OpenMP offload optimization guide: beyond kernels - Lessons learned in QMCPACK
Ye Luo, Argonne National Laboratory
ACKNOWLEDGEMENTS

- Thanks to
  - ECP QMCPACK team
  - ECP SOLLVE team

Supported by:
This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy’s Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation’s exascale computing imperative.
QMCPACK

A real-world production application

- QMCPACK, is a modern high-performance open-source Quantum Monte Carlo (QMC) simulation code. QMCPACK is written in C++ and designed with the modularity afforded by object-oriented programming. It makes extensive use of template metaprogramming to achieve high computational efficiency. 
  https://qmcpack.org/

- It is a real-world production application. All the examples in this presentation reflects OpenMP use patterns in QMCPACK. 
  https://github.com/QMCPACK/qmcpack

- Most OpenMP explorations are done via miniQMC, a miniapp. 
  https://github.com/QMCPACK/miniqmc
ACCELERATOR PORTING MYTH

Why my code runs even slower?

- The repeated story very similar to other GPU porting stories not just OpenMP.
  - Profiler shows GPU activity is low.
  - Host time and overall wall clock time go up.
- OpenMP is limited by the characteristics of accelerator in use.
- The goal is to minimize OpenMP overhead on top of vendor native programing model.
  - Within a kernel. Ensure source code efficient transformation into kernels.
  - Beyond kernel. Ensure OpenMP abstraction with minimal overhead.
    - Adopt best practice in user code.
    - Good OpenMP compiler/runtime implementation.
PORTABILITY AT THE SOURCE CODE
Choosing OpenMP programing style carefully

OPENMP DIRECTIVE

```c
#pragma omp target enter data
map(alloc: a[:100])
```

- Prons:
  - No side effect when turned off
  - Fall back to host for debugging
    `OMP_TARGET_OFFLOAD=disabled`

- Cons:
  - Less verbose

OPENMP API

```c
int * a_dev = omp_target_alloc(omp_get_default_device(), 100);
```

- Prons:
  - Explicit device control

- Cons:
  - Need `#ifdef _OPENMP`
  - Complicated fallback logic
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.

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

// 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 }
HIDE MEMORY_ALLOCATION IN C++

Create customized allocator

- Used in container classes like std::vector
- HostAllocator can be further customized to satisfy
  - Alignment
  - Registration in the accelerator memory space for maximal transfer performance, for example cudaHostRegister.

```cpp
template<typename T, class HostAllocator = std::allocator<T>>
struct OMPAllocator : public HostAllocator
{
    value_type* allocate(std::size_t n)
    {
        value_type* pt = HostAllocator::allocate(n);
        #pragma omp target enter data map(alloc:pt[0:n])
        return pt;
    }
    void deallocate(value_type* pt, std::size_t n)
    {
        #pragma omp target exit data map(delete:pt[0:n])
        HostAllocator::deallocate(pt, n);
    }
};
```
AVOID UNNECESSARY MAP
firstprivate scalars don’t need mapping

- Compilers implement explicit mapping as allocating the memory for each scalar and transferring data.
- Since OpenMP 4.5, scalars are firstprivate by default. Compilers pack them as kernel arguments and no allocation and explicit transfers involved.

```c
// simple case
int a, b, c;
#pragma omp target map(to: a, b, c)
{ // use a, b, c parameters }

// optimized case, no need of map
int a, b, c;
#pragma omp target
{ // use a, b, c parameters }
```
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.

Other vendors support similar features.

```
// simple case
#pragma omp target 
  map(always, tofrom: array[:100])
for(int i ...)
{ // operations on array }
```
**IMPLICIT ASYNCHRONOUS DISPATCH (CONT)**

Maximize asynchronous calls within one target region

<table>
<thead>
<tr>
<th>Process &quot;check_spo_batched -...&quot;</th>
<th>Thread 2957705216</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime API</td>
<td></td>
</tr>
<tr>
<td>Driver API</td>
<td></td>
</tr>
<tr>
<td>Profiling Overhead</td>
<td></td>
</tr>
<tr>
<td>[0] Tesla V100-SXM2-16GB</td>
<td></td>
</tr>
<tr>
<td>Context 1 (CUDA)</td>
<td></td>
</tr>
<tr>
<td>MemCpy (HtoD)</td>
<td></td>
</tr>
<tr>
<td>MemCpy (DtoH)</td>
<td></td>
</tr>
<tr>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>99.9% __omp_offloading...</td>
<td></td>
</tr>
<tr>
<td>0.1% __omp_offloading...</td>
<td></td>
</tr>
<tr>
<td>0.0% __omp_offloading...</td>
<td></td>
</tr>
<tr>
<td>Streams</td>
<td></td>
</tr>
<tr>
<td>Default</td>
<td></td>
</tr>
<tr>
<td>Stream 17</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram showing CUDA driver API calls]

Host CUDA driver API calls from LLVM libomptaget are all asynchronous.
CONCURRENT EXECUTION AND TRANSFER

Overlapping computation and data transfer

- IBM XL and LLVM Clang
  OpenMP runtime select
  independent CUDA streams for
  each offload region.

- Target region executed by
  different threads happens
  concurrently.

- Kernel execution from one target
  region may overlap with kernel
  execution or data transfer
  dispatched by another thread

```c
#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
    }
} // operations on array
```
CONCURRENT EXECUTION (CONT)

From multiple OpenMP threads, in miniQMC
ASYNCHRONOUS TASKING

Coordinating host and offload computation

- Providing a feature complete application enables smoothing user experience when gradually enabling acceleration.
- Not all the features are worth the effort porting to accelerators.
- Using tasking to leverage idle host resource for non-blocking host computation.
- Performance heavily depends on compiler runtime implementation.

```c
#pragma omp parallel for
for (int iw ...) {
    int* array = all_arrays[iw].data();
    // offload task
    #pragma omp target nowait depend(out:a) \
        map(always, tofrom: array[:100])
    for(int i ...) { // operations on array }
    // host task 1
    #pragma omp task
    { // operations on array }
    // host task 2 depend on the offload task
    #pragma omp task depend(in:a)
    { // operations on array }
    #pragma omp taskwait
}
OPENMP 5.0 AND BEYOND

Many promising features

- Meta directives and declare variant functions
  - Help better organized source code and less duplication

- Detached task
  - For composability with other asynchronous runtime.

- Interop object
  - For exposing vendor native queue/streams

- OMPT, OMPD support for profiling and debugging tools.
DETACHED TASK

For better composability

- Desired code example asynchronously calling cuBLAS without explicit waiting.
- Listed 5.0 feature.
- Waiting for actual compiler implementation.

```cpp
#omp task detach(cuda_event1) depend(out:p)
{
    cublas::gemm // first call
    cudaStreamAddCallback(stream, callback, cuda_event1, 0);
}
#omp target nowait depend(inout:p)
{
    // applyW_batched body
}
#omp task detach(cuda_event2) depend(in:p)
{
    cublas::gemm // second call
    cudaStreamAddCallback(stream, callback, cuda_event2, 0);
}
#omp taskwait
```
ESSENTIAL FEATURES FOR APPLICATIONS
Struggled in 2019. A lot of exciting improvements in 2020


<table>
<thead>
<tr>
<th>Compiler</th>
<th>Clang 11</th>
<th>AOMP 11.8-0</th>
<th>XL 16.1.1-5</th>
<th>OneAPI beta08</th>
<th>Cray 9.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>device</td>
<td>NVIDIA</td>
<td>AMD</td>
<td>NVIDIA</td>
<td>Intel</td>
<td>NVIDIA</td>
</tr>
<tr>
<td>math functions</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td>complex arithmetics</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
<td>Fail</td>
</tr>
<tr>
<td>declare target static data</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td>static linking</td>
<td>Fail</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td>multiple streams</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
<td>Functioning</td>
<td>Functioning</td>
</tr>
<tr>
<td>check_spo</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td>check_spo_batched</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td>miniqmc_sync_move</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
<td>Pass</td>
</tr>
</tbody>
</table>

Workaround in CMake
SUMMARY

- Application developers need to pay attention to application performance beyond kernels.
- Many simple patterns may be adopted to have significant performance gain.
- OpenMP offload runtime overhead and be minimized to negligible.
- Task level parallelism becomes essential for accelerators.
- Improved compilers and OpenMP runtimes in 2020 enable production use of OpenMP.
openmp.org

OpenMP API specs, forum, reference guides, and more

link.openmp.org/sc20

Videos and PDFs of OpenMP SC’20 presentations