LESSONS LEARNED IN DESIGNING PERFORMANCE PORTABLE QMCPACK USING OPENMP OFFLOAD TO GPUs

YE LUO
Argonne National Laboratory

Jun 30th OpenMP Users Monthly Telecon
ACKNOWLEDGEMENT

Exascale Computing Project: application development

▪ Lead PI: Paul Kent

▪ 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.
OUTLINE

- QMCPACK redesign
- GPU and OpenMP porting tips
- QMCPACK on AMD and INTEL GPUs
ELECTRONIC STRUCTURE METHODS

QMC can be the new sweet spot

Time scale: picosecond = $10^{-12}$ seconds

Length scale: 10 nm = $10^{-8}$ meters

- **exact** accuracy (eV)
  - CI: $N^7$
  - QMC: $N^{3-4}$
  - CC: $N^7$

- 0.001-0.01
  - superconductivity, magnetism

- 0.05-0.1
  - cohesion, barriers

- 0.5+
  - chemical bonds

- not ab initio

- system size (electrons)
  - 50
  - 2,000
  - 10,000
  - 100,000

quantum chemistry

chemical bonds

cohesion, barriers

superconductivity, magnetism

accuracy (eV)
PETASCALE TO EXASCALE CHALLENGE

How large problem can we solve?

What is next?
1. Solve faster and more petascale problems
2. Solve much larger problems

TiO2 polymorphs
216 atoms with 1536 electrons, 10 meV/f.u.

Metal organic framework
153 atoms with 594 electrons, 10 meV total energy.

1k atoms
10k electrons
PRE AND EXASCALE SYSTEMS IN 2023

Need to make the code work on all machines

NERSC Perlmutter
Argonne Aurora
Argonne Polaris

Make QMCPACK run well everywhere including CPU-only systems
On-node performance is the focus
QMCPACK

- QMCPACK, is a modern high-performance open-source Quantum Monte Carlo (QMC) simulation code for electronic structure calculations of molecular, quasi-2D and solid-state systems.
- The code is C/C++ and adopts MPI+X (OpenMP/CUDA)
- Monte Carlo: massive Markov chains (walkers) evolving in parallel. 1\textsuperscript{st} level concurrency. Good for MPI and coarse level threads.
- Quantum: The computation in each walker can be heavy when solving many body systems (electrons). 2\textsuperscript{nd} level concurrency. Good for fine level threads and SIMD.
- Math libraries: BLAS/LAPACK, HDF5, FFTW
DIFFUSION MONTE CARLO SCHEMATICS

Old configurations  Random walking  Possible new configurations

Population

Walker

Load balancing communication

Random walking

Possible new configurations

New configurations

Walker

Load balancing communication

Possible new configurations

New configurations

Population
WALKER BASED PARALLELISM

Works extreme well on petascale supercomputers

- Weak scaling efficiency 99% on 2/3 Mira and 95% on almost full Titan.
- Weak scaling, fix work per node. Strong scaling, fix the total number of samples.
- Equilibration excluded.

![Graph showing weak and strong scaling efficiency](image-url)
A HIGH-PERFORMANCE DESIGN FOR HIERARCHICAL PARALLELISM
Algorithm 2: Pseudocode for the multi-threaded implementation.
1: for MC generation = 1 \cdots M \ do \quad \text{seq.}
2: #pragma omp parallel for
3: for walker = 1 \cdots N_w \ do \quad \text{para.}
4: \quad \text{for particle } k = 1 \cdots N \ do \quad \text{seq.}
5: \quad \cdots
6: \quad \text{end for } \{ \text{particle} \}
7: \quad \text{end for } \{ \text{walker} \}
8: \quad \text{end for } \{ \text{MC generation} \}

Threads are used for weak scaling.
CUDA-BASED GPU IMPLEMENTATION

Algorithm 3 Pseudocode for the CUDA-based implementation.

1: for MC generation = 1 · · · M do  seq.
2:    for particle k = 1 · · · N do  seq.
3:        Algorithm 1. Line 5,6,7,8,9 over all the \(N_w\) walkers
4:    end for{particle}
5:  local energy \(E_L = \hat{H}\Psi_T(R)/\Psi_T(R)\) over \(N_w\)
6:  reweight and branch walkers based on \(E_L - E_T\)
7:  update \(E_T\) and load balance via MPI.
8: end for{MC generation}
## TRACING ON NVIDIA GPUs

### Single thread + GPU idle gaps

<table>
<thead>
<tr>
<th>Timeline View</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>11s</strong>: 32.105ms, +782.11ms, +782.115ms, +782.12ms, +782.125ms, +782.13</td>
</tr>
</tbody>
</table>

- Processes (2)
  - [1192809] qm
- Threads (1)
  - [1192809] qm
- CUDA API
  - cudaMemcpy, one body r..., cudaMemcpy1,..., cudaMemcpy...
  - one body r...
- Profiler overview
- CUDA HW (NVIDIA)
  - [All Streams]
    - two body ratio grad...
    - Me...
    - Me...
    - two body ratio grad, PBC...
    - Me...

---

U.S. Department of Energy
Argonne National Laboratory
Argonne National Laboratory is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC.
UNIFY BOTH IMPLEMENTATIONS

By design

Number of electrons vs. Number of walkers for Legacy GPU driver and Batched driver.

Target problem size space. Performance required.

Performance portable Batched driver.
DESIGN GOALS

• CPU or GPU specialization used only at lower levels.
• Reimplement GPU kernels using OpenMP offload for portability. Walker batching is needed for good GPU performance.
• Leverage multi-threading using OpenMP. Preferably load balanced.
NEW DESIGN WITH CROWDS

Algorithm 4 Pseudocode for the batched DMC driver.

1: for MC generation = 1 · · · M do
2:     #pragma omp parallel for
3:     for crowd = 1 · · · C do
4:         for particle k = 1 · · · N do
5:             Algorithm 1. Line 5,6,7,8,9 over all walkers with
in this crowd
6:         end for{particle}
7:     local energy $E_L = \hat{H}_T(R)/\Psi_T(R)$ over this crowd
8:     reweight and branch walkers based on $E_L - E_T$
9:     update $E_T$ and load balance via MPI.
10:    end for{crowd} CG
11:   end for{MC generation}

- lock-step walkers within a crowd
- Independent crowds
- Decay to legacy implementations
OPENMP OFFLOAD GPU IMPLEMENTATION

A bit more software technology to handle GPUs

- Use portable OpenMP target feature
  - Portable on NVIDIA, AMD, Intel GPUs. Fallback on CPU as well.
  - Multiple compilers. GNU, Clang, AOMP, NVHPC, OneAPI

- Multiple crowds (CPU threads) to launch kernels to GPUs
  - Maximize GPU utilization. Overlapping compute and transfer by OpenMP.

- Specialized in CUDA/HIP to call NVIDIA/AMD accelerated libraries.
  - cuBLAS/cuSolver, hipBLAS/rocSolver, MKL
HOW LLVM MADE OPENMP FAST

LLVM LIBOMPTARGET implementation

- #pragma omp target map(always, tofrom: vec[:N])
  - Assuming vec[:N] mapped already
  - Turns into async H2D, Kernel Launch, async D2H, StreamSynchronize.

- A pool of CUDA streams services multiple threads.
  - Keep GPU busy
  - Overlapping kernel and transfers

- Opt-out atomic transfer behaviors.
  - Coordinate data shared by multiple tasks/threads ahead of time.
  - Environment variable LIBOMPTARGET_MAP_FORCE_ATOMIC=0
CROWDS ON OPENMP THREADS

Overlapping kernels and transfers.
MULTI-THREADED OFFLOAD

A few more tips

- Using pinned memory to keep CPU cores submitting work to GPUs.
  - Method 1. Pin host memory using vendor APIs like `cudaHostRegister`
  - Method 2. allocated pinned memory using vendor APIs like `cudaMallocHost`.
  - Method 3. Use OpenMP extension `llvm/omp_target_alloc_host`

- Avoid allocating/deallocating GPU memory on the fly
  - Allocating/deallocating operations are very slow
  - Serialization prevents concurrent execution.
ONE VS A FEW CROWDS

- 7 cores per GPU on OLCF Summit
- Fixed 7 crowds
- Small walker count, performance drops
- Large walker count, performance improves.
V100 VS A100

- GPU acceleration is significant
- The larger HBM helps throughput. Very similar to ML.
- A100 is almost 3X when running 256 atom problem.
- 16 GB is the bottleneck on V100
LESSONS LEARNED SO FAR

What is needed for a performance portable code

- Understand how to make CPU and GPU work efficiently
- Analyze the compute pattern and map the existing concurrency to proper parallelism given by the hardware
- OpenMP + vendor libraries strategy works
  - ~100 CUDA kernels down to ~10 CUDA kernels + ~10 offload regions
  - Very maintainable code with decent performance
MY OPINION ABOUT
GPU AND OPENMP PORTING
MANAGE MORE LEVEL OF PARALLELISM

- Put effort in make the algorithm/design work better for GPUs.
  - Minimize data movement needs.

- At least two level of parallelism
  - CPU has core+SIMD
  - GPU has SM+SIMT lanes
  - In QMCPACK we have CPU threads + two level inside GPU kernel
DATA MOVEMENT IS THE KEY OF PORTING

- Data locality is the top priority
  - Interconnect is slower than memory. We write performant MPI code
  - Memory is slower than cache. We implement cache friendly algorithms.
  - CPU-GPU bus is slower than GPU memory. We need to first worry about data transfer.

- Avoid any programming model which
  - Ignores the performance difference from host and GPU memory spaces.
  - Doesn’t provide explicit data movement control

- GPU 101 teaches kernel programming. It is not the key.

- Managing data movement requires understanding the whole algorithm and implementation. This needs domain expert knowledge.
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.

```c
// 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 }
```
CUDA LANGUAGE IS NOT THE BEST CHOICE

For scientists

- Must be protected under macro. Macro means more test variants needed.
- Doesn’t run on hosts without GPUs.
  - Running and debugging on host are lot smoother and should catch most user errors
  - Host tooling are richer. Address and thread sanitizers, code coverage.
- No reduction operation support by the language. SYCL does better.
- Restrict it to only serving library calls.
- Scientist needs to spend time on methods, algorithms. Leave the engineers to worry absolute performance.
- VASP, Quantum ESPRESSO, QMCPACK all reduced exposure to CUDA.
DO NOT USE OPENMP LIKE CUDA

Directives instead of kernels

const int N=100;
int array[N];
#pragma omp target \
  map(from, a[:N])
for(int i=0; i<N; i++)
array[i] = 1;

Do this

const int N=100;
int array[N];
int* array_d = (int*) omp_target_alloc(0, N*sizeof(int));
#pragma omp target is_ptr(array_d)
for(int i=0; i<N; i++)
array_d[i] = 1;
omp_target_memcpy(array, array_d, 0);

Don’t do this
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
AMD AND INTEL
AMD?
Summit vs Spock

- Runs reasonably well on MI100
- Not fully bug free but improves month by month
- Performance is decent.
  - HIP is close to legacy CUDA
  - OpenMP prevails in certain cases
- Expecting further software improvements.
Computing Quantum Mechanical Properties faster.

QMCPACK performance

- Intel Data Center GPU Max Series
- Nvidia H100

16 threads per card

Relative DMC samples/sec/card

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.
CONCLUSION

- The new design of hierarchical parallelism maximized performance on hardware. doi: 10.1109/HiPar56574.2022.00008.
- My OpenMP GPU porting tips. https://www.youtube.com/watch?v=iPGMYVViQzM OpenMP offload optimization guide
- AMD and INTEL GPU performance