Status of OpenMP Target Offloading in Grid

Meifeng Lin
Computational Science Initiative, Brookhaven National Laboratory

July 31-August 4, 2023
Lattice 2023, Fermilab, Batavia, IL

@BrookhavenLab

Exascale Meets Lattice QCD

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

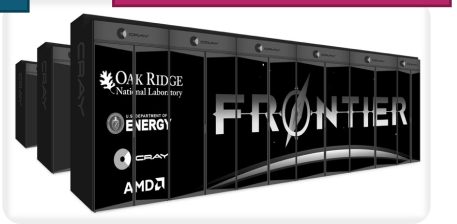
NVIDIA GPUs (CUDA)

Perlmutter (Pre-Exascale)

Intel GPUs (SYCL)

Aurora

AMD GPUs (HIP)



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**

Workflow

Applications

Algorithms

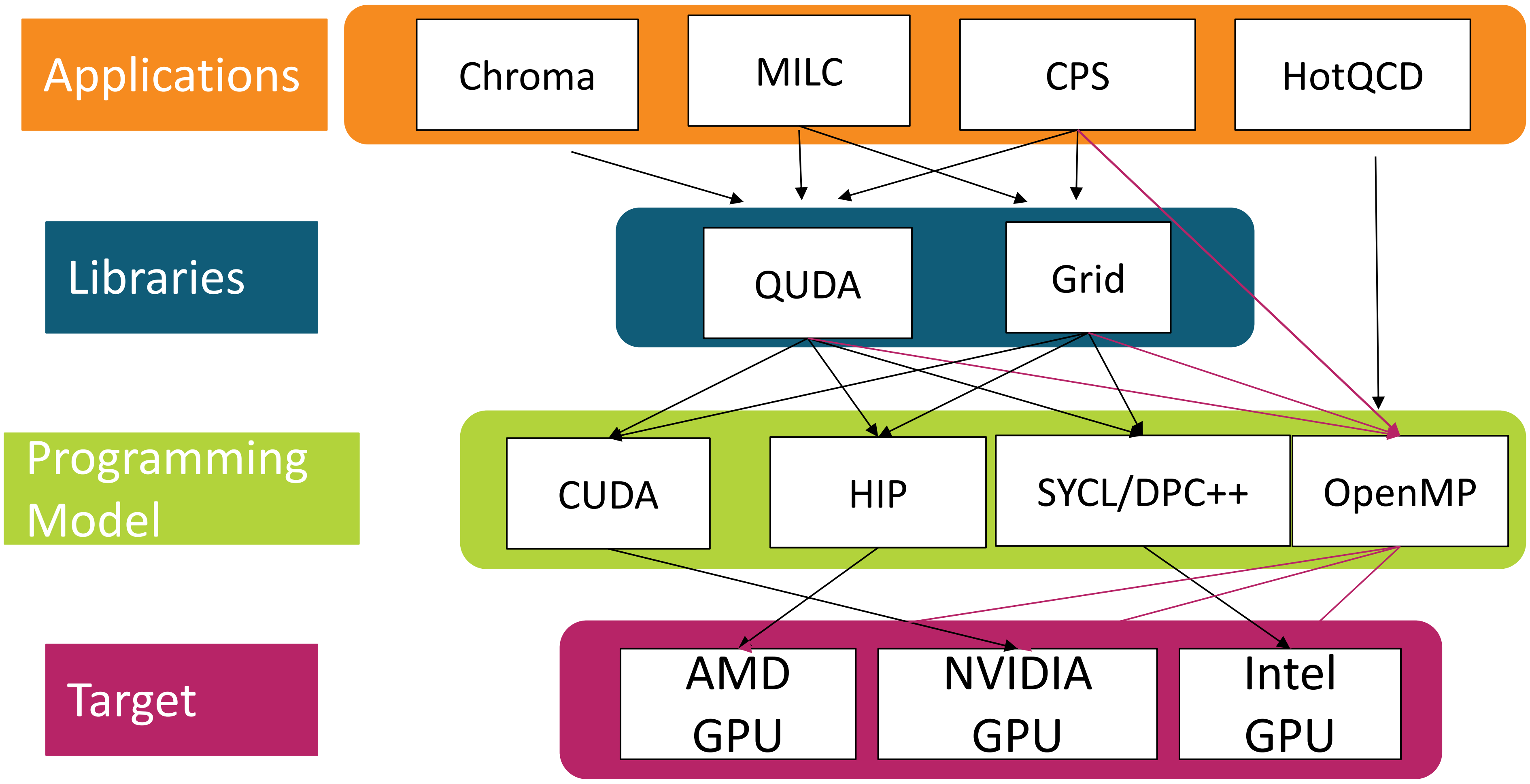
Data Parallel Frameworks

Libraries

Brookhaven National Laboratory

ECP EXASCALE COMPUTING PROJECT

US 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

OpenMP

OpenMP is an API for multithreading that was first developed in 1997 for Fortran.

Later, support for C/C++ was added.

Originally it only supported Shared-Memory parallel computing on multicore architectures.

Since OpenMP 4.0, it added support for “target offloading” on heterogenous architectures, such as CPU+GPU.

Version 5.2 was released in November 2021. PDF/HTML versions are on www.openmp.org. Book on Amazon.

Now supports several programming and memory models, including shared-memory parallelism, task parallelism, and host-device heterogenous computing.

API specification in more than 600 pages!

www.openmp.org

Brookhaven National Laboratory

ECP EXASCALE COMPUTING PROJECT

OpenMP for Shared-Memory Parallelism

OpenMP uses the fork-join model for multithreading.

- The main thread will spawn several parallel child threads when a parallel region is encountered.
- The parallel threads will re-join once exiting the parallel region.

Shared Memory: All the threads have access to the same memory space.

- No on-node data transfer needed.
- Need to avoid data race: when more than 1 thread tries to access the same memory.

OpenMP uses a set of compiler directives and API function calls.

```
!$OMP PARALLEL
PRINT *, “Hello from process: ”,
OMP_GET_THREAD_NUM()
!$OMP END PARALLEL
```

```
#pragma omp parallel
{
   printf(“Hello from process: %d\n”,
omp_get_thread_num());
}
```

Starts a parallel region

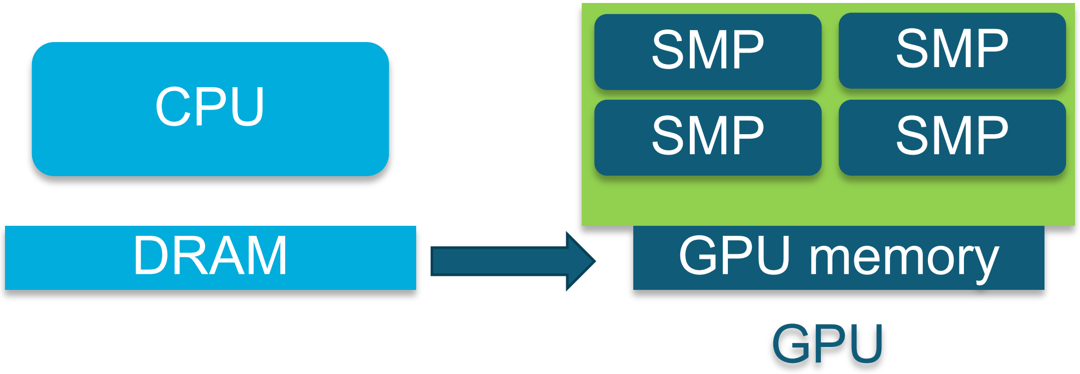
Compatible compilers will use “-fopenmp” or similar to enable OpenMP parallelization. The OpenMP directives are ignored if the compiler does not support them.

Brookhaven National Laboratory

EXASCALE COMPUTING PROJECT

OpenMP for GPU Computing

- To enable GPU computing, OpenMP uses the “target offloading” model.
- When the target region is encountered, the main thread will attempt to initiate the computation on the target device, e.g., the GPU in this case.
- Data will be moved to/from the GPU as needed/specified by the user.
- Two ways to do this:
  - Explicit data management
  - Managed memory



- **OpenMP is a specification**; actual support and implementations for different GPU architectures depend on the compilers.

Brookhaven National Laboratory
ECP EXASCALE COMPUTING PROJECT

A simple example

```
int main(int argc, char* argv[]){
        int N=10000;
        float x=1.0;
        float y=2.0;
        float out[N];
#pragma omp target teams distribute parallel for \
    map(to:x,y) map(from:out[0:N])
        for(int n=0;n<N;n++) {
                out[n]=x*y;
}
return 0;

}
```

Can compile with gcc for NVIDIA GPUs:
g++ -fopenmp -omptargets=nvptx64sm_75-nvidia-linux

target – indicates the code block below will be executed on the target device.

teams – indicates there will be a league of teams doing the work

distribute – the teams will share the work (usually outer loop iterations)

parallel – the work will be shared by parallel threads

How teams/distribute/parallel map to the GPU architectures depends on the compiler

map copies the data associated with the variables to or from the target memory.

Brookhaven National Laboratory

ECP EXASCALE COMPUTING PROJECT

Comparison with CUDA - Kernels

OpenMP

- GPU kernels can be generated implicitly by the compiler inside target region for inline functions.

```
main:
     9, #omp target teams distribute parallel for
        9, Generating Tesla and Multicore code
           Generating "nvkernel_main_F1L9_1" GPU kernel
       11, Loop parallelized across teams and threads, schedule(static)
```

- There is no additional kernel launch call. Kernel launch is implicit inside the target region with default thread/teams numbers.
- Can also write specific kernel functions with #pragma omp declare target
- Can specify #of teams/# of threads by num_teams and thread_limit

  #pragma omp target teams distribute parallel for num_teams(32) thread_limit(128)

CUDA

- GPU kernel functions need to be explicitly defined with __global__ decorator

```
if (i < n) y[i] __global__ void saxpy(int n, float a, float *x, float *y) {   int i = blockIdx.x*blockDim.x + threadIdx.x;    = a*x[i] + y[i]; }
```

- Kernel launch with <<<, >>>

```
saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
```

- Need to specify # of threadblocks/threads explicitly.
- Note that you need to have the device pointers (unless you use UVM) in the kernel calls.

Comparison with CUDA – Data Management

OpenMP

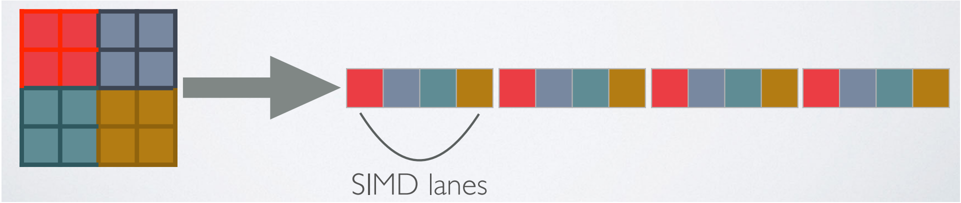
- OpenMP uses the “map” clauses to manage data between CPU and GPU
- #pragma omp target map (tofrom:x)
- to/from is from the host perspective
- Some data are copied implicitly at the kernel launch, such as scalars (firstprivate by default)
- Can use unstructured data clauses for more flexibility
  - #pragma omp target enter data map(alloc:x[0:N])
  - #pragma omp target exit data map(from:x[0:N])
- Also supports API calls, e.g., omp_target_alloc, etc.

CUDA

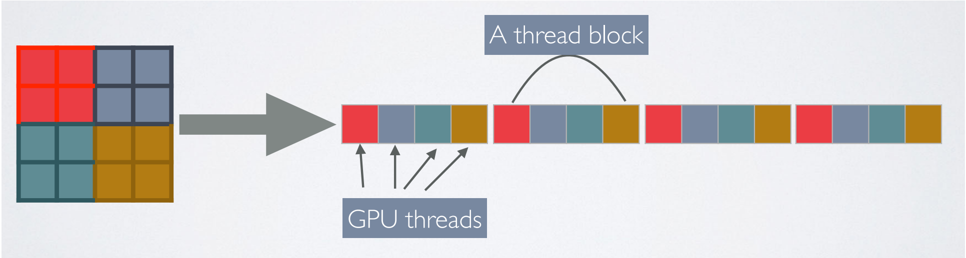
- Need to allocate host and device memory explicitly
- Need to be careful about which pointers to use, host or device
- cudaMemcpy copies data to/from the device
- Unified Virtual Memory/Managed Memory greatly simplifies data management with cudaMallocManaged allocator for both the host and device memory.
- CUDA runtime page faults retrieve device/host data as necessary

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

Brookhaven National Laboratory
EXASCALE COMPUTING PROJECT

Grid's Performance Portable Design

- Header file with macros to encapsulate architecture-dependent implementations
- Currently the main Grid repo supports **CUDA**, **SYCL** and **HIP**

```
#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

OpenMP Offloading in Grid

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

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.
- Useful in the early days of OpenMP offloading experiments as the compilers were being developed.

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
```


Brookhaven National Laboratory
ECP EXASCALE COMPUTING PROJECT

Summary of Current Status

- Porting full Grid to OpenMP offloading is in progress.
  - Added OpenMP target backend for both the compute and data management.
  - Haven’t added SIMT layout support to the OpenMP target backend.
  - Code compiles and runs on NVIDIA and Intel GPUs using LLVM-based compilers. There are still some linking issues on AMD GPUs (stack size overflow).
- Starting from the **miniapp laid a good roadmap for porting**.
  - GridMini runs on NVIDIA, AMD and Intel GPUs, and works with different compilers.
- 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.
  - 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.

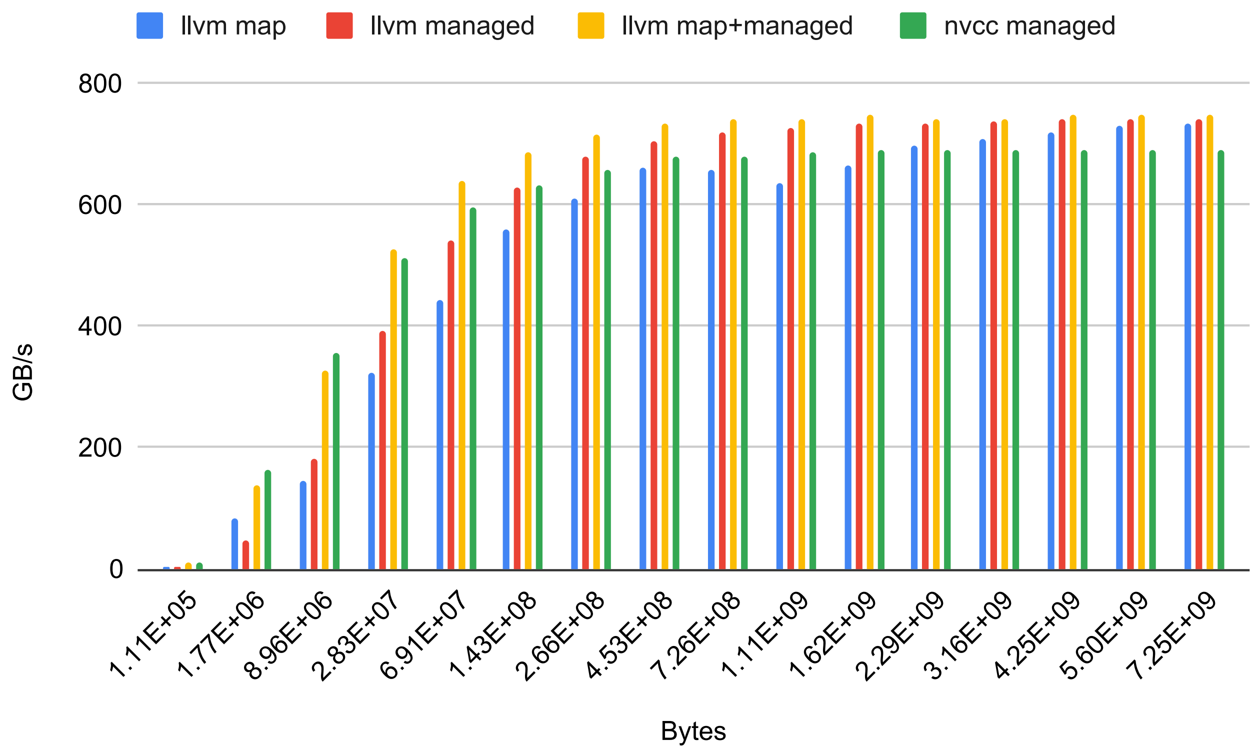
Brookhaven National Laboratory

ECP EXASCALE COMPUTING PROJECT

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**

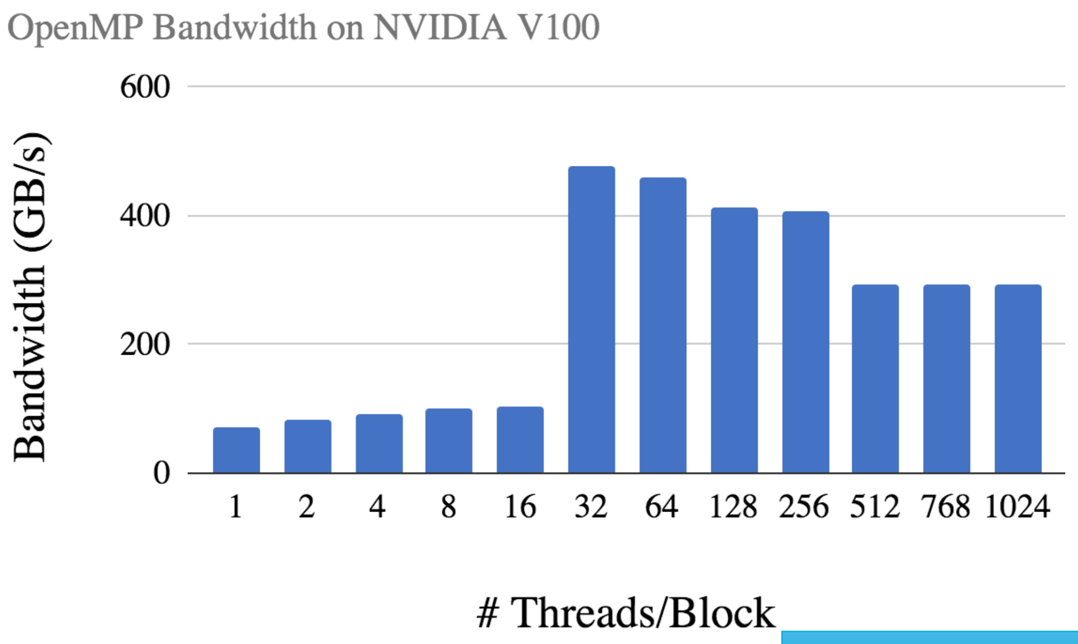


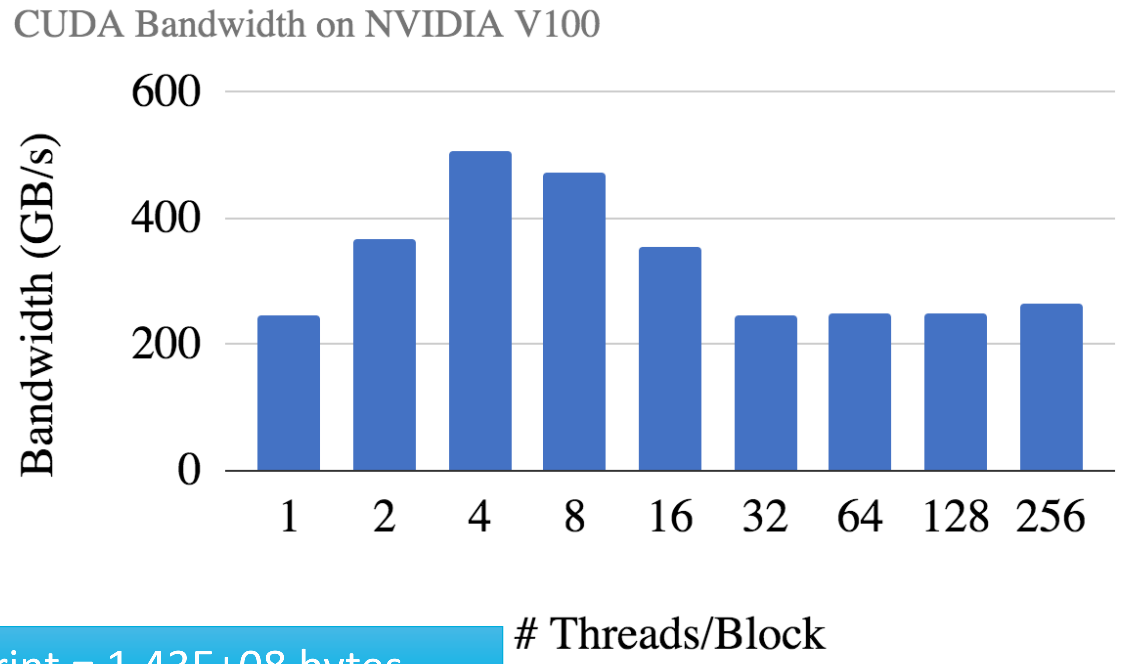
Bak, Seonmyeong, et al. "OpenMP application experiences: porting to accelerated nodes." *Parallel Computing* 109 (2022): 102856.

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

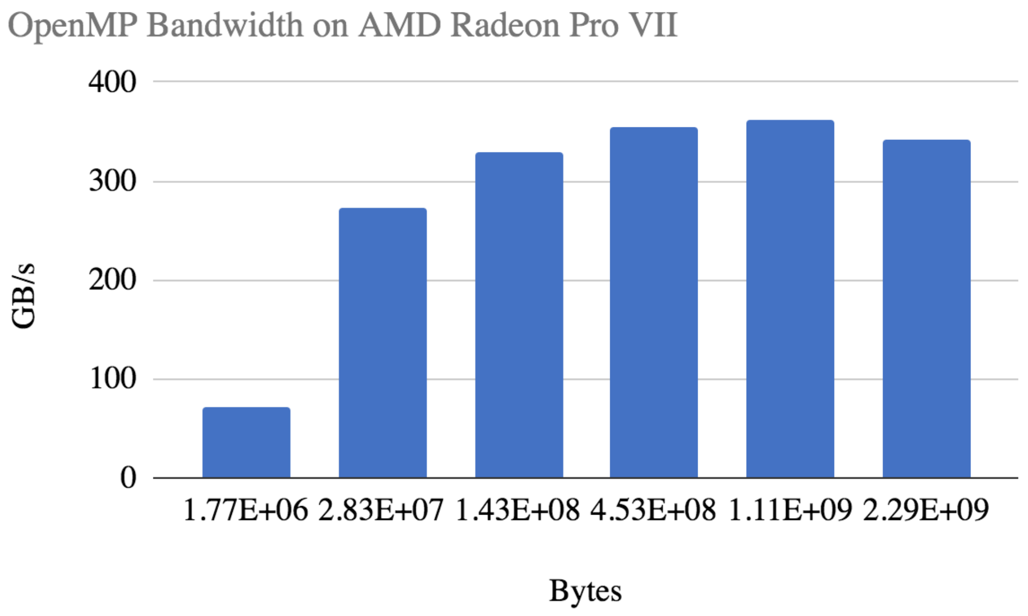
- Choice of # of threads/block affects performance.
- OpenMP and CUDA have different optimal values



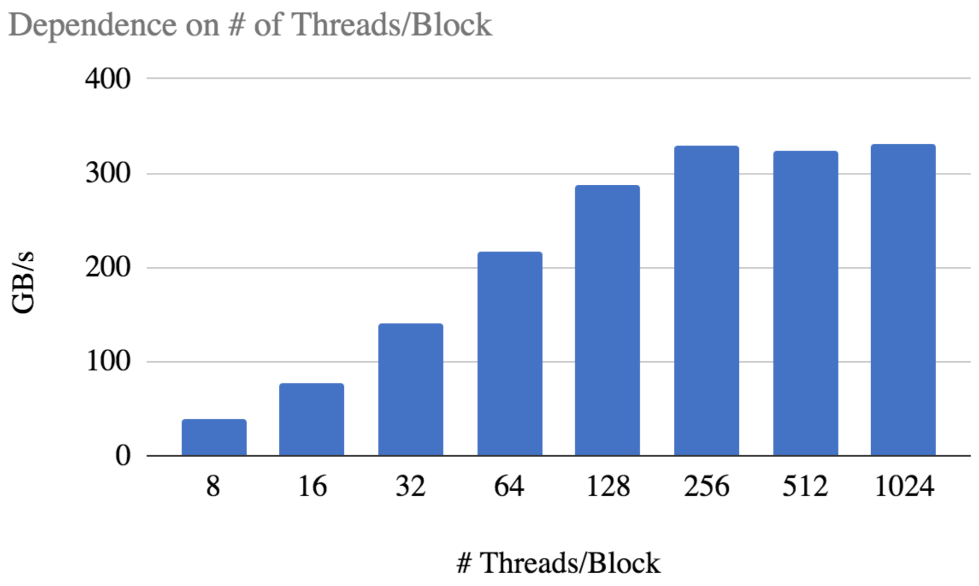


L=24, memory footprint = 1.43E+08 bytes
Compilers: Clang-15.0.0 + CUDA-11.4

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

Conclusions and Outlook

- Compiler support for OpenMP target offloading has improved greatly in the past few years.
- However, getting OpenMP offloading to work with complicated C++ codes such as Grid is still quite challenging.
  - Grid has exposed many issues with the current compilers.
  - We have worked very closely with the LLVM compiler developers to identify and fix these issues.
- Debugging, testing and performance tuning on Frontier and Aurora hardware is in progress.
- TODO: Comparison with CUDA/HIP/SYCL implementations.

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.***

Brookhaven National Laboratory

ECP EXASCALE COMPUTING PROJECT