Performance Portability with Compiler Directives for Lattice QCD in the Exascale Era

Meifeng Lin, Mohammad Atif
Computational Science Initiative, Brookhaven National Laboratory

May 8-12, 2023
CHEP2023, Norfolk, VA
Lattice QCD

• Lattice Quantum Chromodynamics (QCD) is a numerical framework to simulate the strong interactions between quarks and gluons.

• Continuous 4D space-time => 4D lattice after discretization

• Physical observables calculated from lattice QCD provide important insights to the QCD theory through comparisons with experimental results, e.g.
  • Internal structures of protons, pions, etc.
  • Bounds for new physics

• Key Algorithm Motifs
  • Markov Chain Monte Carlo
  • Sparse matrix inversions

• Computationally expensive; often requires years of running on leadership-class supercomputers to achieve %-level errors.
Exascale Meets Lattice QCD

- Exascale HPC systems in the US will feature different types compute accelerators, each with own native/preferred programming API
- Portability across different architectures is essential!

- ECP Application Development for Lattice QCD
  - 4 DOE labs: ANL, BNL, Fermilab, Jefferson Lab
  - 7 university partners: Boston University, Columbia University, University of Illinois, Indiana University, Stony Brook University, University of Utah, William and Mary

- 4 Working Groups targeting different areas:
  - Workflow/Contractions
  - Critical Slowing Down
  - Linear Solvers

- Data-Parallel API

Workflow
Applications
Algorithms
Data Parallel Frameworks
Libraries
Exascale Lattice QCD Software Suite

Applications
- Chroma
- MILC
- CPS
- HotQCD

Libraries
- QUDA
- Grid

Programming Model
- CUDA
- HIP
- SYCL/DPC++
- OpenMP

Target
- AMD GPU
- NVIDIA GPU
- Intel GPU

Multi-pronged approach
Currently focused on architecture-specific programming models for best performance
Also exploring OpenMP offloading for better portability
The Grid C++ QCD Library

- Grid[1] is a C++ library for lattice QCD
  - Initially designed for SIMD architectures with long SIMD length (Intel Knights Landing, Skylake, etc.).
  - Arranges the data layout as if the lattice is divided into virtual “sub-lattices”.
  - Each sub-lattice uses one SIMD lane.
- Same data layout can be mapped to GPU architectures
  - SIMD lanes on CPUs map to GPU threads
  - Requires some data manipulation under the hood

Grid’s Performance Portable Design

- Header file with macros to encapsulate architecture-dependent implementations

```c
#ifdef GRID_NVCC
#define accelerator __host__ __device__
#define accelerator_inline __host__ __device__ inline
#define accelerator_for (...) { //CUDA kernel}

#elif defined (GRID_OMP)
#define strong_inline __attribute__((always_inline)) inline
#define accelerator
#define accelerator_inline strong_inline
#define accelerator_for(...) thread_for(...) //for loop with #pragma omp parallel for

#endif
```

- Common MemoryManager API for dynamic memory allocation on different architectures

```c
void *MemoryManager::AcceleratorAllocate(size_t bytes){
    ...
    ptr = (void *) acceleratorAllocDevice(bytes);
}
```

Architecture-specific implementations
GridMini

www.github.com/meifeng/GridMini

- A substantially reduced version of Grid for easy experimentation with different programming models.

- Retains same Grid structure: data structures/types, data layout, aligned allocators, macros, ...

- Only keeps the high-level components necessary for the benchmarks.

- SU(3)\times SU(3) benchmark: STREAM-like memory bandwidth test

- Important as LQCD is bandwidth bound. Also data movement is the major challenge when porting to GPUs.

```cpp
LatticeColourMatrix z(&Grid); //Arrays of SU(3)
LatticeColourMatrix x(&Grid); //Arrays of SU(3)
LatticeColourMatrix y(&Grid); //Arrays of SU(3)

double start=usecond();
for(int64_t i=0;i<Nloop;i++){
    z=x*y;
}
double stop=usecond();
double time=(stop-start)/Nloop*1000.0;

double bytes=3*vol*Nc*Nc*sizeof(Complex);
double flops=Nc*Nc*(6+8+8)*vol;
double bandwidth=bytes/time; //GB/s
double Gflops=flops/time; //0.9 flops/byte SP
```
Different Programming Models Implemented

• **OpenACC/OpenMP**
  
  • **Pros**: Compiler directives-based approach; easy to add to existing code; portable across many platforms.
  
  • **Cons**: Use in C++ code non-trivial; Performance dependent on compilers; developer has not much fine-grained control.

• **CUDA/HIP/SYCL**
  
  • **Pros**: Vendor-supported API. More ways to control performance.
  
  • **Cons**: Need to write kernels manually; More verbose; Not all portable to different architectures (SYCL is portable).
OpenMP Offloading in Grid/GridMini

New macro definitions for `accelerator_for`, `accelerator_inline` etc.

```c
#elif defined (OMPTARGET)
#define accelerator_inline strong_inline
#define accelerator_for(iterator,num,nsimd, ...) \
{ \
   _Pragma("omp target teams distribute parallel for") \
   naked_for(iterator, num, { \
   __VA_ARGS__ }); \
}
```

```c
inline void *acceleratorAllocDevice(size_t bytes) {
   int devc = omp_get_default_device();
   ptr = (void *) omp_target Alloc(bytes, devc);
}
```

MemoryManager with OpenMP APIs

```
Unified Shared/Virtual Memory for Comparison
```

```c
#ifdef OMPTARGET_MANAGED
   if ( ptr == (_Tp **) NULL ) auto err = cudaMallocManaged((void **) &ptr, bytes);
```

Can also specify # of threads/blocks
GridMini Performance on NVIDIA GPU

- **llvm map**: explicit data mapping with OpenMP offloading with malloc as the memory allocator
- **llvm managed**: OpenMP offloading with cudaMallocManaged as memory allocator
- **llvm map+managed**: explicit data mapping with cudaMallocManaged as memory allocator
- **nvcc managed**: CUDA implementation with cudaMallocManaged (same data layout; no CUDA-specific optimizations)

- **Compiler Version**:
  - clang++: llvm/12.0.0-git_20210117
  - nvcc: CUDA 11
- **Hardware platform**: Cori-GPU with NVIDIA V100 GPU


Grid OpenMP offloading Performance

- Choice of # of threads/block affects performance.
- OpenMP and CUDA have different optimal values

OpenMP Bandwidth on NVIDIA V100

<table>
<thead>
<tr>
<th># Threads/Block</th>
<th>Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>0</td>
</tr>
<tr>
<td>768</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
</tr>
</tbody>
</table>

CUDA Bandwidth on NVIDIA V100

<table>
<thead>
<tr>
<th># Threads/Block</th>
<th>Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td>0</td>
</tr>
</tbody>
</table>

L=24, memory footprint = 1.43E+08 bytes
Compilers: Clang-15.0.0 + CUDA-11.4
GridMini Performance on AMD GPU

- Compiler Version:
  - Rocm4.5
- Hardware platform: BNL lambda1 with AMD Radeon Pro VII GPU and AMD 24-core Ryzen Threadripper 3960X CPU

- L=24, memory footprint= 1.43E+08 bytes
- Best performance is with 256 threads/block
Summary

• Porting full Grid to OpenMP offloading is in progress.
• Starting from the miniapp laid a good roadmap for porting.
• GridMini runs on NVIDIA, AMD and Intel GPUs.
• However, moving from GridMini to Grid still exposes many issues:
  • Layered abstraction makes it hard to identify bugs with data movement => often the main point of failure.
  • At the moment Grid does not compile for AMD GPUs (with rocm clang compiler), failing with stack size overflow.
  • Compilers are constantly evolving: Good – bugs get fixed quickly; Bad – performance can degrade due to internal compiler changes.
• Performance can also depend on runtime parameters (# of threads/block, etc.)
  • important to perform manual/auto tuning.
Acknowledgments

- This work 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.