A case study of performance portability with OpenMP 4.5

Rahul Gayatri, Charlene Yang, Thorsten Kurth, Jack Deslippe

NERSC

pre-print copy
Outline

- General Plasmon Pole (GPP) application from BerkeleyGW
- Baseline CPU implementation using OpenMP 3.0
  - Reference Xeon Phi implementation
- GPU implementation
  - Naive implementation
  - Optimized implementation
  - Comparison with optimized OpenACC and CUDA implementation
- Backport OpenMP 4.5 on CPU
General Plasmon Pole (GPP)

- Mini-app from BerkeleyGW
  - Compute excited state properties of complex materials
- Characteristics of GPP
  - Reduction over a series of double complex arrays involving multiply, divide and add (FMA) instructions
  - For typical calculations, it evaluates to an arithmetic intensity between 1-10, i.e., the kernel has to be optimized for memory locality and vectorization efficiency
GPP- pseudo code

```
for(X)
    // X = 512
    for(N)
        // Y = 32768/20 (1638)
        for(M)
            // M = 32768
            for(iw)
                //iw = 3
                //Computation
                output[iw] +=
        }
    }
}
```

- Memory ~ 2GB
- Typical single-node problem size
- output - double complex
#pragma omp parallel for \
    reduction(+:output_re[0:3], output_im[0:3])
for(X){
    for(N){
        for(M){  //vectorize
            for(iw){  //unroll
                //Compute and store in local variables
            }
        }
        for(iw){
            output_re[iw] += ...
            output_im[iw] += ...
        }
    }
}
Performance on CPUs

xl/2018 - Power processors (Summit)

intel/2018 - Haswell and Xeon Phi (Cori)

- Haswell is 2X faster than P9
Performance on CPUs

Xeon Phi - 2.5 seconds
GPU threads

1. Grid (threadblocks)
2. Block (threads)
Compilers used

- OpenMP 4.5
  - xl (IBM)
    - xl/2018 (latest version on Summit)
  - clang (coral)
    - llvm (latest version on Summit)
- OpenACC
  - pgi/18.4
OpenMP map directives

```c
#pragma omp target teams distribute

#pragma omp parallel for

#pragma omp target
    map(to: input[0:X], ...) \ //Copy-to-device
    map(tofrom: ...) \ //Copy-to-from-device
    map(from: ...) \ //Copy-from-device
```
Naive OpenMP 4.5 porting

```c
#pragma omp target teams distribute \
    map(to:...) \
    map(tofrom:output_re[0:3], output_im[0:3])
for(X){
    #pragma omp parallel for
    for(N){
        for(M){
            for(iw){
                //Compute and store in local variables
            }
        }
        for(iw){
            #pragma omp atomic
            output_re[iw] += ...
            #pragma omp atomic
            output_im[iw] += ...
        }
    }
}
```

- Distribute X among threadblocks
- Distribute N among threads in a threadblock
- No array-reduction with OpenMP offload directives hence use atomic to maintain correctness
  - Hence pass output_re and output_im into map(tofrom:) clause
- Parallelizing M-loop increases the overhead of synchronization
#pragma omp target enter data map(alloc:input[0:X], ...) //Allocate data on device
#pragma omp target update to(input[0:X], ...) //Update the data
#pragma omp target teams distribute collapse(2) map(to:...) \
    map(tofrom:...) \
    reduction(+:output_re{0,1,2}, output_im{0,1,2})
for(X){
    for(N){
#pragma omp parallel for \
    reduction(+:output_re{0,1,2}, output_im{0,1,2})
        for(M){
            for(iw){
                //Compute and store in local variables
            }
            output_re{0,1,2} += …;
            output_im{0,1,2} += …;
        }
    }
#pragma omp target exit data map(delete:input[0:X], ...) //Delete allocated data
Optimized OpenMP 4.5

```c
#pragma omp target enter data map(alloc,input[0:X], ...) //Allocate data on device
#pragma omp target update to(input[0:X], ...) //Update the data
#pragma omp target teams distribute collapse(2) map(to, tofrom,...) 
    reduction(+:output_re{0,1,2}, output_im{0,1,2})
for(X){
    for(N){
        #pragma omp parallel for 
        reduction(+:output_re{0,1,2}, output_im{0,1,2})
        for(M){
            for(iw){
                //Compute and store in local variables
            }
            output_re{0,1,2} += ...;
            output_re{0,1,2} += ...;
        }
    }
}
#pragma omp target exit data map(delete:input[0:X], ...) //Delete allocated data
```

- Reduction required at threadblocks and thread level
- Reduction variables should not be added to the map clauses in the same construct
- Memory allocation improved the performance of the kernel by 10%
- Reduction gave a 3X performance boost
Optimized OpenACC

OpenACC

#pragma enter data create (input[0:X])
#pragma acc parallel loop gang collapse(2)
  present(input[0:X])
  reduction(+:output_re{0,1,2}, output_im{0,1,2})
for(X){
  for(N){
    #pragma acc loop vector
    reduction(+:output_re{0,1,2}, output_im{0,1,2})
    for(M){
      for(iw){
        //Compute and store in local variables
      }
      output_re{0,1,2} += …;
      output_re{0,1,2} += …;
    }
  }
  #pragma exit data delete (input[0:X])
}

OpenMP

#pragma omp target enter data map(alloc:input[0:X])
#pragma omp target update to(input[0:X], …)
#pragma omp target teams distribute collapse(2)
  map(to:…)
  reduction(+:output_re{0,1,2}, output_im{0,1,2})
for(X){
  for(N){
    #pragma omp parallel for
    reduction(+:output_re{0,1,2}, output_im{0,1,2})
    for(M){
      for(iw){
        //Compute and store in local variables
      }
      output_re{0,1,2} += …;
      output_re{0,1,2} += …;
    }
  }
  #pragma omp target exit data map(delete:input[0:X])
OpenMP 4.5 vs OpenACC

- Dashed line - Xeon Phi timing
- xlc-OpenMP and pgi-OpenACC give similar performance on V100
  - 2X faster than Xeon Phi
- clang-OpenMP is 20% slower compared to xlc-OpenMP
## OpenMP 4.5 vs OpenACC on V100

Kernel Generated

<table>
<thead>
<tr>
<th></th>
<th>Grid</th>
<th>Thread</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP (xl)</td>
<td>(1280,1,1)</td>
<td>(512, 1,1)</td>
<td>114</td>
</tr>
<tr>
<td>OpenACC (pgi)</td>
<td>(65535, 1,1)</td>
<td>(128,1,1)</td>
<td>136</td>
</tr>
</tbody>
</table>

- OpenACC generates ~50X more threadblocks, whereas OpenMP has 4X more threads per threadblock.
- Volta has only 80 SM.
CUDA kernels

**Version 1**

```
for(X)
    blockIDx.x
    for(N)
        threadId.x
        for(M)
            for(iw)
                //Compute and store
        
    for(iw)
        output_re[iw] += …; atomicAdd
        output_re[iw] += …; atomicAdd
```

```
dim3 numblocks(X,1,1)
dim3 (32,1,1)
```
OpenMP 4.5 vs OpenACC Vs CUDA (1)

- OpenACC and CUDA-V1 have similar runtime
- OpenMP, OpenACC and CUDA-V1 have similar performance on V100
- CUDA-V2 is 2x faster than the closest counterpart
CUDA kernel V2 (best) vs OpenMP

for(X){ //blockIdx.x
    for(N){ // blockIdx.y
        for(M){ // threadIdx.x
            for(iw){
                //Compute and store
            }
            for(iw){
                output_re[iw] += …; atomicAdd
                output_re[iw] += …;
            }
        }
    }
}

#pragma omp target enter data map(alloc:input[0:X])
#pragma omp target update to(input[0:X], …)
#pragma omp target teams distribute collapse(2)\map(to:...) \reduction(+:output_re{0,1,2}, output_im{0,1,2})
for(N){
    for(X){
        #pragma omp parallel for \reduction(+:output_re{0,1,2}, output_im{0,1,2})
            for(M){
                for(iw){
                    //Compute and store in local variables
                    output_re{0,1,2} += …;
                    output_re{0,1,2} += …;
                }
            }
    }
}
#pragma omp target exit data map(delete:input[0:X])
- OpenACC version improved by 30%
- OpenMP implementation (xl and clang) gave a 2X performance improvement
- xlc-OpenMP and CUDA-V2 give similar runtime performance
  - 4X faster than Xeon Phi
OpenMP 4.5 on CPUs

```c
#pragma omp target enter data map(alloc:input[0:X])
#pragma omp target update to(input[0:X], ...)
#pragma omp target teams distribute collapse(2)\ 
  map(to:...) \ 
 reduction(+:output_re{0,1,2}, output_im{0,1,2})
for(N){
  for(X){
    #pragma omp parallel for 
    reduction(+:output_re{0,1,2}, output_im{0,1,2})
    for(M){
      for(iw){
        //Compute and store in local variables
        }
        output_re{0,1,2} += ...;
        output_re{0,1,2} += ...;
      }
    }
  }
#pragma omp target exit data map(delete:input[0:X])
```

- Compiler on Cori - intel/2018
- Creates a single team and associates all threads to the team
- $X$ and $N$ loops are collapsed and run sequentially
- $M$-loop is distributed among the threads
GPU optimized vs CPU optimized

- GPU optimized OpenMP 4.5 is 18X slower on Xeon Phi
- CPU optimized OpenMP 4.5 is 12X slower on Volta
Summary

- Effort needed to port a kernel onto CPU and GPU using OpenMP
- Current state of OpenMP 4.5
  - Comparison with OpenACC and CUDA
  - Effect on the performance due to loop ordering
- OpenMP 4.5 on CPUs
  - Interpretation of OpenMP 4.5 directives on CPUs by intel compilers
  - Comparison of CPU optimized versus GPU optimized performance on GPU and CPUs respectively
Thank you && Questions ?
OpenMP 4.5 vs OpenACC on V100

### Kernel Generated

<table>
<thead>
<tr>
<th></th>
<th>Grid</th>
<th>Thread</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP (xl)</td>
<td>(1280, 1, 1)</td>
<td>(512, 1, 1)</td>
<td>114</td>
</tr>
<tr>
<td>OpenACC (pgi)</td>
<td>(65535, 1, 1)</td>
<td>(128, 1, 1)</td>
<td>136</td>
</tr>
</tbody>
</table>

### Hardware metrics (nvprof)

<table>
<thead>
<tr>
<th></th>
<th>Dram utilization</th>
<th>Global hit-rate</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP (xl)</td>
<td>7</td>
<td>84.6%</td>
<td>0.27</td>
</tr>
<tr>
<td>OpenACC (pgi)</td>
<td>8</td>
<td>54.05%</td>
<td>0.19</td>
</tr>
</tbody>
</table>

- OpenACC generates ~50X more threadblocks, whereas OpenMP has 4X more threads per threadblock.
- Volta has only 80 SM.
- Latency of misses is hidden by the additional threadblocks that are created.