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
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.

  - JAGUAR: No. 1 in 2009, 2010
  - TITAN: No. 1 in 2012
  - SUMMIT: No. 1 in 2018
  - FRONTIER: Debuting in 2022

• **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:** NVIDIA 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.

- **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

<table>
<thead>
<tr>
<th>Summit</th>
<th>Frontier</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA C/C++</td>
<td>HIP C/C++</td>
<td>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.</td>
</tr>
<tr>
<td>OpenACC</td>
<td>OpenMP (offload)</td>
<td>OpenACC codes can be migrated to OpenMP (offload) for Frontier. <em>Direct support for OpenACC on Frontier is still under discussion.</em></td>
</tr>
<tr>
<td>OpenMP (offload)</td>
<td>OpenMP (offload)</td>
<td>Virtually the same on Summit and Frontier</td>
</tr>
<tr>
<td>FORTRAN w/CUDA C/C++</td>
<td>FORTRAN w/HIP C/C++</td>
<td>As with CUDA, this will require interfaces to the C/C++ API calls</td>
</tr>
<tr>
<td>CUDA FORTRAN</td>
<td>FORTRAN w/HIP C/C++</td>
<td>As with CUDA, this will require interfaces to the C/C++ API calls</td>
</tr>
</tbody>
</table>

[https://www.olcf.ornl.gov/frontier/](https://www.olcf.ornl.gov/frontier/)
miniM Dock

• 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.
  – miniM Dock: ECP proxy app,
    https://proxyapps.exascaleproject.org/app/minimdock

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

miniM Dock
  • A single protein and ligand
  • Solis-Wets local searching method.
  • CUDA, HIP, Kokkos, OpenMP offload, C++ Std-par
miniMDock: Design and Structure

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

miniMDock: Software stack

- 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

CUDA warp-level reduction

```c
#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);\n  value += __shfl_sync(0xffffffff, value, tgx^2);\n  value += __shfl_sync(0xffffffff, value, tgx^4);\n  value += __shfl_sync(0xffffffff, value, tgx^8);\n  value += __shfl_sync(0xffffffff, value, tgx^16);\n  if (tgx == 0) { \n    atomicAdd(pAccumulator, value); } \n} \n__threadfence(); \n__syncthreads(); \nvalue = *pAccumulator; \n__syncthreads();
```

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

HIP wavefront-level reduction

```c
#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);\n  value += __shfl(value, tgx ^ 2);\n  value += __shfl(value, tgx ^ 4);\n  value += __shfl(value, tgx ^ 8);\n  value += __shfl(value, tgx ^ 16);\n  value += __shfl(value, tgx ^ 32);\n  if (tgx == 0) { \n    atomicAdd(pAccumulator, value); } \n} \n__threadfence(); \n__syncthreads(); \nvalue = *pAccumulator; \n__syncthreads();
```

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

**CUDA Kernel**

1. Define global gpu kernel function
2. Choose grid size, nbblocks 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

**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

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 `__syncthreads()`?

CUDA Kernel

```c
__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
    }
    use_gpu_kernel(uint32_t nblocks, uint32_t threads_pblock, parameters ...){
        gpu_kernel<<<nblocks, threads_pblock>>>( parameters ...);
    }
}
```

OpenMP target offload Kernel

```c
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
```
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

CUDA device function

```c
__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, ... )
}
```

Code segment that deals target functions

```c
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_p_ateam;
  int end = start +work_p_ateam;
  for (int end > rot_length ) end = rot_length;
  #pragma omp parallel for
  for (int rot = start; rot < end; rot++){
    rotate_atoms(rot ...);
  }
  // end for rotations; a rotation cycle
}
// end for rotation cycles
```

- 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

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

OpenMP target offload Kernel

```c
void gpu_kernel(void (*gpu_kernel)(parameters ...)){
  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 ...);
}
```

```c
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 = blockDim; idx < nteams; idx += gridDim) {
        // for teams
        float x = device_function(A, ...);
        #pragma omp barrier
        compute(...);
        if (threadIdx == 0) {
          // Work for the master thread
        }
      }
    }
  }
}
```
Performance Evaluation

Test case

<table>
<thead>
<tr>
<th>Ligand</th>
<th>SMALL</th>
<th>MEDIUM</th>
<th>LARGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>nsc1620</td>
<td>21</td>
<td>43</td>
<td>108</td>
</tr>
<tr>
<td>7cpa</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3er5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Number of atoms: SMALL 21, MEDIUM 43, LARGE 108
Number of rotatable bonds: SMALL 2, MEDIUM 15, LARGE 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

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

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
Performance Enhancement by tuning 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 tuning parameters

- Link Time Optimization for LLVM

- Parameter tuning effects

LLVM 15 and cuda/11.4.2

-\texttt{-fopenmp-new-driver -foffload-lto}

-\texttt{-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

- The 2\textsuperscript{nd} 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

On AMD GPUs – MI100
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