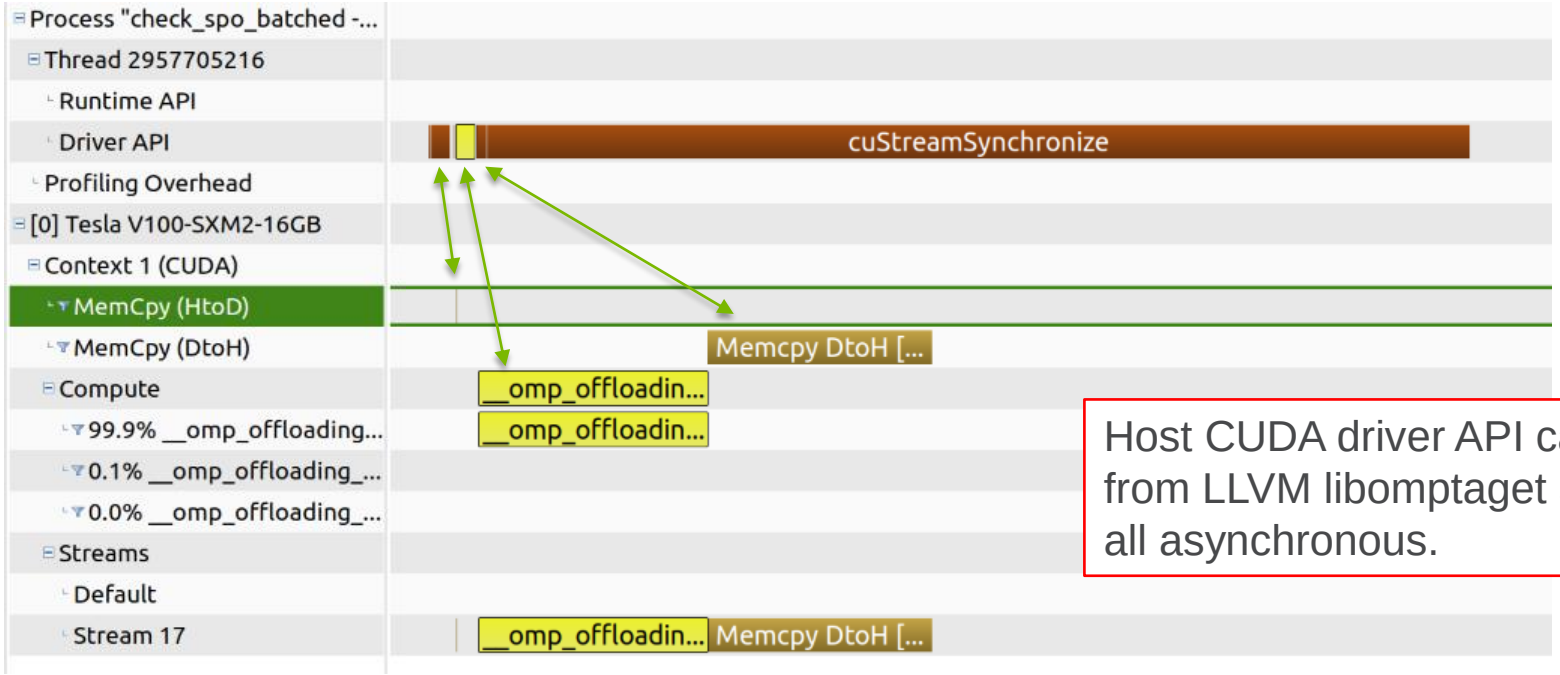
- NVIDIA CUDA supports streams for asynchronous computing
- IBM XL and LLVM Clang OpenMP runtime enqueue non-blocking H2D, kernel, D2H with only one synchronization in the end.
  - Async transfer H2D
  - Async enqueue the kernel
  - Async transfer D2H
  - Synchronization if ‘nowait’ is not used

```
// optimized case
// pre-arrange allocation
#pragma omp target enter data \
  map(alloc: array[:100])
…
// use always to enforce transfer
#pragma omp target map(always, array[:100])
for(int i ...) { // operations on array }

// free memory
#pragma omp target exit data \
  map(delete: array[:100])
```

IMPLICIT ASYNCHRONOUS DISPATCH (CONT)

Maximize asynchronous calls within one target region



Host CUDA driver API calls from LLVM libomptaget are all asynchronous.

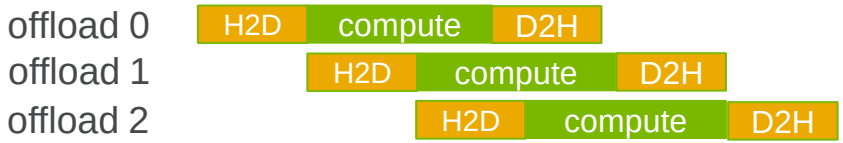
CONCURRENT EXECUTION AND TRANSFER

Overlapping computation and data transfer

- Need multiple concurrent target regions
- IBM XL and LLVM Clang OpenMP runtime select independent CUDA streams for each offload region.
- Kernel execution from one target region may overlap with kernel execution or data transfer from another target region

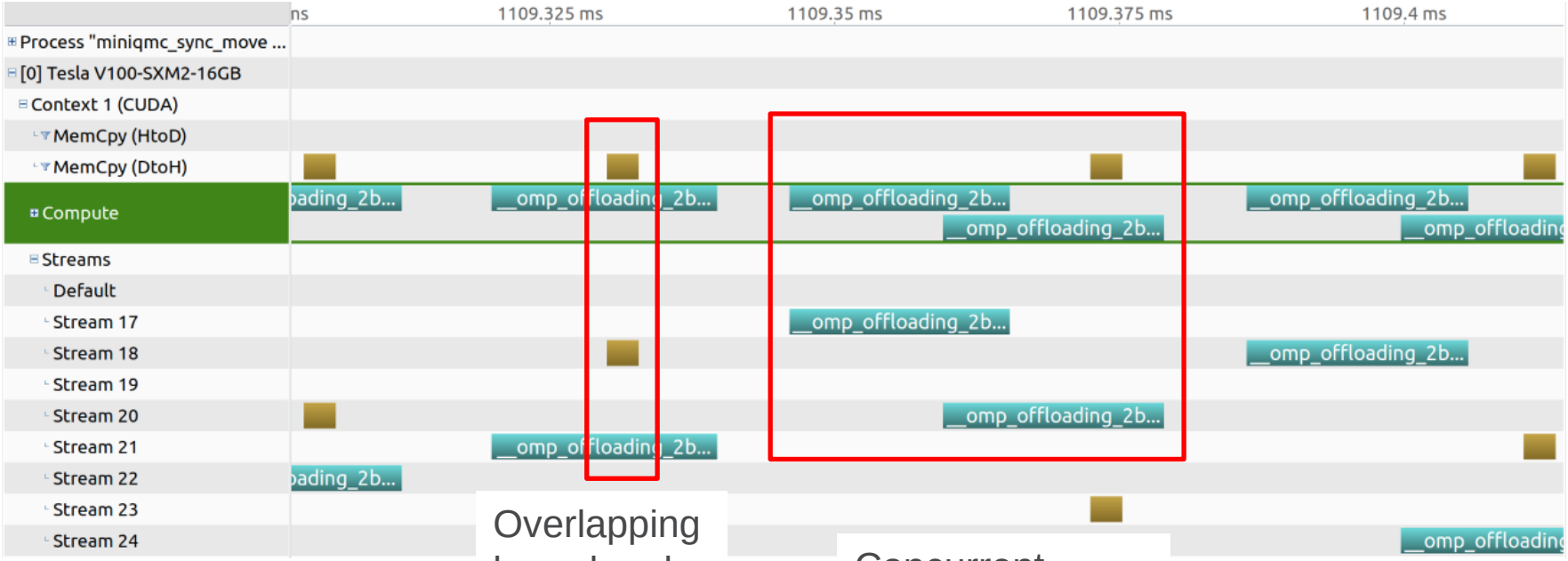
https://github.com/ye-luo/openmp-target/blob/master/hands-on/gemv/6-gemv-omp-target-many-matrices-no-hierachy/gemv-omp-target-many-matrices-no-hierachy.cpp

```
#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 }
}
```



CONCURRENT EXECUTION (CONT)

From multiple OpenMP threads, in miniQMC



ASYNCHRONOUS TASKING

ASYNCHRONOUS TASKING

Ideal case

- 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
```

ASYNCHRONOUS TASKING

When threads are available to process CPU tasks

- No need of helper threads.
- All the threads used for task dispatching are regular OpenMP threads with all the usual affinity control applied.
- Works better with CPU tasks on going as well.

```
#pragma omp parallel // start workers for tasks
{
   #pragma omp single
   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 }
   } // implicit barrier to wait for all tasks
}
```

ASYNCHRONOUS TASKING

When threads are available to process CPU tasks

- Reserve more threads than “for” loop iterations
- Target task goes to idle threads.
- Each iw iteration remains independent

```
OMP_NUM_THREADS=16
n=8
#pragma omp parallel for
for (int iw = 0; iw<n; iw++) {
    int* array = all_arrays[iw].data();
    // target task
    #pragma omp target nowait \
        map(always, tofrom: array[:100])
    for(int i ...) { // operations on array }
    //something else to do on the current thread
}
```

ASYNCHRONOUS TASKING DEPENDENCY

Coordinating host and offload computation

- Not all the features are worth the effort porting to accelerators
- Using tasking to leverage idle host resource for non-blocking host computation as Ncores>>Ngpu
- * performance heavily depends on compiler runtime implementation.

https://github.com/ye-luo/openmp-target/blob/master/hands-on/tests/target_task/target_nowait_task.cpp

```
#pragma omp target map(from: a) \
  depend(out: a) nowait
{
    int sum = 0;
    for (int i = 0; i < 100000; i++)
      sum++;
    a = sum;
}
#pragma omp task
{some independent work on CPU}
#pragma omp task depend(in: a) shared(a)
{ // some postprocessing on CPU}

#pragma omp taskwait
```