Experience with OpenMP 4: Performance Portability and Challenges

David Appelhans

COE Performance Portability Meeting
April 20, 2016

IBM
**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:

\[ H\Psi^{i+1} = L^+\sum_s L\Psi^i + Q \]

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.**
\textbf{Solve of } \mathbf{H}\Psi = \mathbf{RHS \ (Sweep)}

- \( \mathbf{H} \) is block-diagonal in \( \mathbf{G} \) and \( \mathbf{D} \).
- Upwind data dependence in zones, determines scaling.
- \textbf{No library solution available.}

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

Groups and directions add further parallelism.

\textit{Animation plays only in Acrobat Reader}
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

![Bar chart comparing CUDA and OpenMP 4 runtime for different kernels: LTimes, LPlusTimes, Scatter, Hplane Sweep. The chart shows CUDA outperforms OpenMP 4 in terms of runtime.]
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**

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

**Common CUDA**

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

**Better CUDA**

```c
for(int z = blockIdx.x; z<num_zones; z+=num_blocks) {
```
Reason 2: Collapse Example

OpenMP 4: distribute threads among parallelism

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

CUDA 2D grid: fixed parallelism hierarchy

```c
// Kernel called for each hyperplane (slices)
sweep_over_hyperplane_ZDG<<<numBlocks,threadsPerBlock>>>(...){
    int element = offset[sliceID] + blockIdx.x;
    if (element > offset[sliceID+1]) return;
    for (int d = threadIdx.y; d < num_directions; d += blockDim.y){
        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
Performance Portable LTimes

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

```c
void LTimes(...) {
    #if GPU_FLAG
        #pragma omp target teams distribute parallel for collapse(3) schedule(static,1) \ 
            num_teams(NUMTEAMS) thread_limit(THREADNUM)
    #else
        #pragma omp parallel for num_threads(NUMTEAMS)
    #endif
    for (int z = 0; z < num_zones; z++) {
        for (int nm = 0; nm < num_moments; ++nm) {
            for (int g = 0; g < num_local_groups; ++g) {
                const double * __restrict__ psi_z = &psi[z*num_locgd];
                double * __restrict__ phi_z = &phi[z*num_gnm];
                double p0=0;
                for (int d = 0; d < num_local_directions; ++d) {
                    p0 += ell[nm+ d*num_moments] * psi_z[g + d*num_local_groups];
                }
                phi_z[g+nm*num_groups+group0] += p0;
            }
        }
    }

#if will not be needed with future OpenMP language improvements.
```
// Loop over the hyperplanes (slices).
for (int slice = 0; slice < Nslices; slice++) {
    #pragma omp for
    for (int element = offset[slice]; element < offset[slice+1]; ++element) {
        // load i, j, k, z
        // pointer initializations
        for (int d = 0; d < num_directions; ++d) {
            // calculate data depending on d
            for (int group = 0; group < num_groups; ++group) {
                /* Calculate new zonal flux */
                /* Apply diamond-difference relationships */
            }
        }
    }
} //end element (distribute)
}//end of "for (slice"
Combined construct allows flexible parallelism. Compiler optimizations on combined construct.

```c
// Loop over the hyperplanes (slices).
for (int slice = 0; slice < Nslices; slice++) {  
#pragma omp target teams distribute parallel for collapse(3)  
schedule(static,1) num_teams(NUMTEAMS) thread_limit(64) if(GPU_FLAG)  
for (int element = offset[slice]; element < offset[slice+1]; ++element) {  
    for (int d = 0; d < num_directions; ++d) {  
        for (int group = 0; group < num_groups; ++group) {  
            // load i,j,k,z  
            ...  
            // pointer initializations  
            ...  
            // calculate data depending on d  
            ...  
            /* Calculate new zonal flux */  
            ...  
            /* Apply diamond-difference relationships */  
        }  
    }  
} //end element (distribute)  
} //end of "for (slice"
```
// Loop over the hyperplanes (slices).
for (int slice = 0; slice < Nslices; slice++) {
    #if GPU_FLAG
        #pragma omp target teams distribute parallel for collapse(3) \ 
            schedule(static,1) num_teams(NUMTEAMS) thread_limit(64)
        for (int element = offset[slice]; element < offset[slice+1]; ++element) {
            for (int d = 0; d < num_directions; ++d) {
                for (int group = 0; group < num_groups; ++group) {
                    #else
                        #pragma omp for
                        for (int element = offset[slice]; element < offset[slice+1]; ++element) {
                            #endif
                            // load i,j,k,z
                            // pointer initializations
                            #if !(GPU_FLAG)
                                for (int d = 0; d < num_directions; ++d) {
                                    #endif
                                    // calculate data depending on d
                                    #if !(GPU_FLAG)
                                        for (int group = 0; group < num_groups; ++group) {
                                            #endif
                                            /* Calculate new zonal flux */
                                            /* Apply diamond-difference relationships */
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    #else
        #pragma omp for
        for (int element = offset[slice]; element < offset[slice+1]; ++element) {
            #endif
            // load i,j,k,z
            // pointer initializations
            #if !(GPU_FLAG)
                for (int d = 0; d < num_directions; ++d) {
                    #endif
                    // calculate data depending on d
                    #if !(GPU_FLAG)
                        for (int group = 0; group < num_groups; ++group) {
                            #endif
                            /* Calculate new zonal flux */
                            /* Apply diamond-difference relationships */
                        }
                    }
                }
            } //end element (distribute)
        } //end of "for (slice"
**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 beneficial on GPU, but degraded CPU performance.

- Using *collapse* clause was beneficial 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 degradation 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
#pragma omp target data map(........) if(GPU_FLAG) 
// Loop over the hyperplanes (slices).
for (int slice = 0; slice < Nslices; slice++){
    int hplane_size = offset[slice+1]-offset[slice];

#pragma omp target if(GPU_FLAG)
#pragma omp teams distribute num_teams(hplane_size) thread_limit(2)
for (int element = offset[slice]; element < offset[slice+1]; ++element) {
    // load i,j,k,z
    ...
    // Pointer initializations
    ...
#pragma omp parallel for
for (int d = 0; d < num_directions; ++d) {
    // calculate cos info depending on d.
    ...
#pragma omp simd
for (int group = 0; group < num_groups; ++group) {
    ...
    /* Calculate new zonal flux */
    ...
    psi_z_d[gd] = psi_z_d_g;
    /* Apply diamond-difference relationships */
    ...
}
} //end element (distribute)
}//end "for (slice"
Original LTimes()

#define Krippke_USE_OPENMP
#pragma omp parallel for
#endif
for (int z = 0; z < num_zones; z++) {
    double const* __restrict__ psi_z = psi + z*num_locgd;
    double* __restrict__ phi_z = phi + z*num_gnm;
    for (int nm = 0; nm < num_moments; ++nm) {
        double* __restrict__ phi_z_nm_g0 = phi_z + nm*num_groups + group0;
        for (int d = 0; d < num_local_directions; d++) {
            double const ell_d_nm = ell[nm + d*num_moments];
            double const* __restrict__ psi_z_d = psi_z + d*num_local_groups;
            for (int g = 0; g < num_local_groups; ++g) {
                phi_z_nm_g0[g] += ell_d_nm*psi_z_d[g];
            }
        }
    }
}

OpenMP 4

#if GPU_FLAG
#pragma omp target teams distribute parallel for collapse(3)
    schedule(static,1) \ 
    num_teams(NUMTEAMS)
    thread_limit(THREADNUM)
#else
    #pragma omp parallel for
    num_threads(NUMTEAMS)
#endif
for (int z = 0; z < num_zones; z++) {
    for (int nm = 0; nm < num_moments; ++nm) {
        for (int g = 0; g < num_local_groups; ++g) {
            const double* __restrict__ psi_z = &psi[z*num_locgd];
            double* __restrict__ phi_z = &phi[z*num_gnm];
            double p0=0;
            for (int d = 0; d < num_local_directions; ++d) {
                p0 += ell[nm + d*num_moments] * psi_z[g + d*num_local_groups];
            }
            phi_z[g+nm*num_groups+group0] += p0;
        }
    }
}