OAK RIDGE National Laboratory

Performance Portability of Molecular Docking Application for Exascale Architectures using OpenMP Offloading: Challenges and Solutions.

Mathialakan Thavappiragasam
Biophysics Group, Biosciences Division,
Oak Ridge National Laboratory

Thanks to:

Ada Sedova
Biophysics Group, Biosciences Division,
Oak Ridge National Laboratory

Wael Elwasif
Computer Science & Mathematics Division,
Oak Ridge National Laboratory

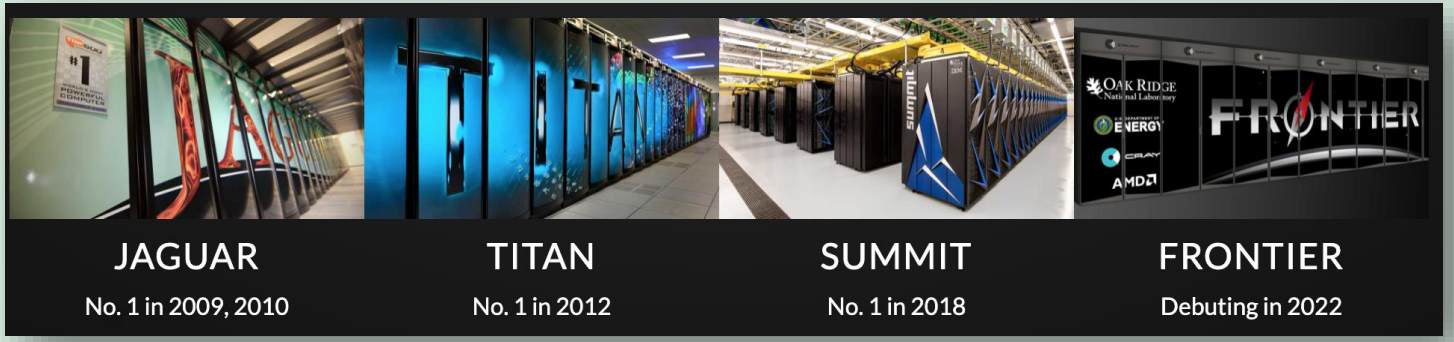
OAK RIDGE National Laboratory

Agenda

- Performance Portability
- miniMDock
- Translating CUDA to HIP
- Translating CUDA to OpenMP Target Offloading
- Performance Evaluation and Enhancement

Performance Portability

- **Changing Computer Architectures**: Changes on the HPC facilities. Heterogeneous multi-node systems that uses accelerators such as GPUs together with CPUs have been taken lead in providing performance for many different type of applications.



- **Diversity in computer architecture for HPC**

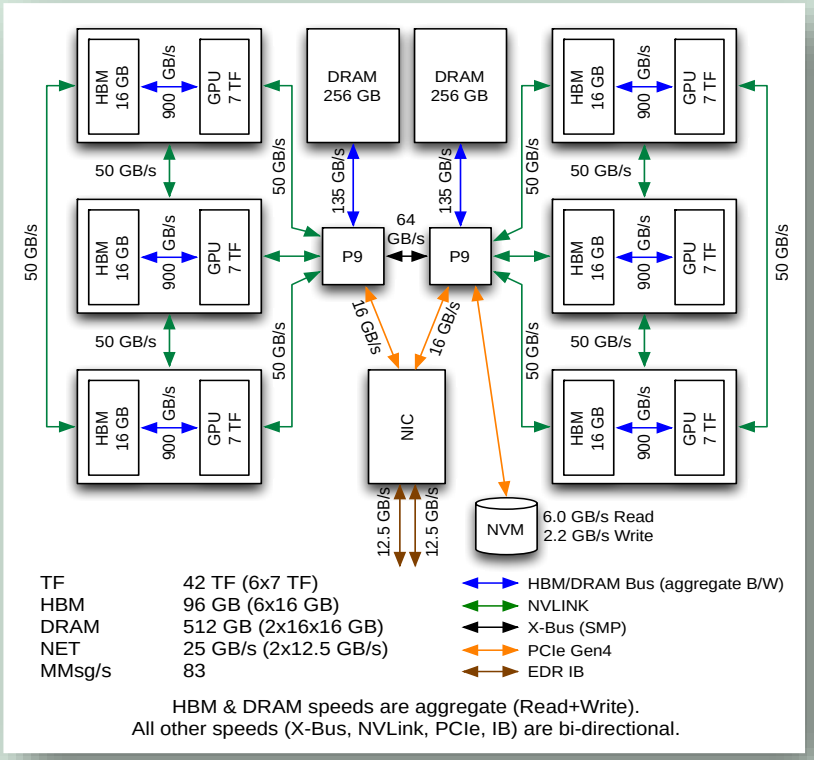
AMD Radeon Instinct GPUs.
> 1.5 EF

Intel® Xeon® Scalable processor accelerated by Intel's Xe compute architecture.
>=1 EF

Leonardo: NVIDAI Ampere architecture-based GPUs and NVIDIA® Mellanox® HDR 200Gb/s InfiniBand networking.
10 EF of AI Performance

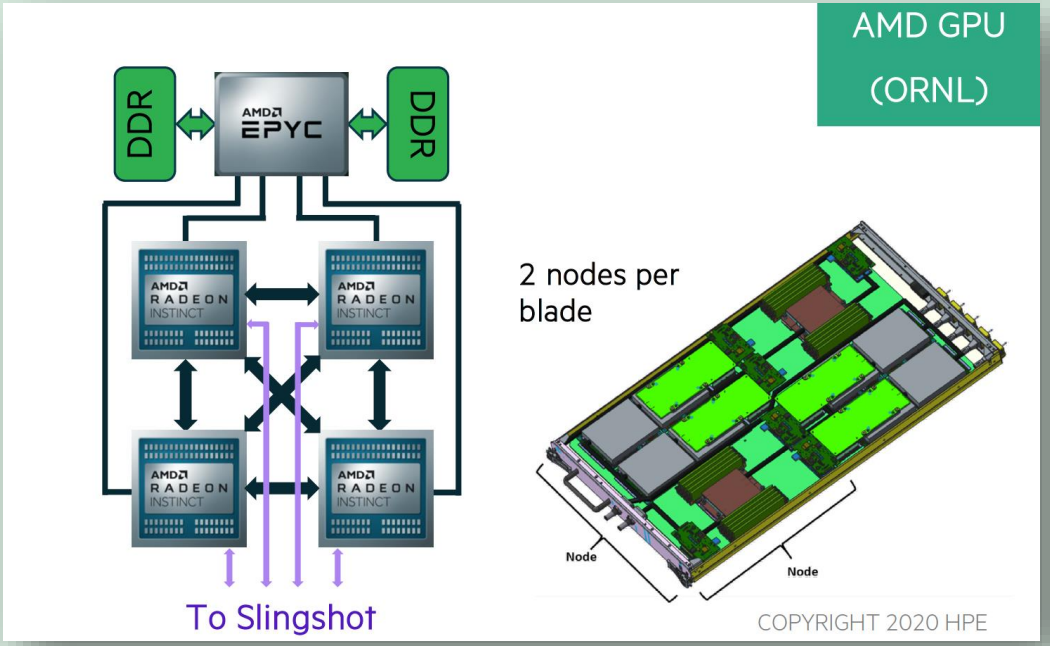
OLCF Supercomputing Platforms

- **OLCF Summit supercomputer**: an IBM AC922 system consisting of 4608 large nodes each with six NVIDIA Volta V100 GPUs and two POWER9 CPU sockets providing 42 usable cores per node.



200 PF → **> 1.5 EF**

- **Upcoming Frontier**: Single AMD EPYC CPU with 4 AMD Radeon Instinct GPUs with AMD Infinity Fabric links and coherent memory between them within the node. The nodes are connected with a Slingshot interconnect network port for every GPU (100 GB/s aggregate network bandwidth.)



Portability for Migration

Migration Path from Summit to Frontier

Summit	Frontier	Comments
CUDA C/C++	HIP C/C++	HIP provides tools to help port existing CUDA codes to the HIP layer. HIP is not intended to be a drop-in replacement for CUDA, and developers should expect to do some manual coding and performance tuning work to complete the port.
OpenACC	OpenMP (offload)	OpenACC codes can be migrated to OpenMP (offload) for Frontier. *Direct support for OpenACC on Frontier is still under discussion.*
OpenMP (offload)	OpenMP (offload)	Virtually the same on Summit and Frontier
FORTRAN w/CUDA C/C++	FORTRAN w/HIP C/C++	As with CUDA, this will require interfaces to the C/C++ API calls
CUDA FORTRAN	FORTRAN w/HIP C/C++	As with CUDA, this will require interfaces to the C/C++ API calls

CUDA

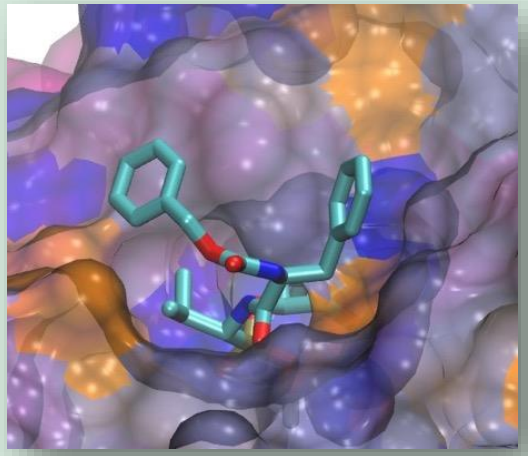
Kokkos

OpenMP offload

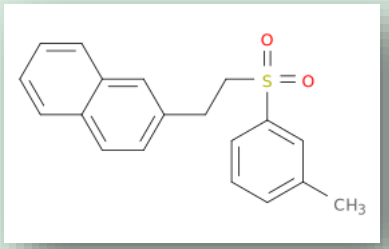
HIP

https://www.olcf.ornl.gov/frontier/

miniMDock



rcsb.org: 7cpa



Enamine Database:
https://enamine.net

- **How it was born:**
  - **AutoDock-GPU**: The COVID-19 pandemic has fueled a flurry of activity in computational drug discovery, including the use of supercomputers and GPU acceleration for massive virtual screens for therapeutics.
  - **miniAutoDock:** Performance portability evaluation - especially relevant as facilities transition from petascale systems and prepare for upcoming exascale system.
  - **miniMDock:** ECP proxy app, https://proxyapps.exascaleproject.org/app/minimdock
  - https://www.osti.gov/doecode/biblio/70713

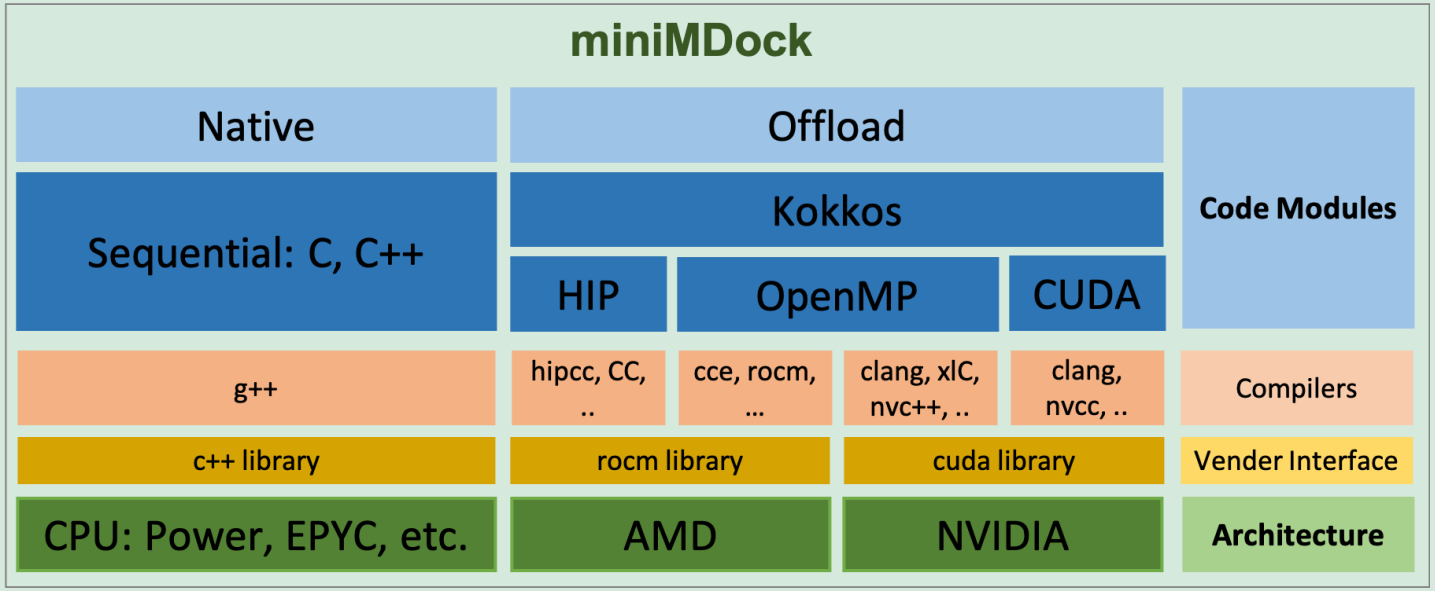
AutoDock-GPU

- A set of protein and ligands
- Choosable local searching methods: ADADELTA, Solis-wets, etc.
- OpenCL and CUDA versions, OpenMP offload

miniMDock

- A single protein and ligand
- Solis-Wets local searching method.
- **CUDA**, **HIP**, **Kokkos**, **OpenMP offload**, C++ Std-par

miniMDock: Design and Structure

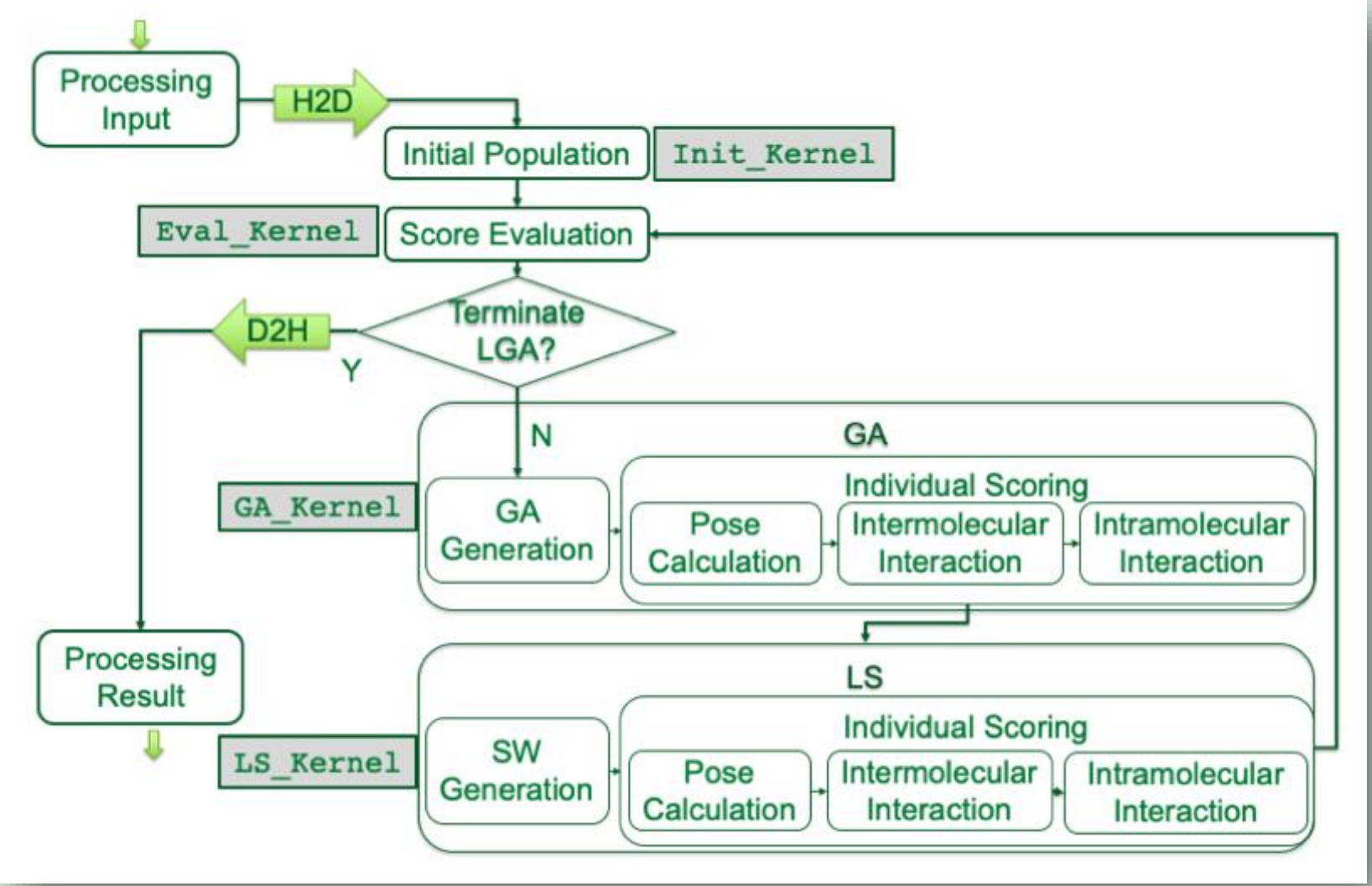


miniMDock: Software stack

- Preloaded ligand-protein grids
  - Common CPU based code
- Docking on GPU
  - Package of device codes
  - CUDA, HIP kernels
  - Kokkos framework
  - OpenMP offload
  - …

- Targeting systems
  - NVIDIA GPU – Evaluating on Summit
  - AMD GPU – Evaluating on the Cray Frontier Center of Excellence and SPOCK
    - Frontier
  - Intel GPU
- The CUDA version was heavily optimized, including hardware-level optimizations and warp-level primitives.

miniMDock: Algorithm



- Initial and final data on CPU
- Computing on GPU
- Four GPU Kernels
- Lamarckian GA
- Random Optimizer
  – Solis-Wets
- Iterative method
- Heavy worker - LS-Kernel
- Multiple runs
- Best scoring pose

Translating CUDA to HIP

- **Porting Highly Optimized Kernels**:
  - Porting low level, architecture specific warp-level CUDA optimizations to a different architecture is a challenging task due to the very nature of such optimizations.
  - NVIDIA GPU with warp size 32 to AMD-GCN GPU with wavefront size 64
- **Available/Evolving features:**
  - Differences in low level intrinsics and details, availability and semantics of shuffle operations.
  - The available warp vote (`_any`) and shuffle (`_shfl`) functions in AMD cannot be directly mapped to functions in new CUDA versions because they have been deprecated in CUDA 9.0 for all NVIDIA devices.
- **Need of Two Versions:**
  - Portable code with optimized low-level kernels will necessitate the development and maintenance of two versions of the kernels, even though HIP can provide a functionally portable implementation that can run on both NVIDIA and AMD GPUs systems.

Translating CUDA to HIP - it requires manual translation

- Architecture specific optimization

```
#define REDUCEINTEGERSUM(value, pAccumulator) \
if (threadIdx.x == 0) { *pAccumulator = 0; } \
__threadfence(); \
__syncthreads(); \
if (__any_sync(0xffffffff, value != 0)) { \
    uint32_t tgx = threadIdx.x & cData.warpmask; \
    value +=__shfl_sync(0xffffffff, value, tgx^1);\
    value +=__shfl_sync(0xffffffff, value, tgx^2);\
    value +=__shfl_sync(0xffffffff, value, tgx^4);\
    value +=__shfl_sync(0xffffffff, value, tgx^8);\
    value +=__shfl_sync(0xffffffff, value, tgx^16);\
    if (tgx == 0) { \
    atomicAdd(pAccumulator, value); } \
} \
__threadfence(); \
__syncthreads(); \
value = *pAccumulator; \
__syncthreads();
```

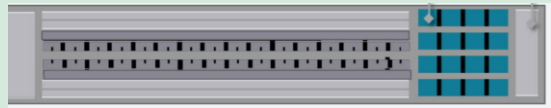
CUDA warp-level reduction

```
#define REDUCEINTEGERSUM(value, pAccumulator) \
if (hipThreadIdx_x == 0) { *pAccumulator = 0; } \
__threadfence(); \
__syncthreads(); \
if (__any(value != 0)) { \
    uint32_t tgx = hipThreadIdx_x & cData.warpmask;
    \
    value += __shfl( value, tgx ^ 1);\
    value += __shfl( value, tgx ^ 2);\
    value += __shfl( value, tgx ^ 4);\
    value += __shfl( value, tgx ^ 8);\
    value += __shfl( value, tgx ^ 16);\
    value += __shfl( value, tgx ^ 32);\
    if (tgx == 0) { \
        atomicAdd(pAccumulator, value); } \
} \
 __threadfence(); \
__syncthreads(); \
value = *pAccumulator; \
__syncthreads();
```

HIP wavefront-level reduction

warp size: 32
warp mask: 31 (11111)
warp bits: 5

wavefront size: 64
wavefront mask: 63(111111)
wavefront bits: 6



Four 16-wide SIMD vectors

Translating CUDA to OpenMP Target Offload -challenges

- Hierarchy of parallel constructs

```
__global__ void gpu_kernel( parameters ... ){
  __shared__ float3 A[N];
  float  x = device_function( A, ... );
  compute( ... );
  if (threadIdx.x == 0) {
      //Work for the master thread
  }
}
void use_gpu_kernel(uint32_t nblocks, uint32_t threads_pblock
    , parameters ...){
    gpu_kernel<<<nblocks, threads_pblock>>>( parameters ...);
}
```

CUDA Kernel

1. Define global gpu kernel function
2. Choose grid size, `nblocks` and block size, `threads_pblock`
3. Define shared variables explicitly
4. Assign work for master thread (`threadIdx == 0`) explicitly
5. API specific data types, `float3`

```
void gpu_kernel( uint32_t nteams, uint32_t threads_pteam,
    parameters ... ){
#pragma omp target teams distribute num_teams(nteams)
    thread_limit(threads_pteam)
  for (int i = 0; i < nit_atteam; i++){
      float3_struct A[N];
#pragma omp parallel for
      for (int j = 0; j < work_pteam; j++){
          float  x = device_function( A, ... );
          compute( ... );
      }// end of a team
      { ... } //Work for the master thread
  }// end of teams
}
```

OpenMP target offload Kernel

1. Host function that utilizes OpenMP offloading
2. Set upper limit for league size, `nteams` and team size `threads_pteam`
3. Define shared variable inside the league
4. Define **for loop** explicitly
   - For teams – a set of single threaded team
   - For a team
5. User defined data type, `float3_struct`

Translating CUDA to OpenMP Target Offload -challenges

- Thread synchronization

```
__global__ void gpu_kernel( parameters ... ){
  __shared__ float3 A[N];
  float  x = device_function( A, ... );
  __syncthreads();
  compute( ... );
  if (threadIdx.x == 0) {
      //Work for the master thread
  }
}
void use_gpu_kernel(uint32_t nblocks, uint32_t threads_pblock
    , parameters ...){
    gpu_kernel<<<nblocks, threads_pblock>>>( parameters ...);
}
```

CUDA Kernel

```
void gpu_kernel( uint32_t nteams, uint32_t threads_pteam,
    parameters ... ){
#pragma omp target teams distribute num_teams(nteams)
    thread_limit(threads_pteam)
  for (int i = 0; i < nit_atteam; i++){
      float3_struct A[N];
#pragma omp parallel for
      for (int j = 0; j < work_pteam; j++){
          float  x = device_function( A, ... );
      }// end of a team -- implicit barrier
#pragma omp parallel for
      for (int j = 0; j < work_pteam; j++){
          compute( ... );
      }// end of a team
      { ... } //Work for the master thread
  }// end of teams
}
```

OpenMP target offload Kernel

- CUDA - Explicit thread synchronization
- `#pragma omp barrier` inside a team – doesn't work
- Generate unique team of threads
  - Implicit barrier at the end of each team

What can we do if `device_function` has `__synchthreads()` ?

Generating unique team of threads inside the target function – doesn't work

Translating CUDA to OpenMP Target Offload -challenges

- Thread synchronization

  - Decompose the device function into a set of target functions – work for a single thread
  - In the main kernel, Generate team for each and call those functions – redundant codes

```
__device__ void device_function( energy, ... ){
    for (int atom_id = threadIdx.x; atom_id < natoms; atom_id
    += blockDim.x) {
        get_atompos( atom_id, ... );
    }
    __syncthreads();
    for (int rot  = threadIdx.x; rot  < rot_length; rot +=
    blockDim.x){
        rotate_atoms(rot, ...);
       __syncthreads();
    }
    for (int atom_id = threadIdx.x; atom_id < natoms; atom_id
    += blockDim.x){
        energy += calc_energy( atom_id, ...);
    }
    // warp-level reduction to calculate energy
    REDUCEFLOATSUM(energy, ...)
}
```

CUDA device function

```
    float energy = 0.0f;
#pragma omp parallel for
    for (int atom_id = 0; atom_id < natoms; atom_id++) {
        get_atompos( atom_id, ... );
    }
    for(int rotcyc=0; rotcyc < nrotcyc; rotcyc++){
        int start = rot*work_pteam;
        int end = start +work_pteam;
        if ( end > rot_length ) end = rot_length;
#pragma omp parallel for
        for (int rot = start; rot  < end; rotr++){
            rotate_atoms(rot ...);
        }// end for rotations; a rotation cycle
    }// end for rotation cycles
    // team-level reduction
#pragma omp parallel for reduction(+:energy)
    for (int atom_id = 0; atom_id < natoms; atom_id++){
        energy += calc_erenergy( atom_id, ... );
    }
```

Code segment that deals target functions

- Major issue with porting in a reverse direction
  - Need to understand deeply – e.g. rotate atoms, `nrotcyc x threads_pblock`
- warp-level reduction vs team-level reduction
  - May loss the performance

Translating CUDA to OpenMP Target Offload –change strategy

- Using teams---parallel

```
__global__ void gpu_kernel( parameters ... ){
  __shared__ float3 A[N];
  float  x = device_function( A, ... );
  __syncthreads();
  compute( ... );
  if (threadIdx.x == 0) {
      //Work for the master thread
  }
}
void use_gpu_kernel(uint32_t nblocks, uint32_t threads_pblock
    , parameters ...){
    gpu_kernel<<<nblocks, threads_pblock>>>( parameters ...);
}
```

CUDA Kernel

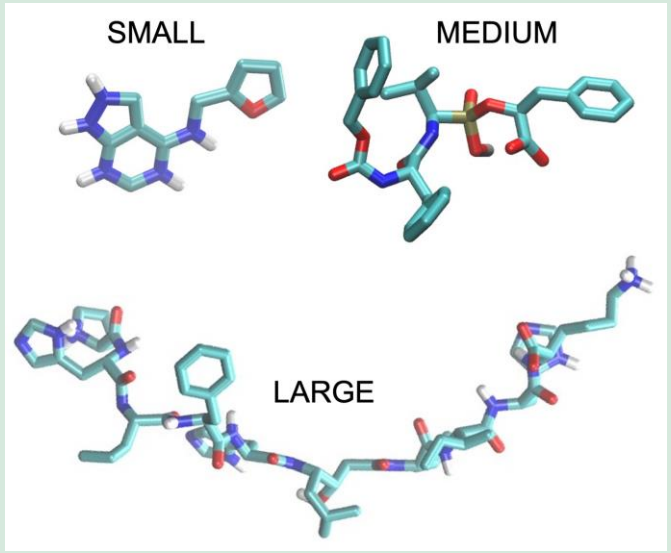
- Direct conversion from CUDA
- More portable – fundamental features
- More hardware-level programming style.
- Manual work sharing for threads
- Explicit thread synchronization
  - `#pragma omp barrier`
  - Usable inside the device function

```
void gpu_kernel( uint32_t nteams, uint32_t
    threads_pteam, parameters ... ){
#pragma omp target teams num_teams(nteams)
    thread_limit(threads_pteam)
    {
    float3_struct A[N];  // shared data
  #pragma omp parallel
   {
       const int threadIdx = omp_get_thread_num();
       const int blockDim = omp_get_num_threads();
       const int blockIdx = omp_get_team_num();
       const int gridDim = omp_get_num_teams();
       for (uint32_t idx = blockIdx; idx < nteams;
    idx+=gridDim) {  // for teams
           float  x = device_function( A, ... );
             #pragma omp barrier
             compute( ... );
             f (threadIdx == 0) {
               //Work for the master thread
               }
          }
      } // end of parallel region
  } // end of teams region
}
```

OpenMP target offload Kernel

Performance Evaluation

Test case



	SMALL	MEDIUM	LARGE
Ligand	nsc1620	7cpa	3er5
Number of atoms	21	43	108
Number of rotatable bonds	2	15	31

Heterogeneous Systems

Summit

CPU: IBM Power9
GPU: NVIDIA Tesla V100
Connection: NVLINK with 25 GB/s transfer rate

Compilers:
NVHPC 21.11
LLVM 14.0 and 15.0

Spock

CPU: AMD EPYC 7662
GPU: AMD MI100
Connection: PCIe Gen4 with 32 GB/s transfer rate

Compilers:
Rocm 4.5
AOMP 14.0.1
CCE 12.0.1

Docking parameters

Maximum number of energy evaluations: 2500000
Maximum umber of generations: 27000
Population size: 150
nrun : 10
Maximum number of iterations: 300

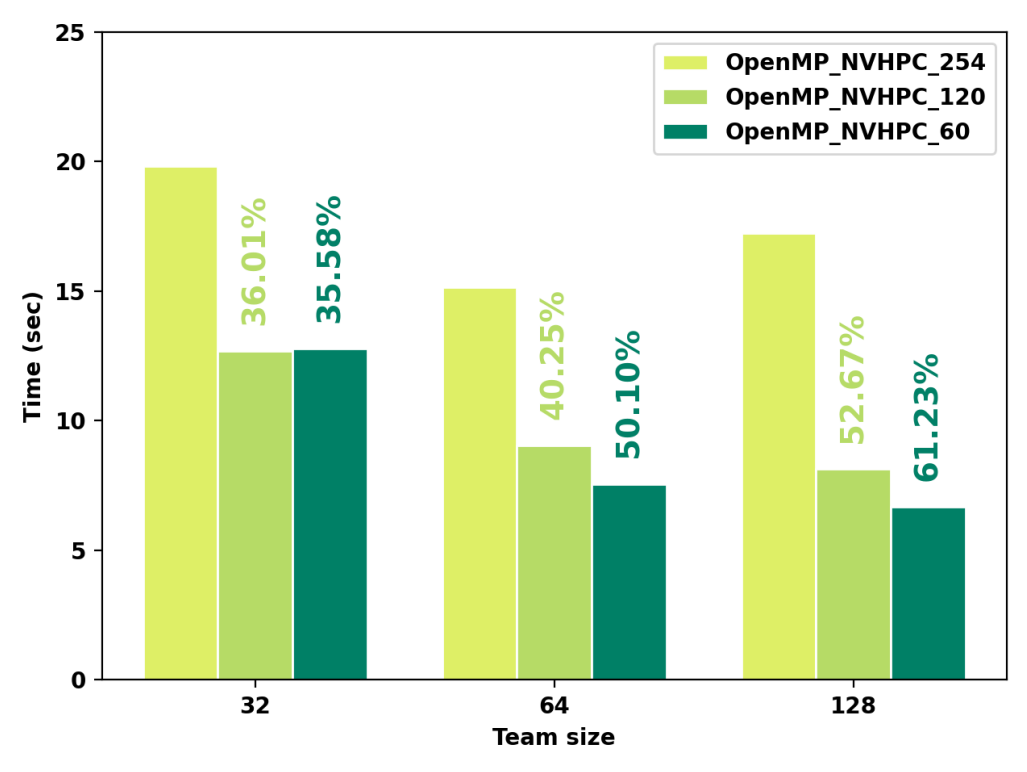
Performance Enhancement by tunning parameters

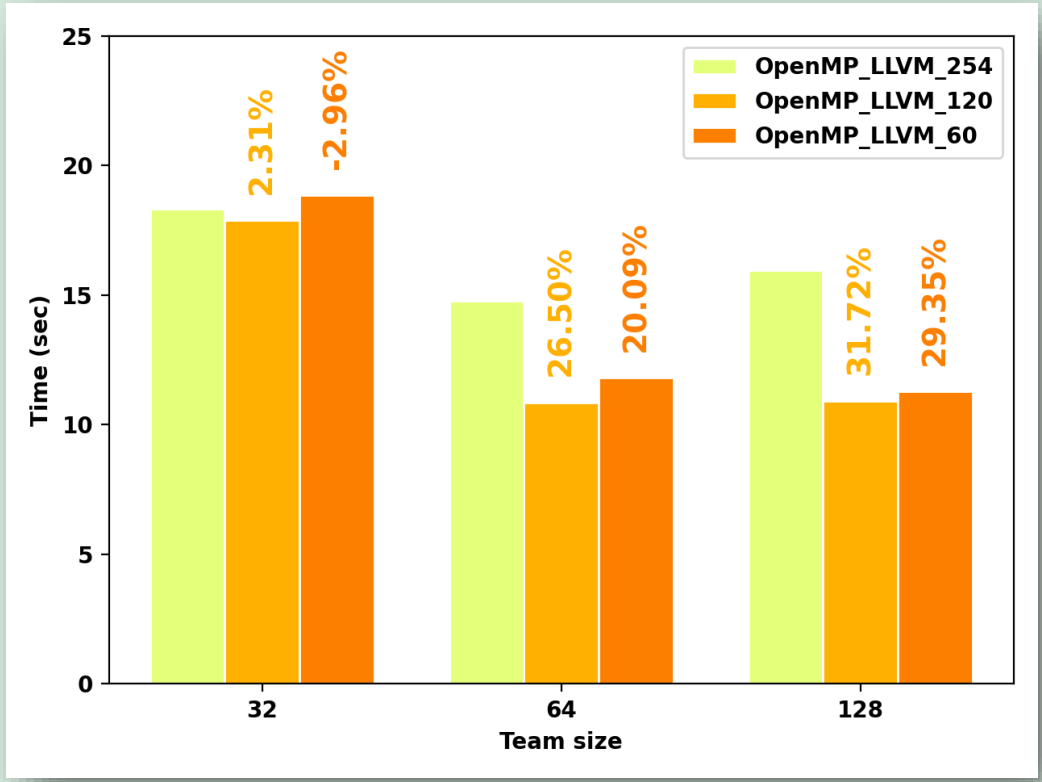
- Tuning maximum thread register count

LLVM clang: `-Xcuda-ptxas --maxrregcount=120`

NVIDIA nvhpc: `-gpu=maxrregcount:60`

- Tuning team size: `thread_limit()`

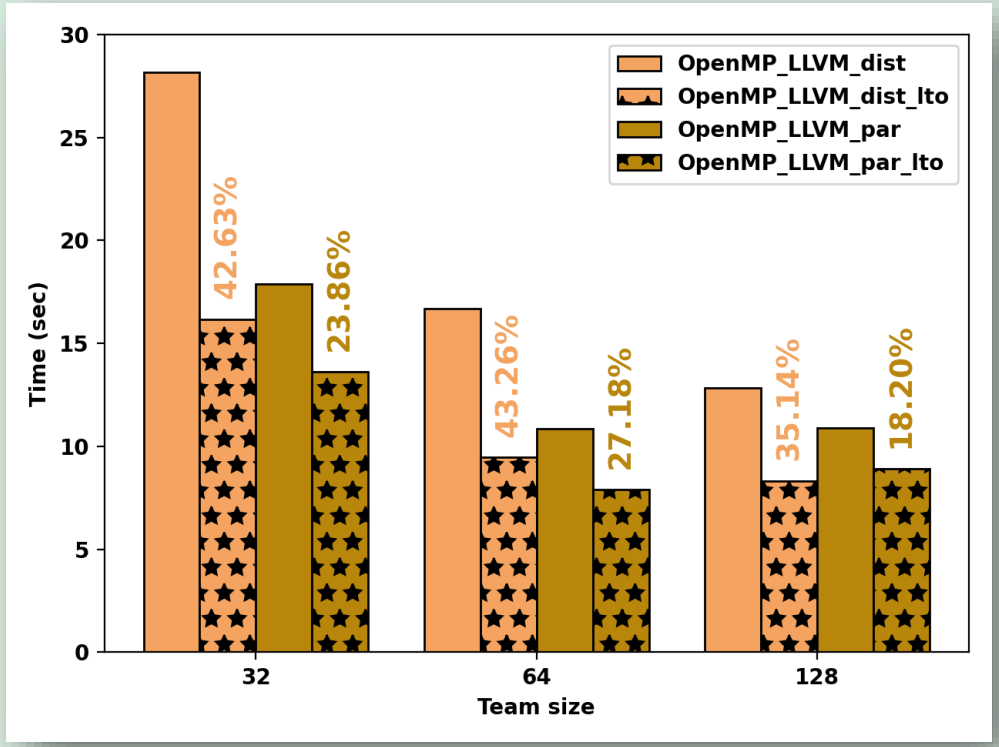




- For default thread register count, team size 64 shows the best performance for both compilers
- Optimum value for maximum thread register count is 60 for NVHPC and 120 for LLVM

Performance Enhancement by tunning parameters

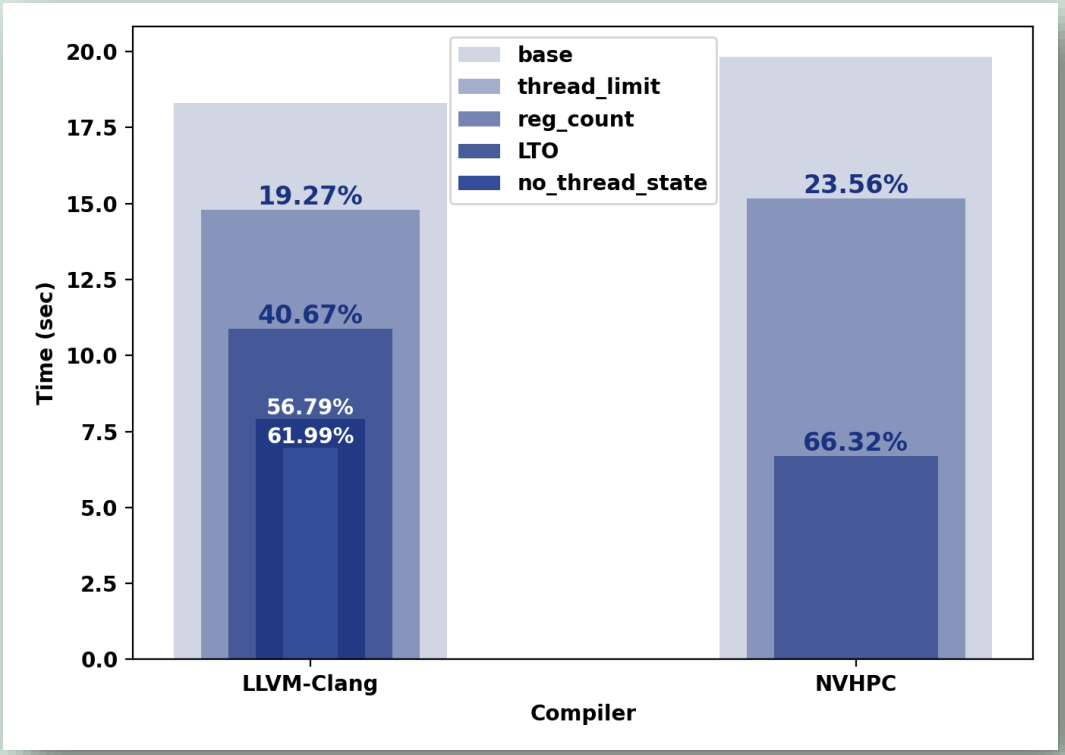
- Link Time Optimization for LLVM



LLVM 15 and cuda/11.4.2

-fopenmp-new-driver -foffload-lto

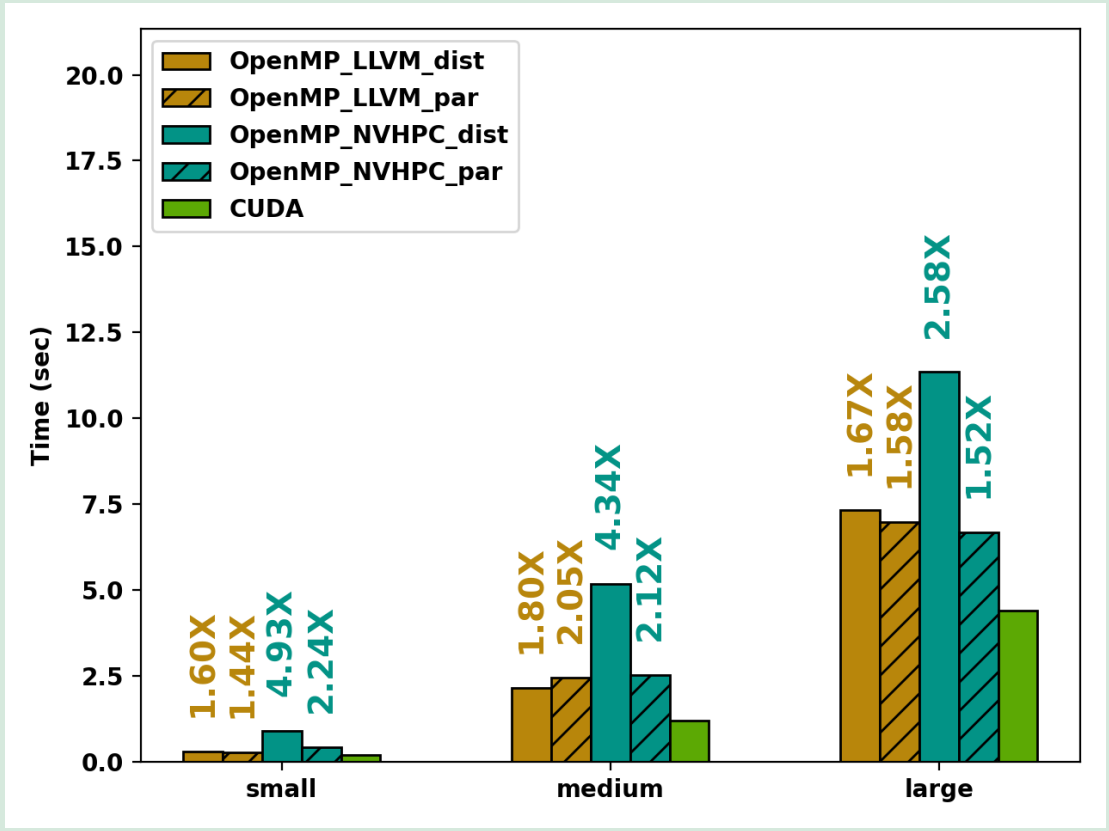
- Parameter tunning effects



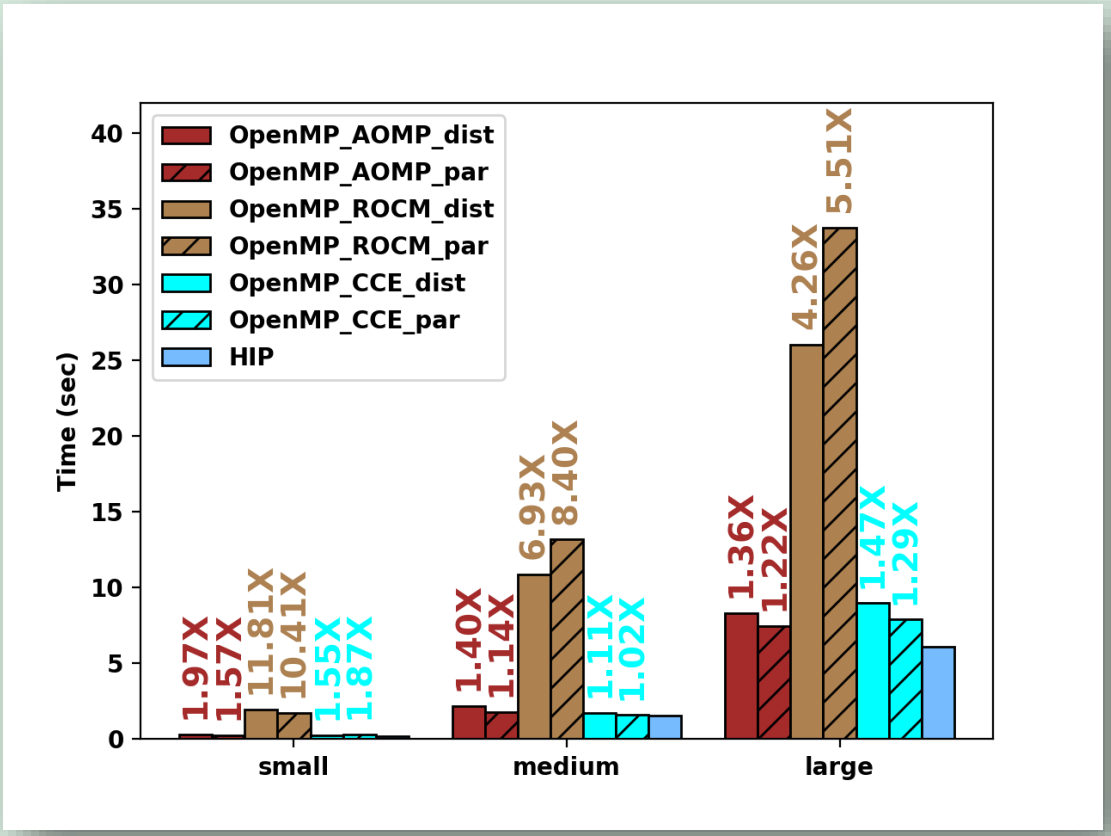
-fopenmp-assume-no-thread-state

- LTO: More sensitivity for the approach 1 (using distributive) ~ 40% improvement
- Step by step improvement for the second approach ~ 62% improvement

Performance Evaluation OpenMP Target Offload on AMD and NVIDIA GPUs



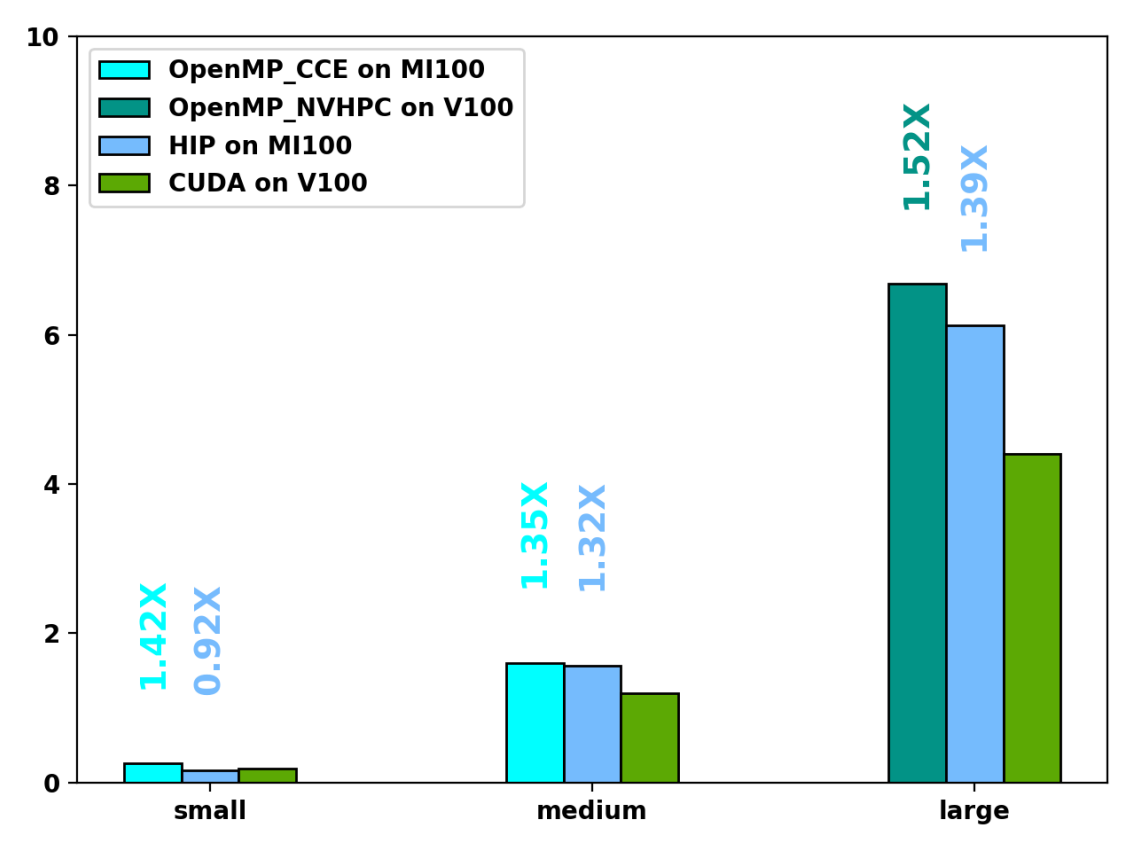
On NVIDIA GPUs - V100



On AMD GPUs – MI100

- The 2nd approach gives better performance in general, except for ROCM (4.5) on Spock
- The NVHPC 21.11 is better than LLVM-clang 14/15 on Summit
- OpenMP-CCE 12.0.1 and AOMP 14.0.1 show the good performance on Spock

Performance Evaluation cross-platform



- Best performance across platforms
- HIP gets better performance than CUDA for the small input, otherwise CUDA gets better.

- OpenMP-LLVM on the NVIDIA GPU and OpenMP-CCE on the AMD GPU provide identical performance for the small input.
- OpenMP-CCE provides best performance for the medium input.
- OpenMP-NVHPC and OpenMP-AOMP give better performance for the large input.
- OpenMP-NVHPC outperforms for the large input. (?...)

Conclusion

- Vender specific APIs give better performance for molecular docking tools on their own platforms – less portability.
- Directive based programming models are portable, to make sure those are performance portable we need to put some efforts.
  - Choose appropriate strategy
    - Productive based
    - Performance based
  - Tunning the number of teams and team size
  - Choose appropriate compiler
    - Vender specific compilers give better performance – again not portable compilers.
    - LLVM-Clang is a portable compiler and gives acceptable performance.
    - Evolving and advancing compiler features …
  - Tuning compiler parameters

Acknowledgements

- NVIDIA staff, especially Oscar Hernadz and Scott LeGrand
- Scripps Research
- Joseph Huber
- ECP-SOLLVE team
- ORNL OLCF staff
- Biological Sciences staff

Thank You