Evaluating the Performance of a Portable Version of the HPGMG Benchmark for Accelerators

Christopher Daley, Hadia Ahmed, Sam Williams, Nicholas Wright (LBNL/NERSC), Mat Colgrove (NVIDIA)
P3HPC 2020 – Sep 1
This presentation will evaluate OpenMP target offload and OpenACC Performance, Portability and Productivity (PPP) for the HPGMG benchmark.

HPGMG is a Finite Volume Geometric Multigrid benchmark.

OpenMP target offload/OpenACC PPP will be shown for:
1. A base version of HPGMG ported from a CUDA Managed Memory version of HPGMG.
2. A new version of HPGMG using explicit data movement instead of Managed Memory.
Multigrid methods and HPGMG overview

Multigrid method use a hierarchy of levels to solve elliptic PDEs:

- Levels consist of $2^3$, $4^3$, $8^3$, … grid points (full Multigrid configuration)
- HPGMG divides the level data into blocks and distributes the blocks across MPI ranks
- HPGMG allocates large data buffers per level: block pointers are used to read/write at various offsets in these large data buffers
 CODE VERSION #1: A MANAGED MEMORY IMPLEMENTATION OF HPGMG

- HPGMG-CUDA is an NVIDIA fork of HPGMG ([https://bitbucket.org/nsakharnykh/hpgmg-cuda](https://bitbucket.org/nsakharnykh/hpgmg-cuda))
  - Depends on Managed Memory
  - Shallow copies a level data structure to the GPU

- We ported HPGMG-CUDA to OpenMP target offload and OpenACC using the following approach
  - Copy the body of the CUDA kernels into new functions
  - Replace CUDA thread indexing (blockIdx, threadIdx) with work-shared OpenMP target offload / OpenACC loops
  - Retain cudaMemcpyManaged CUDA runtime API calls
<table>
<thead>
<tr>
<th></th>
<th>Cori-GPU</th>
<th>Summit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node architecture</td>
<td>Cray CS-Storm 500NX</td>
<td>IBM AC922</td>
</tr>
<tr>
<td>Node CPUs</td>
<td>2 x Intel Skylake</td>
<td>2 x IBM Power 9</td>
</tr>
<tr>
<td>Available cores per CPU</td>
<td>20 @ 2.40 GHz</td>
<td>21 @ 3.07 GHz</td>
</tr>
<tr>
<td>Node GPUs</td>
<td>8 x 16 GB NVIDIA V100</td>
<td>6 x 16 GB NVIDIA V100</td>
</tr>
<tr>
<td>CPU-GPU interconnect</td>
<td>PCIe 3.0 switch</td>
<td>NVLink 2.0</td>
</tr>
<tr>
<td>Compiler</td>
<td>GPU offload</td>
<td>Cori-GPU version</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>GCC + NVCC</td>
<td>CUDA</td>
<td>7.3.0 + 10.1.244</td>
</tr>
<tr>
<td>NVIDIA/PGI</td>
<td>OpenACC</td>
<td>20.4</td>
</tr>
<tr>
<td>Cray CCE</td>
<td>OpenMP</td>
<td>9.1.0 (LLVM version)</td>
</tr>
<tr>
<td>IBM XL</td>
<td>OpenMP</td>
<td>-</td>
</tr>
<tr>
<td>LLVM/Clang</td>
<td>OpenMP</td>
<td>11.0.0-git (#17d8334)</td>
</tr>
</tbody>
</table>
HPGMG configuration used

- We used the Top-500 HPGMG configuration: 4\textsuperscript{th} order accurate, GSRB smoother, and BiCGStab bottom solver
- Grid spacing = 1/512: creates 9 levels from 2\textsuperscript{3} to 512\textsuperscript{3} grid points
  - Maximum box size = 32\textsuperscript{3}
- Memory footprint \textasciitilde 38 GiB
- CPU-only configuration run on 1 CPU socket: 1 MPI rank per core
- GPU configuration run on 1 CPU socket and 3 GPUs: 1 MPI rank per GPU
Managed Memory performance on Summit: 1 Power 9 CPU and 3 Volta GPUs

NVCC CUDA: 16x faster than the MPI-only configuration on a single CPU (21c)

GPU offload using directives can be competitive with CUDA:
  PGI OpenACC: 0.89x
  XL OpenMP: 0.70x

Clang performed poorly because of OpenMP runtime overheads (~80% of total runtime spent in cuMemAlloc and cuMemFree)
Managed Memory performance on Cori-GPU: 1 Skylake CPU and 3 Volta GPUs

NVCC CUDA and PGI OpenACC are 2.6x and 3.1x slower on Cori-GPU than Summit!

3 reasons for the slowdown:
• More page faults
• More data movement between CPU and GPU
• Lower bandwidth transfers between CPU and GPU

CCE OpenMP performed poorly because –O0 compilation used for correctness
Addressing Managed Memory Performance Gaps

- We don’t understand why there were more page faults and data moved on x86
  - The inefficient data transfer could potentially be addressed by prefetching the data using cudaMemPrefetchAsync() [Future work]
- We do understand that LLVM/Clang performed poorly on x86 and Power because of significant OpenMP runtime memory management
  - There is a repeated transfer of ~1 KB to shallow copy the level data structure for each OpenMP target region
  - Shilei Tian (Stony Brook University) has enhanced the LLVM OpenMP runtime to buffer target memory instead of returning it to the device: commit #0289696 (08/19/2020)
LLVM Memory Manager significantly improved Clang performance on Cori-GPU

Nvprof shows significantly less time spent in memory management API calls

Original:
34,139 calls to cuMemFree (38.4% time)
34,139 calls to cuMemAlloc (35.5% time)

LLVM Memory Manager Patch:
0 calls to cuMemFree (0.0% time)
5 calls to cuMemAlloc (0.0% time)
Modifying the directives further improved Clang performance on Cori-GPU

Achieved 0.89x of CUDA performance!

Required convoluted code changes to remove user code between “target” and “parallel” directives to use a faster code generation scheme (SPMD): https://clang.llvm.org/docs/OpenMPSupport.html#directives-execution-modes

Johannes Doerfert (ANL) has started compiler development work to save users from manually doing this transformation: https://reviews.llvm.org/D59319
void smooth(level_type level, ...)
{
#pragma omp target teams distribute map(to:level)
  for (int blk=0; blk < level.num_my_blocks; blk++) {

void smooth(level_type *level, ...)
{
#pragma omp target teams distribute map(to:level[0])
  for (int blk=0; blk < level->num_my_blocks; blk++) {

The Managed Memory version does a shallow copy of “level” to the device for each target region

The explicit data management version creates “level” on the device at program start and then passes a pointer to “level” for each target region

Thanks to Mat Colgrove for the initial OpenACC implementation
The “level” data structure is complicated – ~250 lines of code to create it on the device

```c
typedef struct {
    struct {
        double * ptr;
        // + other variables
    } read, write;
} blockCopy_type;

typedef struct {
    double ** send_buffers;
    double ** recv_buffers;
    blockCopy_type * blocks[3];
    // + other variables
} communicator_type;

typedef struct {
    double ** vectors;
    communicator_type exchange_ghosts[STENCIL_MAX_SHAPES];
    communicator_type restriction[4];
    communicator_type interpolation; // + other variables
} level_type;
```

level_type is a nested data structure containing many pointers and double pointers

Block pointers (see blockCopy_type “ptr”) may be NULL or may point to communicator_type “send_buffers” or “recv_buffers”
Use “target enter data” to point the block pointers to device data buffers

```c
for (shape=0; shape<STENCIL_MAX_SHAPES; shape++) {
    for (block=0; block<3; ++block) {
        for (b=0; b<level->exchange_ghosts[shape].num_blocks[block]; ++b) {
            #pragma omp target enter data
            map(alloc:level->exchange_ghosts[shape].blocks[block][b].read.ptr[:0])
        }
    }
}
```

Update device pointer using zero length array section
It worked but exposed issues in multiple compilers

- The PGI compiler successfully executed the OpenACC version
- Only LLVM/Clang successfully executed the OpenMP version of the application
  - Runtime errors in XL and CCE compilers
- LLVM/Clang performance was worse than the unoptimized Managed Memory version of the code
  - A profile showed that a huge amount of time was spent in a “target update from” directive used to copy data from GPU to CPU
  - Most of the time was spent in the OpenMP runtime rather than moving data!
Addressing LLVM OpenMP runtime overhead

- OpenMP runtimes use a present table to maintain the association between host and device pointers.
- We added ~100K entries to the present table when updating HPGMG block pointers (using “target enter data” directive).
  - This caused high lookup time in “target update from” directive (https://bugs.llvm.org/show_bug.cgi?id=46107)
- We tested 2 ways to minimize the present table:
  1. Don’t update a device pointer if the host block pointer is NULL.
  2. Update the block pointers on the device in a OpenMP target region: avoids adding an entry to the present table.
Incremental optimizations to improve LLVM/Clang performance on Summit

Present table optimizations improved performance by 24.2x

CUDA-aware MPI and SPMD code transformations improved performance by another 2.4x
Code version #3: Explicit data management using OpenMP runtime API

- Used separate host and device “level” variables
- Data directives replaced by `omp_target_alloc` and `omp_target_memcpy` (~2x increase in data management code)

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Managed Memory (#1)</th>
<th>Explicit Mgmt - directives (#2)</th>
<th>Explicit Mgmt - runtime API (#3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XLC 16.1.1-5</td>
<td>✓</td>
<td>X (RE)</td>
<td>✓</td>
</tr>
<tr>
<td>CCE 10.0.2</td>
<td>✓</td>
<td>X (RE)</td>
<td>✓</td>
</tr>
<tr>
<td>LLVM 11.0.0-git (#17d8334)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GCC 9.1.0 (Mentor)</td>
<td>✓</td>
<td>X (CE)</td>
<td>✓</td>
</tr>
</tbody>
</table>

CE = Compile Error
RE = Runtime Error
Explicit data management significantly improves upon Managed Memory performance on Cori-GPU (x86)
Conclusions

- 3 OpenMP / OpenACC compilers achieved 70-90% of CUDA performance (LLVM/Clang, XL, PGI)
- OpenMP / OpenACC data directives enabled us to add explicit data management to HPGMG in a much simpler way than runtime APIs
  - Hard to imagine porting HPGMG to runtime API data management without first starting from a data directive version
  - Non-trivial usage of data directives caused issues in 3 OpenMP compilers (XL, CCE, GCC) [compiler maturity issues]
- Managed Memory performance was relatively poor on x86 whether using CUDA, OpenACC or OpenMP
  - Explicit data management performed well on x86 and Power
Thanks for listening

Contact: csdaley AT lbl.gov

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

This research also used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.