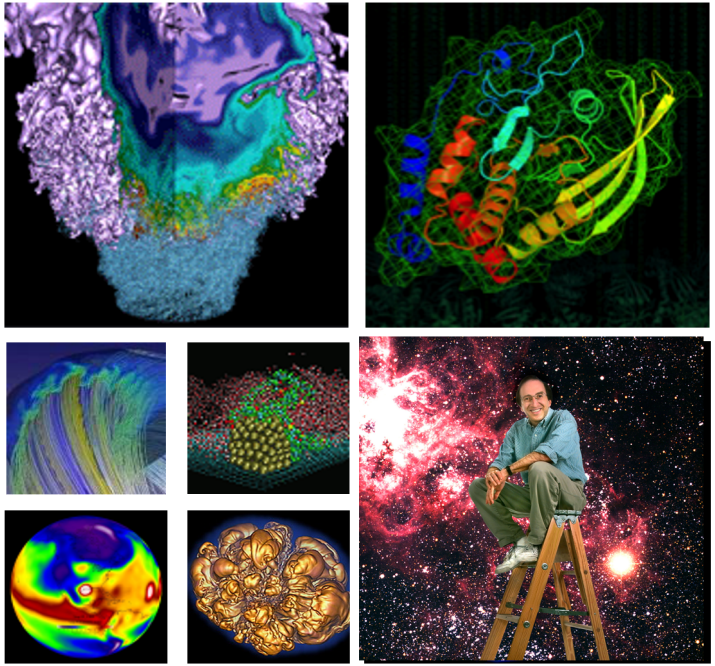
A Case Study of Porting HPGMG from CUDA to OpenMP Target Offload



Christopher Daley, Hadia Ahmed, Sam Williams, Nicholas Wright (LBNL/NERSC),
IWOMP 2020 – September 22

U.S. DEPARTMENT OF ENERGY | Office of Science

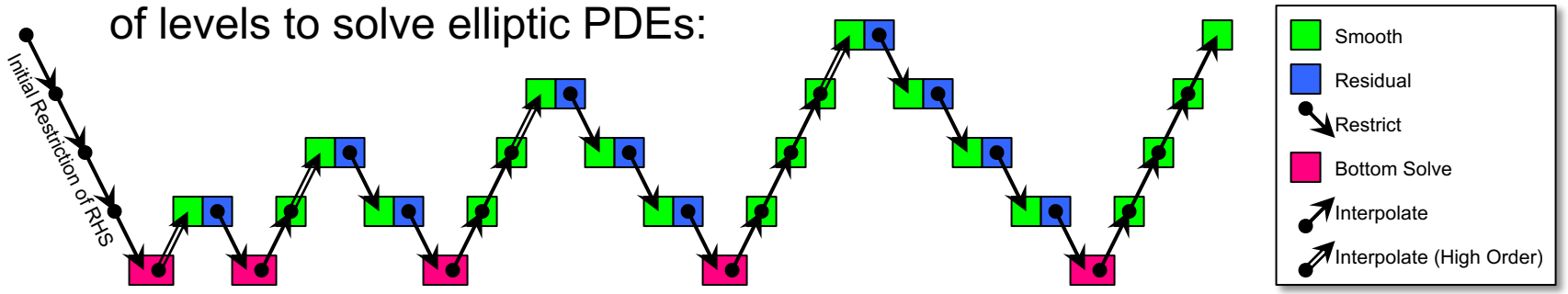
BERKELEY LAB

Overview

- **This presentation will describe how we ported HPGMG to OpenMP target offload and show performance results for several compilers**
- **HPGMG is a Finite Volume Geometric Multigrid benchmark**
- **We will consider two versions of HPGMG**
  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 methods 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)**
  - Level data allocated in Managed Memory (cudaMallocManaged)
  - Level data structure shallow copied in each CUDA kernel
- **We ported HPGMG-CUDA to OpenMP target offload 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 loops
  - Map Level data structure in every single OpenMP target region (data is still allocated using cudaMallocManaged)

Platforms used

	Cori-GPU	Summit
Node architecture	Cray CS-Storm 500NX	IBM AC922
Node CPUs	2 x Intel Skylake	2 x IBM Power 9
Available cores per CPU	20 @ 2.40 GHz	21 @ 3.07 GHz
Node GPUs	8 x 16 GB NVIDIA V100	6 x 16 GB NVIDIA V100
CPU-GPU interconnect	PCIe 3.0 switch	NVLink 2.0

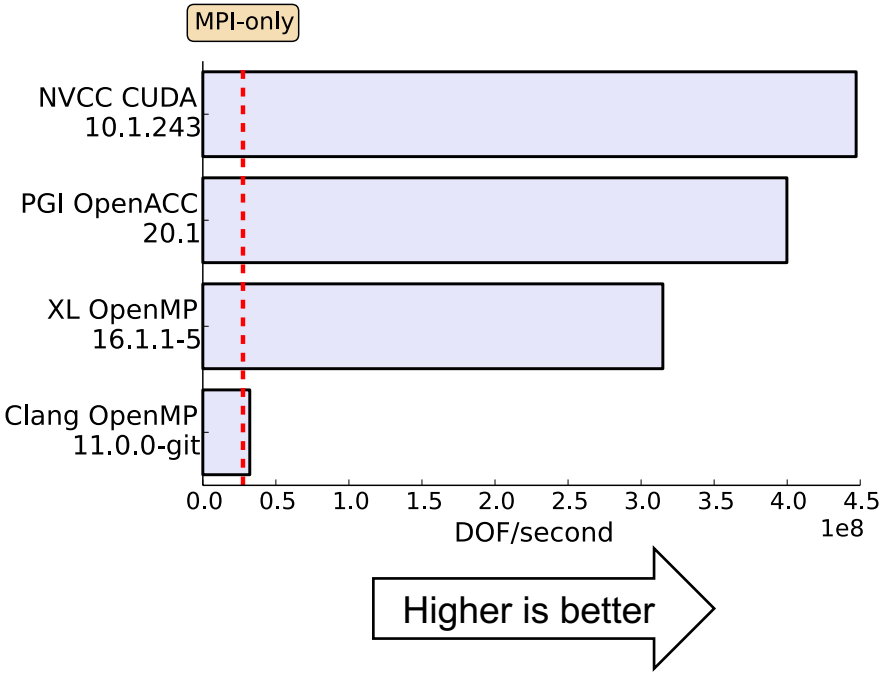
Compilers used

Compiler	GPU offload	Cori-GPU version	Summit version
GCC + NVCC	CUDA	7.3.0 + 10.1.243	7.4.0 + 10.1.243
NVIDIA/PGI	OpenACC	20.4	20.1
Cray CCE	OpenMP	9.1.0 (LLVM version)	-
IBM XL	OpenMP	-	16.1.1-5
LLVM/Clang	OpenMP	11.0.0-git (#17d8334)	11.0.0-git (#17d8334)

HPGMG configuration used

- **We used the Top-500 HPGMG configuration: 4th order accurate, GSRB smoother, and BiCGStab bottom solver**
- **Grid spacing = 1/512: creates 9 levels from 2^3 to 512^3 grid points**
  - Maximum block size = 32^3
  - Thousands of blocks on the finest level
- **Memory footprint ~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

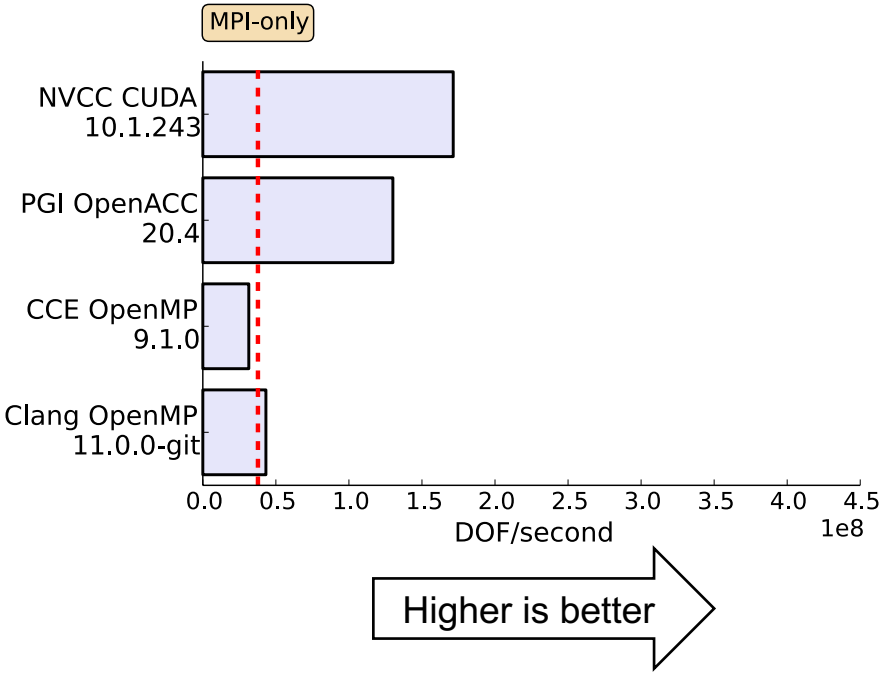


NVCC CUDA: 16x faster than the MPI-only configuration on a single CPU (21c)

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



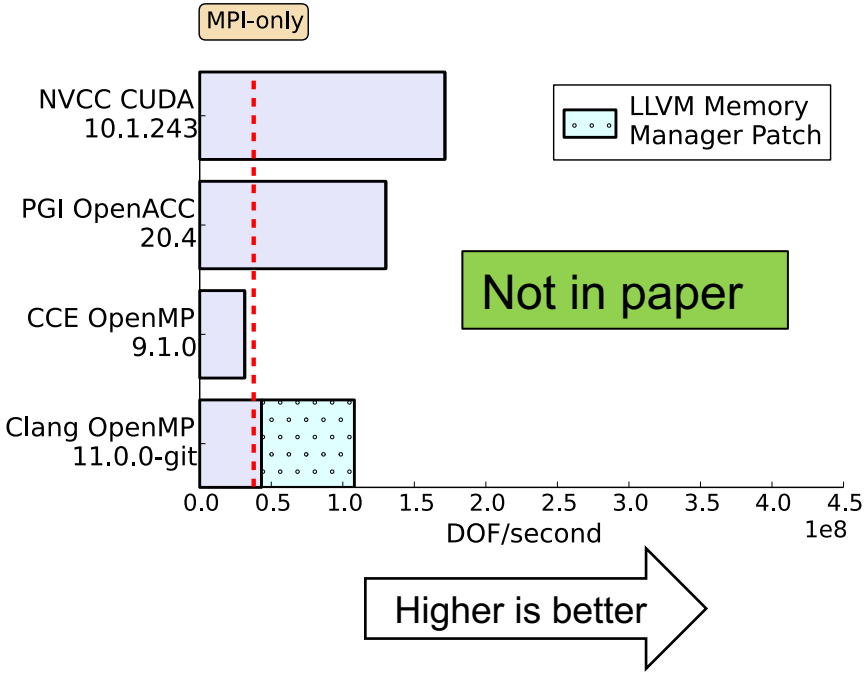
NVCC CUDA and PGI OpenACC are 2.6x and 3.1x slower on Cori-GPU than Summit!

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

Managed Memory performance on Cori-GPU: 1 Skylake CPU and 3 Volta GPUs



LLVM Memory manager patch from Shilei Tian improves Clang performance (upstream commit #0289696):

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)

Code Version #2: Explicit data management using data directives


```
void smooth(level_type level, ...)
{
#pragma omp target teams distribute map(to:level)
  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

```
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 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

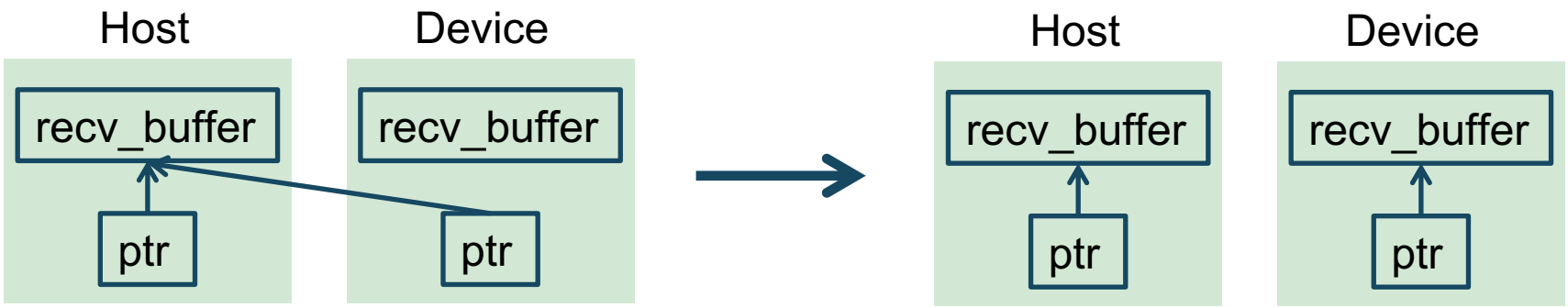
The “level” data structure is complicated – ~250 lines of code to create it on the device

```
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

We mapped dynamically allocated data to the GPU, however, a complication is that 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




```
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

- **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!

Optimizing performance with the LLVM/Clang compiler

- **We found that LLVM OpenMP runtime overhead was related to the size of the OpenMP present table (https://bugs.llvm.org/show_bug.cgi?id=46107)**
  - An OpenMP runtime uses a present table to maintain the association between host and device pointers
- **The present table got large because we updated ~100K HPGMG block pointers using "target enter data"**
- **In the following slides we show 2 ways that we reduced the size of the OpenMP present table to improve performance**
  - We also show 2 other optimizations to improve performance

Optimization #1: Don't update device pointer if host pointer is NULL

```
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) {

     if (level->exchange_ghosts[shape].blocks[block][b].read.ptr) {

#pragma omp target enter data \
   map(alloc:level->exchange_ghosts[shape].blocks[block][b].read.ptr[:0])
```

Add if statement

Summit: 5.9x speedup
Cori-GPU: 6.6x speedup

Optimization #2: Minimize present table size by manually attaching device pointers


```
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) {

      if (level->exchange_ghosts[shape].blocks[block][b].read.ptr) {

       omp_attach((void**)&level->exchange_ghosts[shape].blocks[block][b].read.ptr);
```

Retain if statement

Create a function *omp_attach* to attach a device pointer in a GPU kernel – does not add an entry to the LLVM OpenMP present table

Summit: 4.1x speedup
Cori-GPU: 5.3x speedup

Optimization #3: Use CUDA-aware MPI

Initial code

```
#pragma omp target update from(send_buf[:level->exchange_ghosts[shape].send_sizes[n]])
MPI_Isend(send_buf, ...); // send_buf is a host address
```

CUDA-aware MPI code

```
#pragma omp target data use_device_ptr(send_buf)
{
  MPI_Isend(send_buf, ...); // send_buf is a device address
}
```

Summit: 1.1x speedup
Cori-GPU: 1.3x speedup

Optimization #4: SPMDize kernels

Initial code

```
#pragma omp target teams distribute
for (int block = 0; block < num_my_blocks; block++) {
# pragma omp parallel for collapse(3)
  for (int k = klo; k < khi; k++) {
    // ...
```

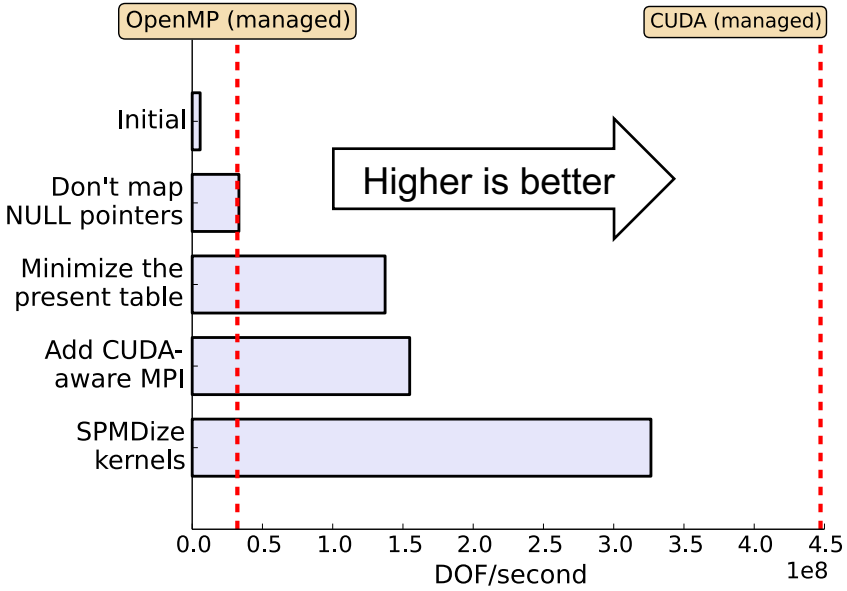
SPMDized code

```
#pragma omp target teams
# pragma omp parallel
{
  // Manually distribute outer loop over teams using team ID
  for (int block = blockStart; block < blockEnd; block++) {
#  pragma omp for collapse(3)
    for (int k = klo; k < khi; k++) {
      // ...
```

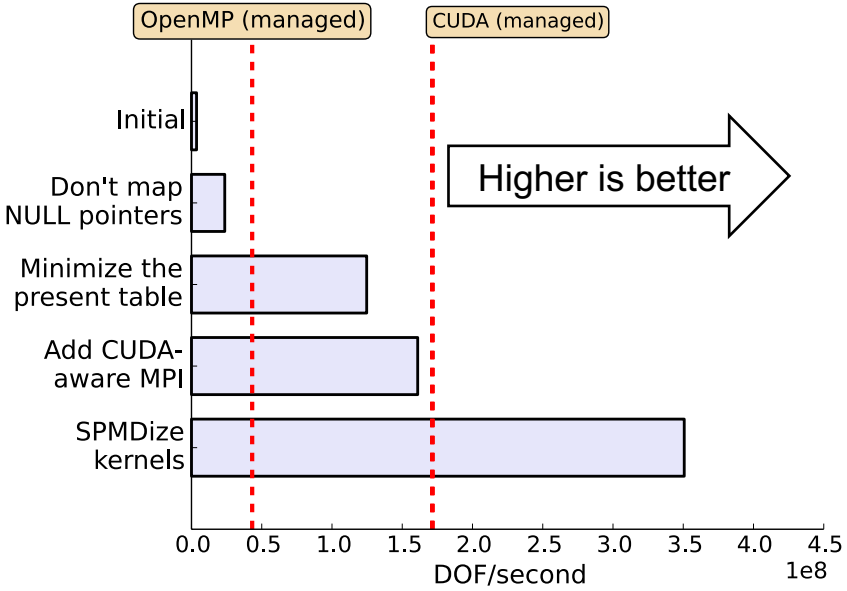
Transformed version creates all parallelism upfront to ensure each thread is executing the same code

Summit: 2.1x speedup
Cori-GPU: 2.2x speedup

Impact of successive optimizations on Summit and Cori-GPU



a). Summit – 57.6x gain



b). Cori-GPU – 97.0x gain

Final version has similar performance on both platforms

Conclusions

- **LLVM/Clang, XL and Cray compilers successfully executed the managed memory version of HPGMG**
  - The XL compiler achieved 70% of CUDA performance on Summit
- **We created an explicit data management version of HPGMG using OpenMP directives – much simpler than using APIs**
- **Only LLVM/Clang successfully executed the explicit data management version of HPGMG**
  - Initial performance was poor (worse than managed memory version)
  - We improved performance significantly by working around overheads in LLVM/Clang: 57.6x on Summit and 97.0x on Cori-GPU

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.

The LLVM OpenMP runtime spends a long time in “target update from” directive

```
122      if (omp_get_num_devices() > 0) {
123        int isp = omp_target_is_present(send_buf, omp_get_default_device());
124        if (isp != 0) {
125#  pragma omp target update from(send_buf[:level->exchange_ghosts[shape].send_sizes[n]])
126        }
127      }
128# endif
129    } /* End of send_buf_d != NULL */
130    /* The OpenMP target data region remains open for the MPI_Isend */
131#endif
```

Top-down view | Bottom-up view | Flat view

Scope	REALTIME (usec):Sum (I)		REALTIME (usec):Sum (E)	
std::_Rb_tree_increment(std::_Rb_tree_node_base*)	8.20e+09	11.4%	8.20e+09	11.4%
target_data_update(DeviceTy&, int, void**, void**, long*, long*)	8.13e+09	11.3%	8.13e+09	11.3%
__tgt_target_data_update	8.13e+09	11.3%	8.13e+09	11.3%
125: exchange_boundary	7.99e+09	11.1%	7.99e+09	11.1%
smooth	7.78e+09	10.8%	3.00e+04	0.0%
11: residual	7.22e+08	1.0%		
192: interpolation_v2	3.23e+08	0.4%	5.00e+03	0.0%
259: interpolation_v4	1.89e+08	0.3%		
95: rebuild_operator_blackbox	1.45e+08	0.2%		
195: rebuild_operator	8.82e+06	0.0%		
990: MGBuild	1.75e+06	0.0%		
327: copy_one_vector_to_device_openmp	6.92e+07	0.1%	6.92e+07	0.1%
359: copy_one_vector_to_host_openmp	6.86e+07	0.1%	6.86e+07	0.1%

It is not because of data movement!

80% of total runtime spent in a libstdc++ function called by the LLVM OpenMP runtime

(HPCToolkit percentages are a little confusing: total inclusive time in HPGMG is considered to be 14.3%. 11.4% of 14.3% is 80%)

omp_attach implementation

```
void omp_attach(void **ptr)
{
  void *dptr = *ptr;
  if (dptr) {
#pragma omp target data use_device_ptr(dptr)
    {

#pragma omp target is_device_ptr(dptr)
      {
        *ptr = dptr;
      }

    }
  }
}
```

omp_attach is passed the address of host pointer

Get the address of the host pointer target (pointee)

Get the device pointer target corresponding to the host pointer target

Use a GPU kernel to update the device pointer, *ptr, to point to the device pointer target (i.e. the mapped array)

omp_attach implementation (version 2)

```
void omp_attach(void **hptr_address)
{
  if (*hptr_address) {
    void *dptr_address = hptr_address;
    void *dptr_value = *hptr_address;
#pragma omp target data use_device_ptr(dptr_address, dptr_value)
    {
      // Do a bitwise copy of the pointer address value (&dptr_value) stored on the
      // host to the device pointer address (dptr_address)
      omp_target_memcpy(dptr_address, &dptr_value, sizeof(void*), 0, 0,
                        omp_get_default_device(), omp_get_initial_device());
    }
  }
}
```