Experience with using OpenMP offloading to achieve performance portability for the Grid lattice QCD library

Meifeng Lin (Brookhaven National Laboratory)

*Collaborators:* Peter Boyle (BNL/U. Edinburgh), Lingda Li (BNL), Kate Clark (NVIDIA), Alejandro Vaquero (U. Utah), Vivek Kale (BNL), Barbara Chapman (BNL/SBU)

P3HPC Forum, September 1-2, 2020
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
Computational Kernel

• The core computational kernel of lattice QCD is matrix vector multiplications – the so-called Dslash operator.

\[ D_{\alpha\beta}^{ij}(x, y)\psi_{\beta}(y) = \sum_{\mu=1}^{4} \left[ (1 - \gamma_{\mu})_{\alpha\beta} U_{\mu}^{ij}(x) \delta_{x+\bar{\mu},y} + (1 + \gamma_{\mu})_{\alpha\beta} U_{\mu}^{ij}(x + \bar{\mu}) \delta_{x-\bar{\mu},y} \right] \psi_{\beta}(y) \]

– \( x, y \): 4D coordinates
– \( \gamma_{\mu} \): 4×4 matrices (fixed)
– \( U_{\mu}(x) \): complex 3×3 matrices, 4 per lattice site (main memory usage)
– \( \psi(y) \): complex 12-component vectors, 1 per site (main memory usage)

• Matrix-vector multiplication form is known analytically. No actual matrices are stored.
• Memory requirements per site: \((9\times2\times4 + 12\times2)\times8 = 768 \) bytes (DP)
• Floating point operations for Wilson Dslash: 1320 flops per site
• Low arithmetic intensity: 1.7 flops/byte (DP) or 3.4 flops/byte (SP)
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
• C++11 (lambda, auto types, etc.)
• Extensive use of templates for high-level abstraction
• Custom expression template engine for performance

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}
#else
#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
```

• Custom AlignedAllocator for dynamic memory allocation on different architecture

```c
#ifdef GRID_NVCC
    if ( ptr == (_Tp *) NULL ) auto err = cudaMallocManaged((void **)&ptr,bytes);
#else
    #ifdef HAVE_MM_MALLOC_H
        if ( ptr == (_Tp *) NULL ) ptr = (_Tp *) _mm_malloc(bytes,GRID_ALLOC_ALIGN);
    #else
    ...
```
Grid Structure

- Grid
- HMC
- Hadrons
- benchmarks
- documentation
- gcc-bug-report
- m4
- scripts
- tests

Core Software

- algorithms
  - allocator
  - cartesian
  - communicator
  - cshift
  - json
  - lattice
  - log
  - parallelO
  - perfmon
  - pugixml

- qcd
  - serialisation
  - simd
  - simlo_rng
  - stencil
  - tensors
  - threads
  - util

- approx
  - iterative
  - Algorithms.h
  - CoarsenedMatrix.h
  - FFT.h
  - LinearOperator.h
  - Preconditioner.h
  - SparseMatrix.h

- action
  - hmc
  - modules
  - observables
  - representations
  - smearing
  - spin
  - utils
GridMini

- A substantially reduced version of Grid for 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)×SU(3) benchmark**: STREAM-like memory bandwidth test
- Important as LQCD is bandwidth bound.

```cpp
Benchmark_su3
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
```
Previous Work

- Previously[2] explored porting Grid to GPUs using OpenACC, JIT and CUDA.
- Use \texttt{coalesced\_ptr}[3] to force data coalescing to get best performance.

- In the \textbf{CUDA implementation}, Grid’s SIMD data layout can be used to force data coalescing.
- Same data layout across SIMD and SIMT archs.

OpenMP Offloading

• To add OpenMP offloading (to NVIDIA GPUs) for Benchmark_su3, only two changes required.

• New macros

```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__ }); \
}
```

• Use `cudaMallocManaged` for the memory allocator (for now)

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

- **Deep copy:** explicit data mapping **nontrivial** due to deep nested data structures
  - Managed/Unified memory greatly simplifies things

- **Incorrect results:** output always 0 after offloading
  - Compiler bug related to the use of `struct` of short vectors as device function return type.

```c
struct vec {
    float v[2];
};

inline vec mult(vec x, vec y){
    vec out;
    out.v[0]=x.v[0]*y.v[0];
    out.v[1]=x.v[1]*y.v[1];
    return out; //causing issue here
}
```

```c
vec in1,in2;
vec out[N];
#pragma omp target teams distribute parallel for
map(to:in1,in2) map(from:out[0:N])
for(int n=0;n<N;n++) {
    out[n]=mult(in1,in2);
}
```

- Fixed as of llvm/12.0.0-git_20200731

Got: 0.000000 0.000000
Expected: 2.000000 2.000000
Compiling and Performance

- Compiling with clang++ *(Original)*

```bash
CXX=clang++
CXXFLAGS=-std=c++14 -g -fopenmp -O3 -fopenmp-targets=nvptx64-nvidia-cuda \ -lcudart
```

- Performance of Benchmark_su3 maxed out at **125 GB/s** on NVIDIA V100 (Cori-GPU)
  - Peak memory bandwidth should be 900 GB/s

- Learned about `-fopenmp-cuda-mode` at the SOLLVE OpenMP Hackathon*.

```bash
CXX=clang++
CXXFLAGS=-std=c++14 -g -fopenmp -O3 -fopenmp-targets=nvptx64-nvidia-cuda \ -lcudart -fopenmp-cuda-mode
```

  - Guarantees to the compiler that the target region is in SPMD mode
  - Performance improved significantly (5X)!

*Special thanks: Rahul Gayatri (LBNL), Johannes Doerfert (ANL)*
SU(3)xSU(3) Performance Comparison

- With -fopenmp-cuda-mode
- thread_limit(8) seems to be slightly better than 32 or 128.
- CUDA bandwidth plateaus much earlier: why?
- With large data, OpenMP version is about 90% of CUDA performance.
- With small data sizes, OpenMP performs much worse: OpenMP version has more overhead?

*benchmark performed on Cori GPU (V100); llvm/12.0.0_git_20200731; cuda/10.1.243
Profiling

Culprit for poor performance at small memory footprints: 30% time spent on `cuMemAlloc/cuMemFree`

- A known issue of LLVM when many short lived objects with non-overlapping lifetimes are mapped.

- **Solution:** Keep and reuse previously freed memory instead of giving and to CUDA and asking for it again. See [https://reviews.llvm.org/D81054](https://reviews.llvm.org/D81054)

- Pool allocation is next to reduce the cost for many consecutive small allocations with overlapping lifetimes. See [https://reviews.llvm.org/D85274](https://reviews.llvm.org/D85274)
A patch exists to optimize the memory management in LLVM.

The patch improves the performance significantly, in particular with small- to medium-sized data footprints.

The performance is on par with the CUDA version for the most part.

The patch has been merged into the LLVM mainline.

*benchmark performed on Cori GPU (V100); patched version uses llvm/12.0.0_git_20200731-shilei; cuda/10.1.243
Progression of Performance (LLVM12)

- Original gets best performance with 128 threads (also compiler default).
- 8 threads per block seems to be the sweet spot with `-openmp-cuda-mode`, both patched and unpatched.
- Setting `num_teams` together with `thread_limit` doesn’t seem to have much effect, since compiler generates `num_teams` based on # of threads and the loop count, same as the manual `num_teams`.

*benchmark performed on Cori GPU (V100); patched version uses llvm/12.0.0-git_20200731-shilei; others use llvm/12.0.0-git_20200731; cuda/10.1.243
Summary and Next Steps

• OpenMP offloading to GPUs seems feasible in GridMini.
• LLVM compiler support for omp target has improved a lot!
• Communication with the compiler team is very important.
  – Benefits go both ways.

• Next Steps:
  – Make the code truly portable: replace cudaMallocManaged with either explicit target data clauses or OpenMP’s unified_shared_memory clause (when supported).
  – Investigate the effects of SIMD vs Scalar (SIMD length=1) data layout
    • right now using Scalar data layout.
  – Move on to the Dslash benchmark: this is our true interest.
Acknowledgments

• ML thanks Johannes Doerfert (ANL) and Rahul Gayatri (LBNL) for their help during the OpenMP Hackathon on August 3-7, 2020.

• Some of the benchmarks used the computing resources at NERSC and OLCF.

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations – the Office of Science and the National Nuclear Security Administration – responsible for the planning and preparation of a capable exascale ecosystem – including software, applications, hardware, advanced system engineering, and early testbed platforms – to support the nation’s exascale computing imperative. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract No. DE-AC05-00OR22725.