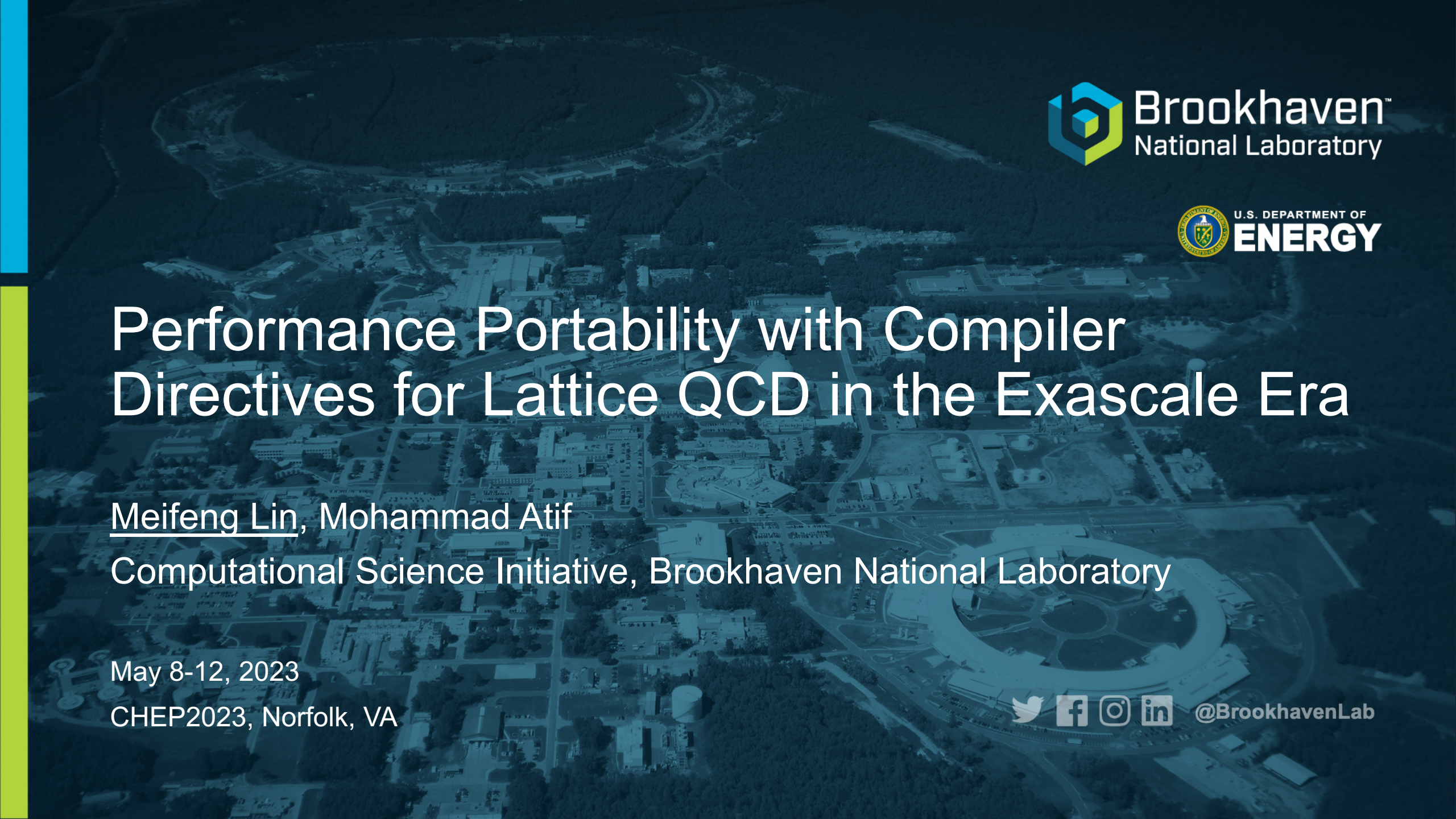


Performance Portability with Compiler Directives for Lattice QCD in the Exascale Era

Meifeng Lin, Mohammad Atif

Computational Science Initiative, Brookhaven National Laboratory

May 8-12, 2023

CHEP2023, Norfolk, VA

@BrookhavenLab

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.

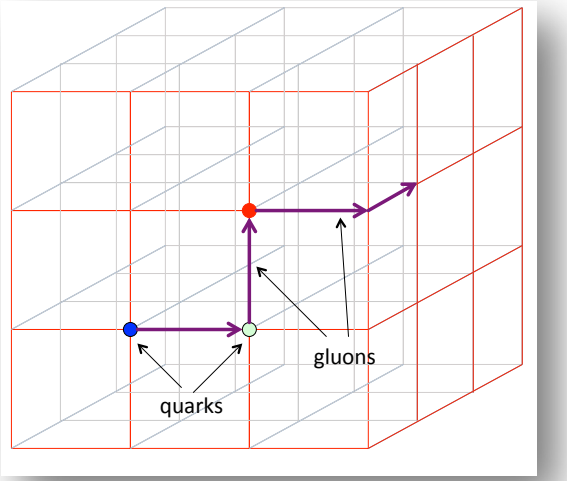
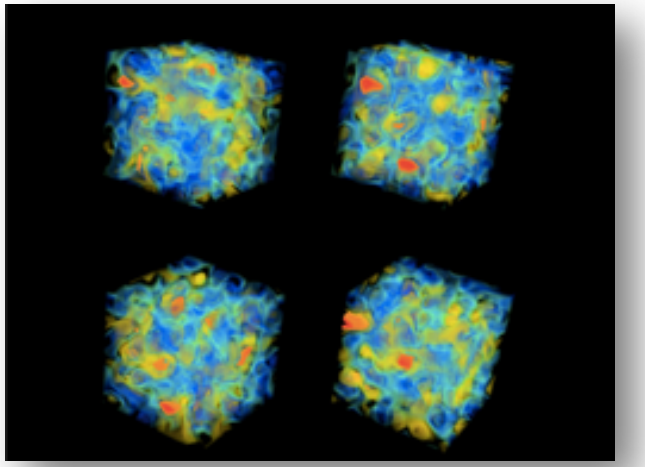


Illustration of a 3D lattice



Visualization of QCD topological charge density. Credit: Brookhaven National Laboratory

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!**

NVIDIA GPUs (CUDA)

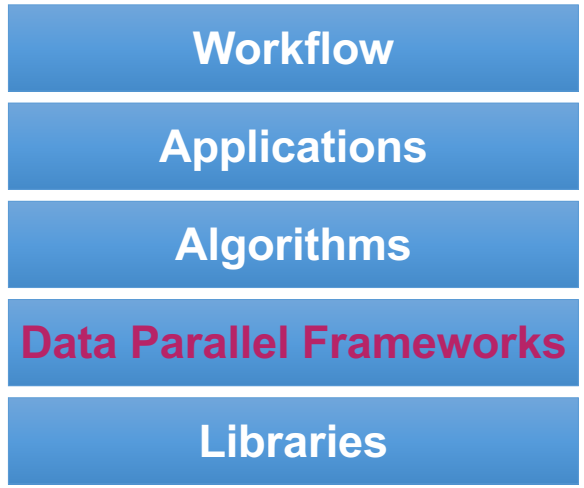
Perlmutter (Pre-Exascale)

Intel GPUs (SYCL)

AMD GPUs (HIP)

Aurora Frontier

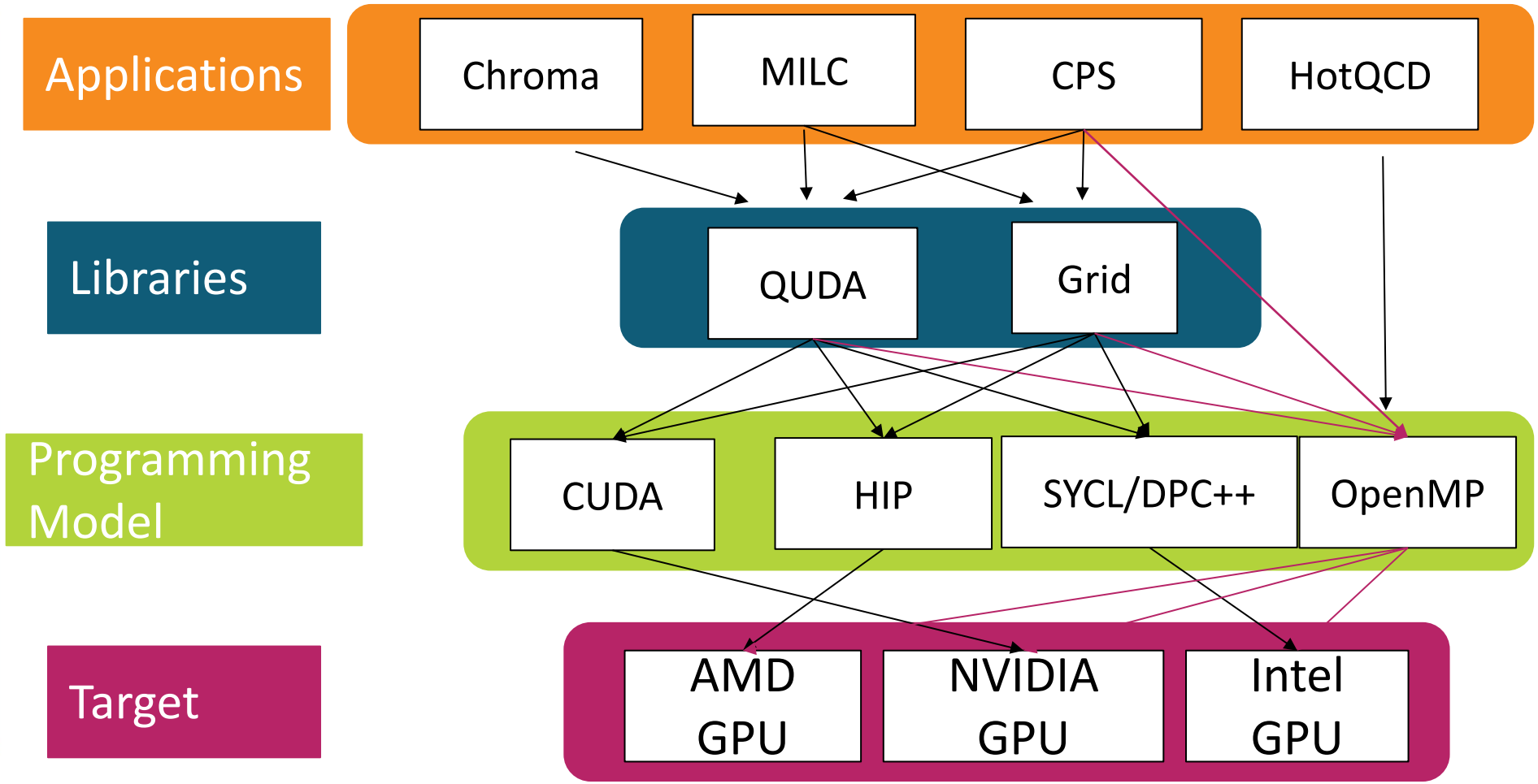
- **ECP Application Development for Lattice QCD**
  - 4 DOE labs: ANL, BNL, Fermilab, Jefferson Lab
  - 7 university partners: Boston University, Columbia University, University of Illinois, Indiana University, Stony Brook University, University of Utah, William and Mary
- **4 Working Groups targeting different areas:**
  - Workflow/Contractions
  - Critical Slowing Down
  - Linear Solvers
  - **Data-Parallel API**



Brookhaven National Laboratory

ECP EXASCALE COMPUTING PROJECT

Exascale Lattice QCD Software Suite



Multi-pronged approach

Currently focused on architecture-specific programming models for best performance

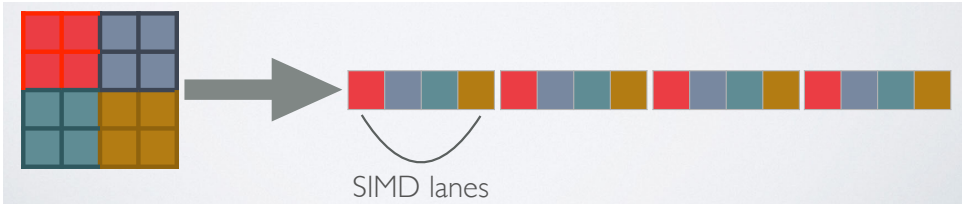
Also exploring OpenMP offloading for better portability

Brookhaven National Laboratory

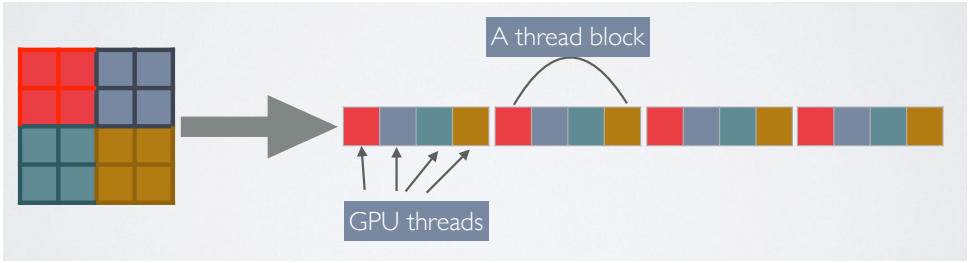
ECP EXASCALE COMPUTING PROJECT

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



Data mapping on SIMD architecture



Data mapping on SIMT architecture

[1] P. Boyle et al., arXiv:1512.0348, https://github.com/paboyle/Grid

Grid's Performance Portable Design

- Header file with macros to encapsulate architecture-dependent implementations

```
#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
```

- Common MemoryManager API for dynamic memory allocation on different architectures

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

Architecture-specific implementations

Brookhaven National Laboratory

ECP EXASCALE COMPUTING PROJECT

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)×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.

Benchmark_su3

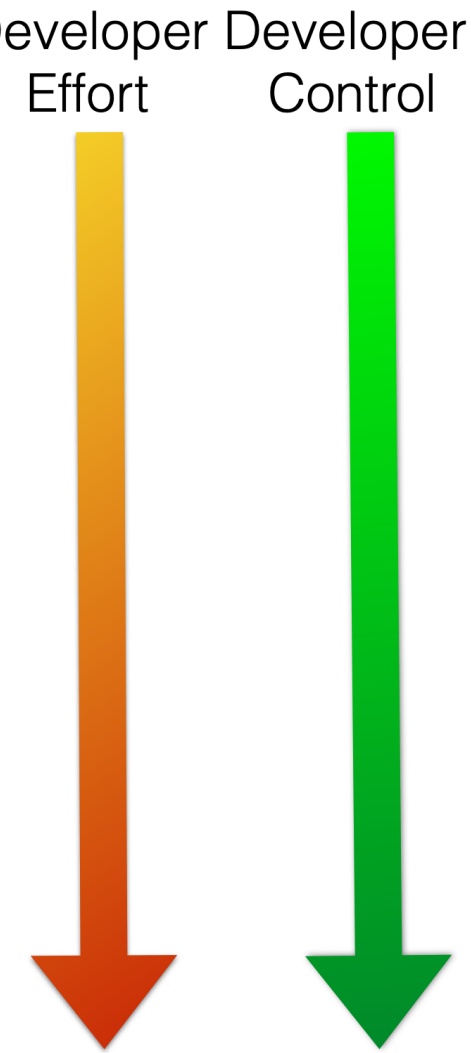
```
LatticeColourMatrix z(&Grid); //Arrays of SU(3)
LatticeColourMatrix x(&Grid); //Arrays of SU(3)
LatticeColourMatrix y(&Grid); //Arrays of SU(3)

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
```

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.

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

Can also specify # of threads/blocks

Compute

MemoryManager with OpenMP APIs

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

Unified Shared/Virtual Memory for Comparison

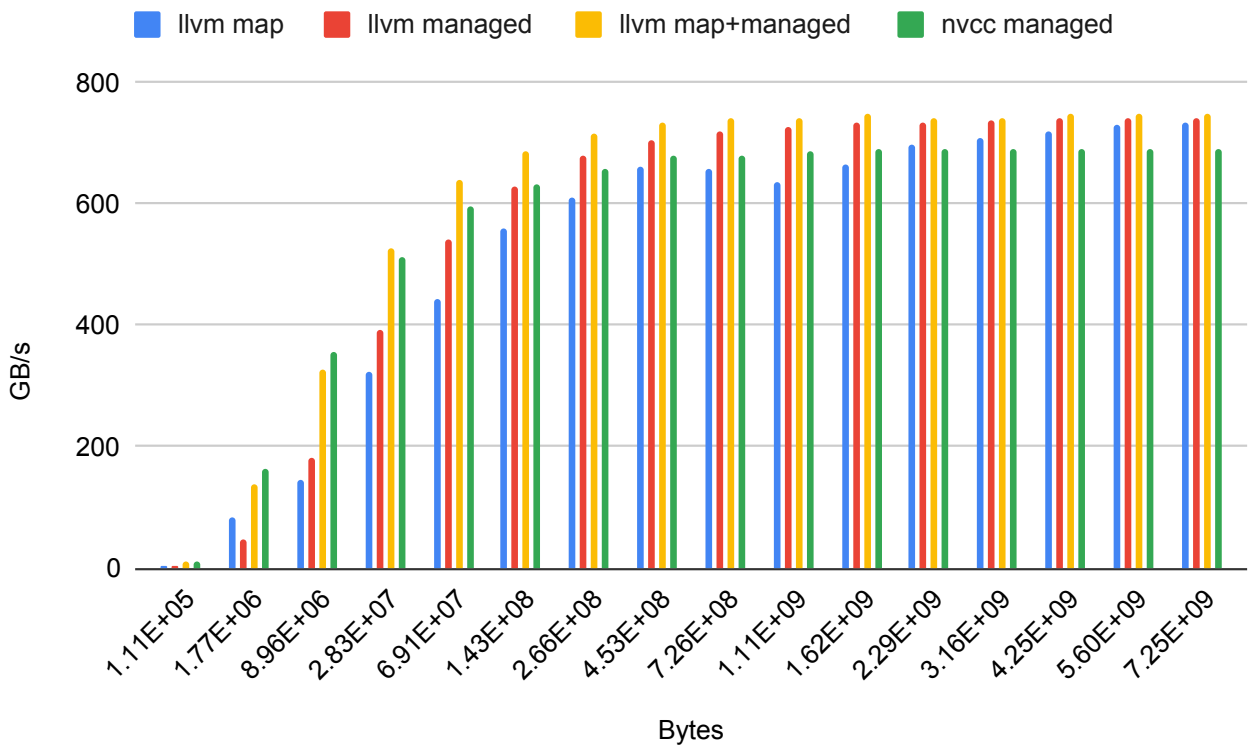
```
#ifdef OMPTARGET_MANAGED
    if ( ptr == (_Tp *) NULL ) auto err = cudaMallocManaged((void **)&ptr,bytes);
```

Data

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**

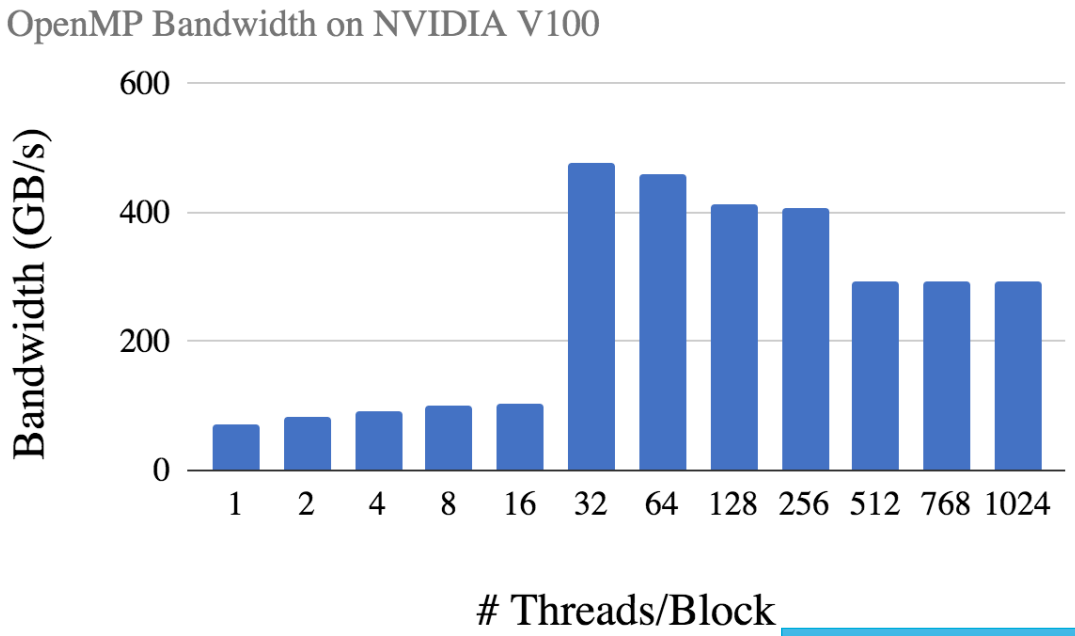


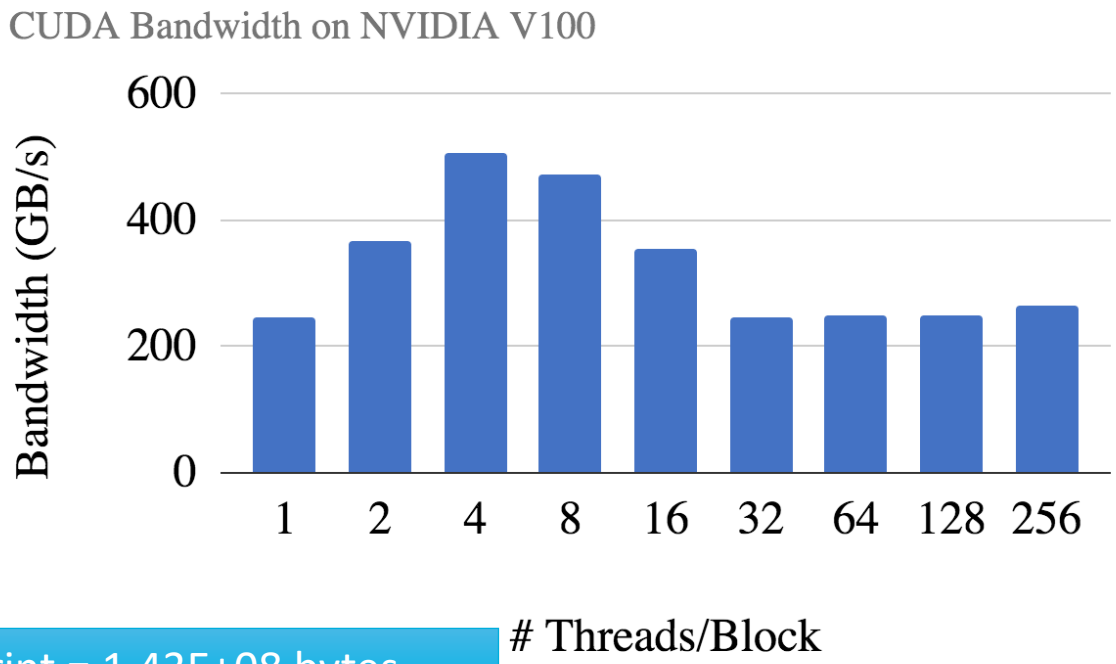
Bak, Seonmyeong, et al. "OpenMP application experiences: porting to accelerated nodes." *Parallel Computing* 109 (2022): 102856.

Chapman, Barbara, et al. "Outcomes of OpenMP Hackathon: OpenMP Application Experiences with the Offloading Model (Part I&II)." *International Workshop on OpenMP*. Springer, Cham, 2021.

Grid OpenMP offloading Performance

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



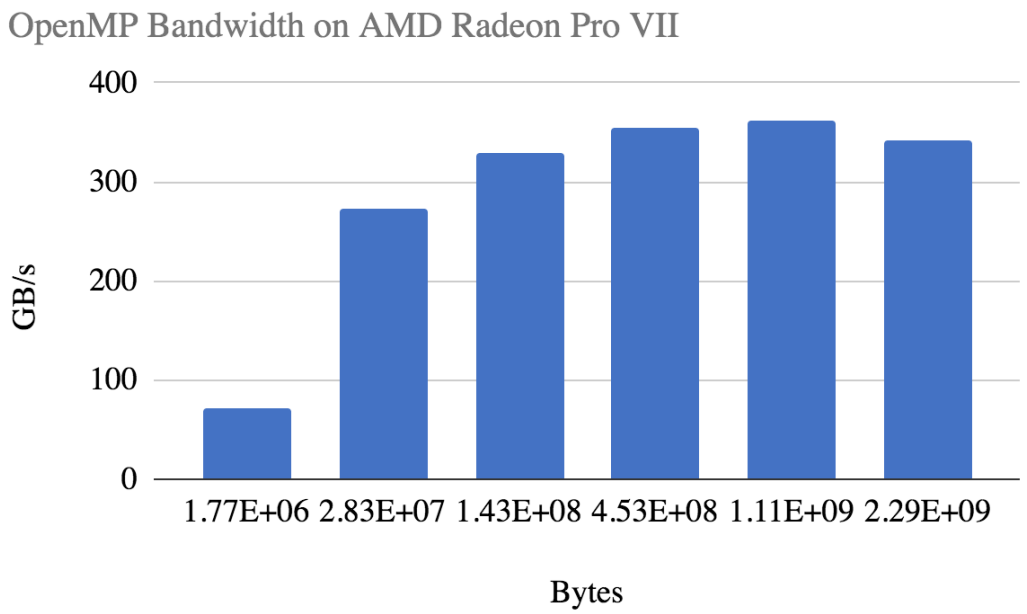


L=24, memory footprint = 1.43E+08 bytes
Compilers: Clang-15.0.0 + CUDA-11.4

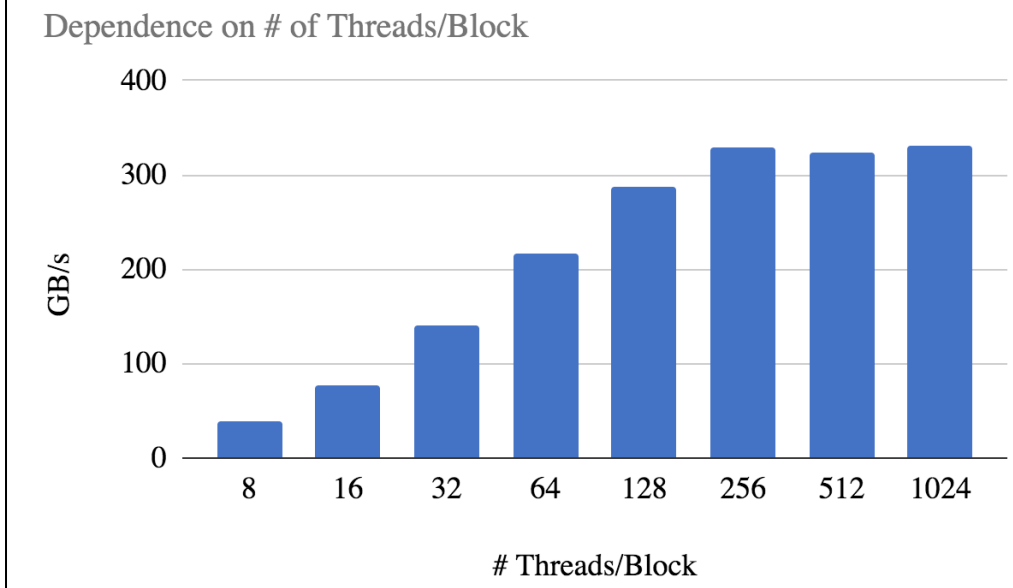
Brookhaven National Laboratory

ECP EXASCALE COMPUTING PROJECT

GridMini Performance on AMD GPU



- **Compiler Version**:
  - Rocm4.5
- **Hardware platform**: BNL lambda1 with **AMD Raedon 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

Brookhaven National Laboratory

ECP EXASCALE COMPUTING PROJECT

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.*