Experience with OpenMP 4: Performance Portability and Challenges

David Appelhans

COE Performance Portability Meeting
April 20, 2016

Use Case

Kripke mini-app from LLNL (Adam Kunen).

Captures behavior of neutron transport code Ardra.

Discretization space:

- (Z) Diamond-Difference discretization into spatial zones.
- (D) Quadrature points over directions.
- (G) Energy is binned into G groups.

Solution Steps

Iterative solution:

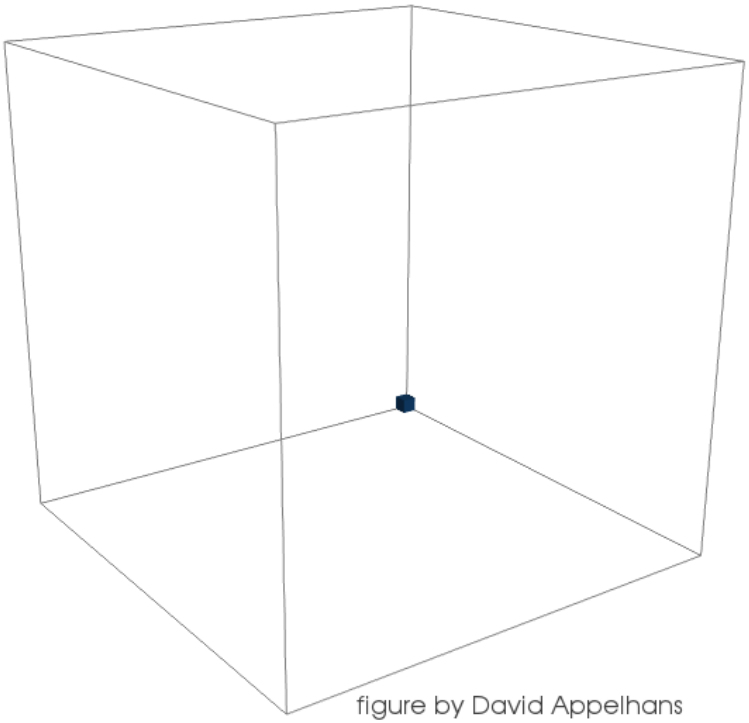
$$H\Psi^{i+1} = L^{+}\Sigma_s L\Psi^{i} + Q$$

Calculation of RHS

- Mostly matrix-matrix multiplication.
- All implementations (C / OpenMP / CUDA) can use a batched GEMM library for the RHS construction.
- This achieves 900 GF/s, or 60% of the achievable peak on the GPU.
- **Library solution to performance portability.**

SOLVE OF $H\Psi = RHS$ (SWEEP)

- H is block-diagonal in G and D.
- Upwind data dependence in zones, determines scaling.
- **No library solution available.**



Animation plays only in Acrobat Reader

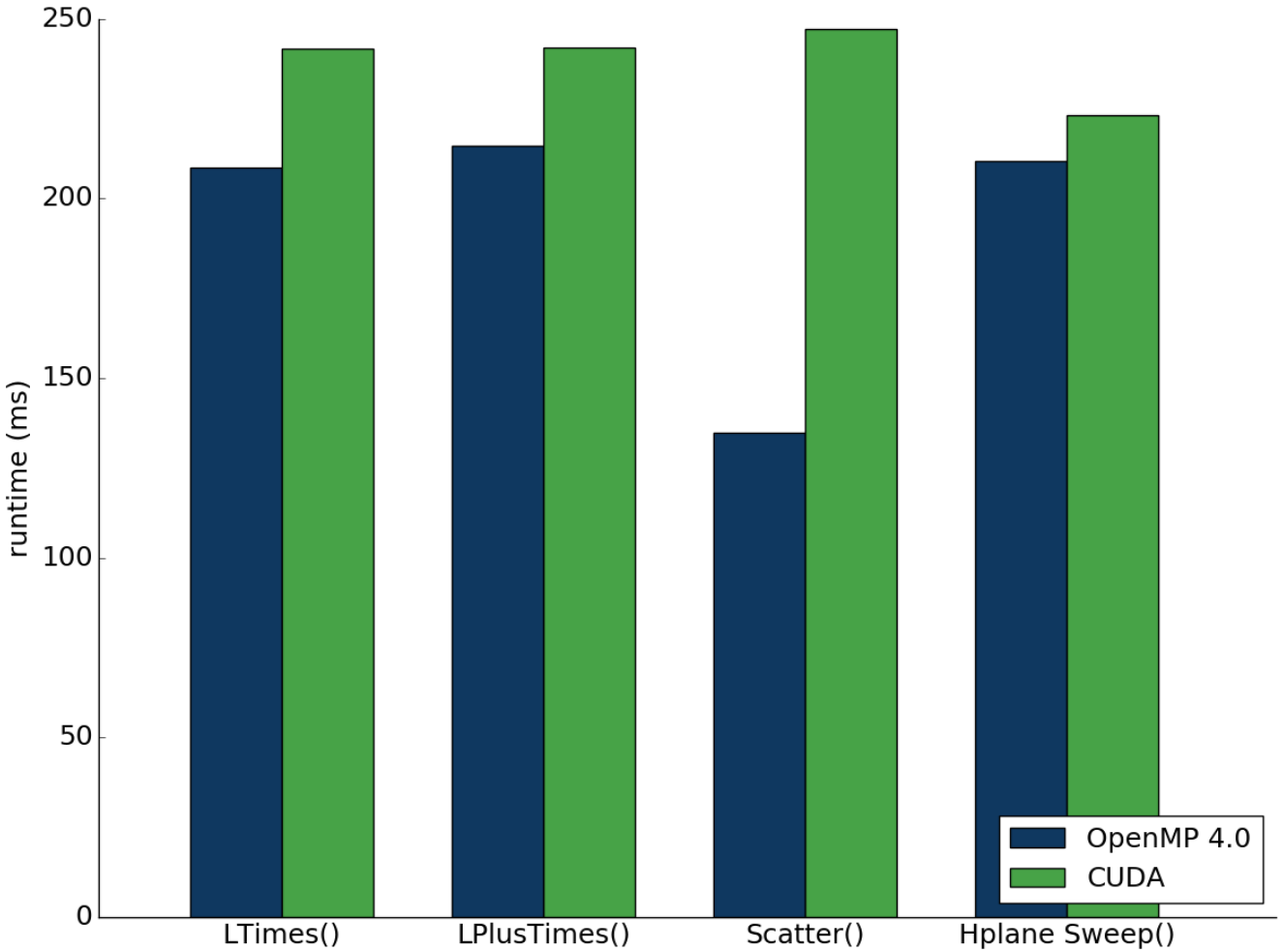
Operate in parallel along hyperplanes of zones where data dependency has been met.

Groups and directions add further parallelism.

Optimized CUDA as a Baseline

- What if library dependence is not desired, or does not exist?
- Previous researcher developed handwritten, tuned, CUDA versions of kernels.
- Tuned: reordered loops, register blocking, unrolling, shared memory, const, restrict...
- Expected CUDA to give a baseline for OpenMP 4 performance.

CUDA and OpenMP 4 Kernel Comparison



NOT THE EXPECTED RESULTS. WHY?

- OpenMP 4 uses *teams* which naturally lead to launching a number of blocks that are a multiple of SM's.
- Collapse clause allows easy and flexible combining of nested loops.
- Easy to merge updates to Scatter() helper arrays in development trunk with OpenMP 4 codebase branch.

Not the Expected Results. Why?

- With enough coding, CUDA could replicate performance and be slightly faster.
- **That is the point: optimization was easy and natural in OpenMP 4.**
- **Shows OpenMP 4 performance can be very close to the CUDA implementation.**

REASON 1: GPU BLOCK LAUNCH EXAMPLE

OpenMP 4

```
#pragma omp teams distribute num_teams(16*SMS)
for (int z = 0; z < num_zones; z++) {
```

Common CUDA

```
dim3 threadsPerBlock(32,4);
LTimes_ZDG< «num_zones,threadsPerBlock,shared_size» >(...)
int z = blockIdx.x;
```

Better CUDA

```
for(int z = blockIdx.x; z<num_zones; z+=num_blocks) {
```

Reason 2: Collapse Example

OpenMP 4: distribute threads among parallelism

```
1  // Loop over the hyperplanes (slices).
2  for (int slice = 0; slice < Nslices; slice++){
3    #pragma omp target teams distribute parallel for collapse(3) schedule(static,1) \\
4       num_teams(NUMTEAMS) thread_limit(64)
5    for (int element = offset[slice]; element < offset[slice+1]; ++element) {
6      for (int d = 0; d < num_directions; ++d) {
7        for (int group = 0; group < num_groups; ++group) {
```

CUDA 2D grid: fixed parallelism hierarchy

```
1  // Kernel called for each hyperplane (slices)
2  sweep_over_hyperplane_ZDG<<<numBlocks,threadsPerBlock>>>(...){
3
4    int element = offset[sliceID] + blockIdx.x;
5    if (element > offset[sliceID+1]) return;
6    for (int d = threadIdx.y; d < num_directions; d += blockDim.y){
7      for (int group = threadIdx.x; group < num_groups; group += blockDim.x){
```

What if num_groups < 32? OpenMP Collapse still uses all threads in a warp.

OpenMP GPU remarks

- OpenMP 4 built-in functionality allows efficient, clean code to be written.
- **Clean code makes optimization and maintenance easier.**
- Underlying code is not obscured by messy loop scheduling, parameter choices, etc.
- Quick to interchange loops, unroll, register block, etc.

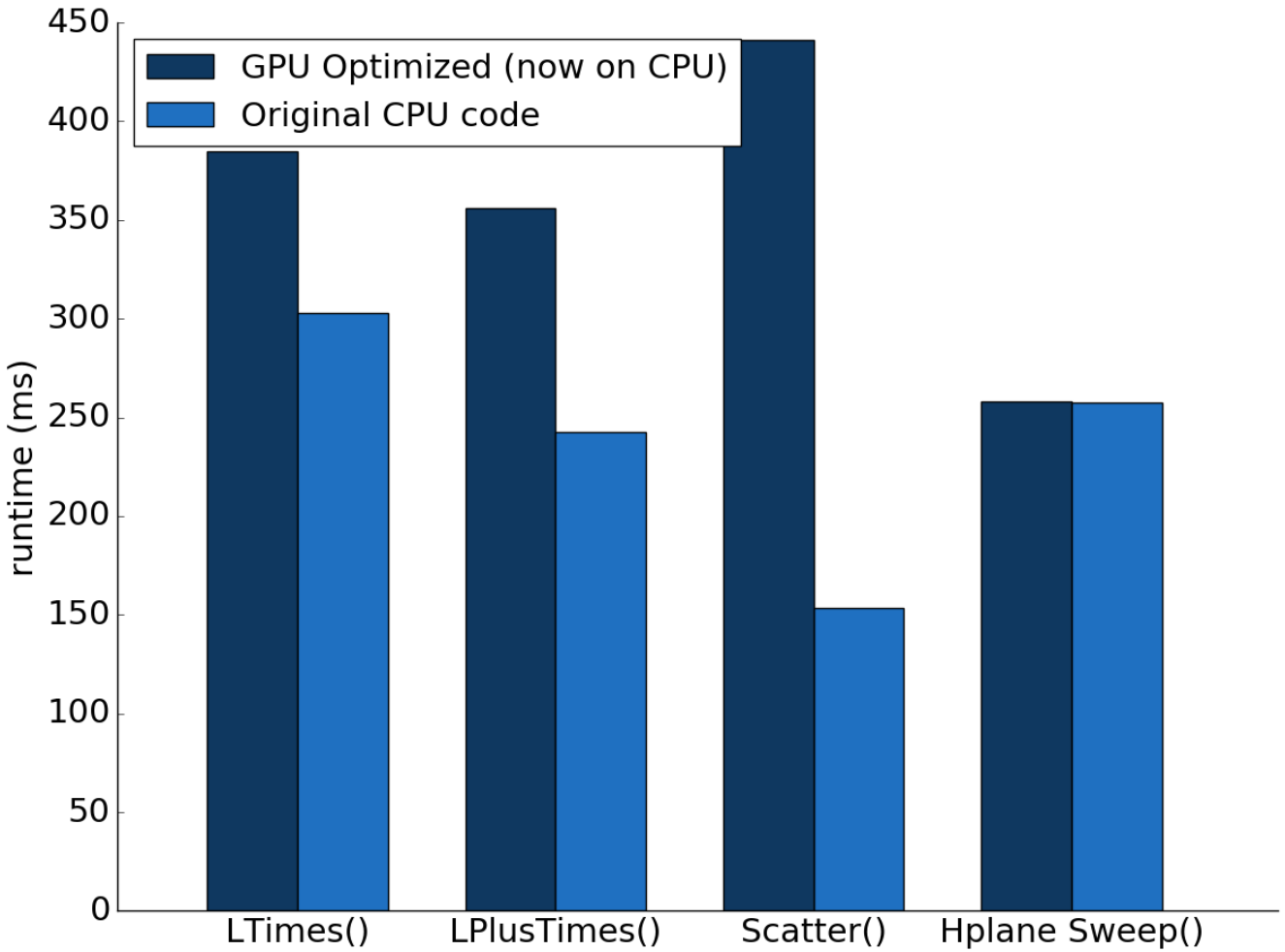
CPU CONSIDERATIONS

- OpenMP 4 code optimized for GPU runs correctly on CPU.
- What about performance?
- Some GPU optimizations hurt CPU performance.
- **Goal:** achieve original CPU performance with GPU optimized Kernels.

Tip: CPU threads usually perform well on outermost parallelism.

Expect team level parallelism to translate well to CPU threads, set environment variable OMP_NESTED=FALSE

PORTABLE OPENMP 4 KERNELS VS ORIGINAL OPENMP ON CPU



ORIGINAL LTIMES

```
1  void LTimes(...){
2
3    #ifdef KRIPKE_USE_OPENMP
4    #pragma omp parallel for
5    #endif
6    for (int z = 0; z < num_zones; z++) {
7      double const* __restrict__ psi_z = psi + z*num_locgd;
8      double* __restrict__ phi_z = phi + z*num_gnm;
9
10     for(int nm = 0;nm < num_moments;++nm){
11       double* __restrict__ phi_z_nm_g0 = phi_z + nm*num_groups + group0;
12
13       for (int d = 0; d < num_local_directions; d++) {
14         double const ell_d_nm = ell[nm + d*num_moments];
15         double const* __restrict__ psi_z_d = psi_z + d*num_local_groups;
16
17         for (int g = 0; g < num_local_groups; ++g) {
18           phi_z_nm_g0[g] += ell_d_nm*psi_z_d[g];
19         }
20       }
21     }
22   }
23 }
```

PERFORMANCE PORTABLE LTIMES

Notice loop ordering of g and d switched. Accumulating in p0 was faster on GPU.

```
1  void LTimes(...) {
2
3    #if GPU_FLAG
4      #pragma omp target teams distribute parallel for collapse(3) schedule(static,1) \\
5        num_teams(NUMTEAMS) thread_limit(THREADNUM)
6    #else
7      #pragma omp parallel for num_threads(NUMTEAMS)
8    #endif
9    for (int z = 0; z < num_zones; z++) {
10     for(int nm = 0;nm < num_moments;++nm){
11       for (int g = 0; g < num_local_groups; ++g) {
12         const double * __restrict__ psi_z = &psi[z*num_locgd];
13         double * __restrict__ phi_z = &phi[z*num_gnm];
14         double p0=0;
15         for (int d = 0; d < num_local_directions; ++d) {
16           p0 += ell[nm+ d*num_moments] * psi_z[g + d*num_local_groups];
17         }
18         phi_z[g+nm*num_groups+group0] += p0;
19       }
20     }
21   }
22 }
```

#if will not be needed with future OpenMP language improvements.

Original Hyperplane

```
1  #pragma omp parallel
2  // Loop over the hyperplanes (slices).
3  for (int slice = 0; slice < Nslices; slice++){
4    #pragma omp for
5    for (int element = offset[slice]; element < offset[slice+1]; ++element) {
6      // load i,j,k,z
7
8      // pointer initializations
9
10     for (int d = 0; d < num_directions; ++d) {
11
12       // calculate data depending on d
13
14       for (int group = 0; group < num_groups; ++group) {
15
16         /* Calculate new zonal flux */
17
18         /* Apply diamond-difference relationships */
19
20       }
21     }
22   } //end element (distribute)
23 }//end of "for (slice"
```

OpenMP 4 Hyperplane on GPU

Combined construct allows flexible parallelism. Compiler optimizations on combined construct.

```
1  // Loop over the hyperplanes (slices).
2  for (int slice = 0; slice < Nslices; slice++){
3  #pragma omp target teams distribute parallel for collapse(3) \\
4    schedule(static,1) num_teams(NUMTEAMS) thread_limit(64) if(GPU_FLAG)
5    for (int element = offset[slice]; element < offset[slice+1]; ++element) {
6      for (int d = 0; d < num_directions; ++d) {
7        for (int group = 0; group < num_groups; ++group) {
8          // load i,j,k,z
9          ...
10         // pointer initializations
11         ...
12         // calculate data depending on d
13         ...
14         /* Calculate new zonal flux */
15         ...
16         /* Apply diamond-difference relationships */
17
18       }
19     }
20   } //end element (distribute)
21 }//end of "for (slice"
```

OpenMP 4 Performance Portable Hyperplane

Need to use #if to move loops for best CPU performance...

```
1  // Loop over the hyperplanes (slices).
2  for (int slice = 0; slice < Nslices; slice++){
3  #if GPU_FLAG
4  #pragma omp target teams distribute parallel for collapse(3) \\
5      schedule(static,1) num_teams(NUMTEAMS) thread_limit(64)
6    for (int element = offset[slice]; element < offset[slice+1]; ++element) {
7      for (int d = 0; d < num_directions; ++d) {
8       for (int group = 0; group < num_groups; ++group) {
9  #else
10 #pragma omp for
11       for (int element = offset[slice]; element < offset[slice+1]; ++element) {
12 #endif
13         // load i,j,k,z
14         // pointer initializations
15 #if !(GPU_FLAG)
16         for (int d = 0; d < num_directions; ++d) {
17 #endif
18           // calculate data depending on d
19 #if !(GPU_FLAG)
20           for (int group = 0; group < num_groups; ++group) {
21 #endif
22             /* Calculate new zonal flux */
23             /* Apply diamond-difference relationships */
24           }
25         }
26       } //end element (distribute)
27     }//end of "for (slice"
28   }
```

CPU REMARKS:

- #if statements to turn off entire OpenMP directives is a *temporary* issue. Should be addressed by the standard.
- Reordering loops and accumulating in registers was benificial on GPU, but degraded CPU performance.
- Using collapse clause was benificial on GPU because of amount of parallelism exposed.
- Manual loop hoisting still needed, which means use of #if to move loop locations when on CPU.

OpenMP 4 Conclusions

- Can be a competitive alternative to CUDA performance on the GPU.
- Unaltered code runs on different architectures (**portability**).
- Achieving maximum **performance** on the GPU can lead to a degredation in CPU **performance**.
- These are highly GPU optimized kernels–there is a middle ground.
- Ability to selectively move loops or have automatic hoisting would benefit performance portability.

QUESTIONS?

David Appelhans - dappelh@us.ibm.com

Alternative OpenMP 4 Hyperplane SIMD

```
1   #pragma omp target data map(........) if(GPU_FLAG)
2      // Loop over the hyperplanes (slices).
3
4      for (int slice = 0; slice < Nslices; slice++){
5        int hplane_size = offset[slice+1]-offset[slice];
6
7   #pragma omp target if(GPU_FLAG)
8   #pragma omp teams distribute num_teams(hplane_size) thread_limit(2)
9         for (int element = offset[slice]; element < offset[slice+1]; ++element) {
10          // load i,j,k,z
11          ...
12          // Pointer initializations
13          ...
14  #pragma omp parallel for
15          for (int d = 0; d < num_directions; ++d) {
16            // calculate cos info depending on d.
17            ...
18  #pragma omp simd
19            for (int group = 0; group < num_groups; ++group) {
20              ...
21              /* Calculate new zonal flux */
22              ...
23              psi_z_d[gd] = psi_z_d_g;
24              /* Apply diamond-difference relationships */
25              ...
26            }
27          }
28        } //end element (distribute)
29     }//end "for (slice"
```

Original LTimes()

```
1  #ifdef KRIPKE_USE_OPENMP
2  #pragma omp parallel for
3  #endif
4  for (int z = 0; z < num_zones; z++) {
5    double const* __restrict__ psi_z = psi +
         z*num_locgd;
6    double* __restrict__ phi_z = phi +
         z*num_gnm;
7    for(int nm = 0;nm < num_moments;++nm){
8      double* __restrict__ phi_z_nm_g0 =
           phi_z + nm*num_groups + group0;
9      for (int d = 0; d <
           num_local_directions; d++) {
10       double const ell_d_nm = ell[nm +
             d*num_moments];
11       double const* __restrict__ psi_z_d =
             psi_z + d*num_local_groups;
12       for (int g = 0; g < num_local_groups;
             ++g) {
13         phi_z_nm_g0[g] +=
               ell_d_nm*psi_z_d[g];
14       }
15     }
16   }
17 }
```

OpenMP 4

```
1  #if GPU_FLAG
2    #pragma omp target teams distribute
         parallel for collapse(3)
         schedule(static,1) \\
3      num_teams(NUMTEAMS)
           thread_limit(THREADNUM)
4  #else
5    #pragma omp parallel for
         num_threads(NUMTEAMS)
6  #endif
7  for (int z = 0; z < num_zones; z++) {
8    for(int nm = 0;nm < num_moments;++nm){
9      for (int g = 0; g < num_local_groups;
           ++g) {
10       const double * __restrict__ psi_z =
             &psi[z*num_locgd];
11       double * __restrict__ phi_z =
             &phi[z*num_gnm];
12       double p0=0;
13       for (int d = 0; d <
             num_local_directions; ++d) {
14         p0 += ell[nm+ d*num_moments] *
               psi_z[g + d*num_local_groups];
15       }
16       phi_z[g+nm*num_groups+group0] += p0;
17     }
18   }
19 }
```