Optimizing Large Reductions in BerkeleyGW on GPUs Using OpenMP and OpenACC

Rahulkumar Gayatri, Charlene Yang
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory
March 8, 2019

rgayatri@lbl.gov, cjiang@lbl.gov
Why Attend this Talk

- 5 of the top 10 supercomputers are using NVIDIA GPUs
- Most of the codes optimized for CPUs have to now be rewritten for GPUs
- Compiler directive based approaches are attractive due to their ease of use
  - Port incrementally for big codes
- This talk would provide a detailed analysis of the current state of the directive based programming models
  - Their performance compared to optimized CUDA code
  - Supported compilers
  - Differences in compiler implementations
Outline of the Presentation

- BerkeleyGW, a material science code
  - General Plasmon Pole (GPP), a mini-app
- Baseline CPU implementation
- GPU programming models (OpenMP, OpenACC, CUDA)
- GPP on GPU
  - Naive implementation
  - Optimized implementation
  - Compare approaches and performance of each implementation
- Backport GPU implementation on CPU for performance portability
The GW method is an accurate approach to simulate the excited state properties of materials

- What happens when you add or remove an electron from a system
- How do electrons behave when you apply a voltage
- How does the system respond to light or x-rays

- Extract stand alone kernels that could be run as mini-apps
General Plasmon Pole (GPP)

- Mini-app from BerkeleyGW
  - Computes the electron self-energy using the General Plasmon Pole approximation

- Characteristics of GPP
  - Reduction over a series of double complex arrays involving multiply, divide and add instructions (partial FMA)
  - For typical calculations, it evaluates to an arithmetic intensity (Flops/Byte) between 1-10, i.e., the kernel has to be optimized for memory locality and vectorization/SIMT efficiency
Complex Number Class

- BerkeleyGW consist of double-complex number calculation

- std::complex difficulties
  - Performance issues
  - Difficult to vectorize
  - Cannot offload operations onto the device using OpenMP 4.5

- Thrust::complex
  - Challenges in offloading complex operator routines on device

- Built an in-house complex class
  - 2-doubles on CPU
  - double2 vector type on GPU
GPP pseudo code - reduction in the innermost loop

**Code**

```c
for(X){  // X = 512
    for(N){  // N = 1638
        for(M){  // M = 32768
            for(int iw = 0; iw < 3; ++iw){
                //Some computation
                output[iw] += ...
            }
        }
    }
}
```

- Memory O(2GBs)
- Typical single node problem size
- output - double complex
GPP On CPU
OpenMP 3.0 parallelization of GPP

```c
#pragma omp parallel for
    reduction(output_re[0-2], output_im[0-2])
for(X){
    for(N){
        for(M){ //Vectorize
            for(int iw = 0; iw < 3; ++iw){ //Unroll
                //Store local
            }
        }
        for(int iw = 0; iw < 3; ++iw){
            output_re[iw] += ...
            output_im[iw] += ...
        }
    }
}
```

- Unroll innermost iw-loop
- Vectorize M-loop
- Collapse increased the runtime by 10%
- Check compiler reports (intel/2018) to guarantee vectorization and unrolling
- Flatten arrays into scalars with compilers that do not support array reduction
• Performance numbers from Cori at NERSC,LBL
  ○ Haswell
  ○ Xeon Phi

• intel/2018 compilers

• A perfect scaling would allow a KNL execution to be $4 \times$ faster than Haswell
  ○ KNL implementation of GPP is approximately $3.5 \times$ faster than Haswell
Runtime of GPP on Cori

- Performance numbers from Cori at LBNL
  - Haswell
  - Xeon Phi

- intel/2018 compilers

- A perfect scaling would allow a KNL execution to be $4 \times$ faster than Haswell
  - KNL implementation of GPP is $3 \times$ faster than Haswell
GPP On GPU
GPU Hardware

- Going from 272 to 164k threads
- 164k threads
  - 80 SMs
  - 2048 threads within a SM
GPU Programming Models

Programming Models used to port GPP on GPU

- OpenMP 4.5
  - Cray
  - XL (IBM)
  - Clang
  - GCC

- OpenACC
  - PGI
  - Cray

- CUDA

Volta GPU available on Cori and Summit

Target architecture - Volta
OpenMP offloading to GPU

- **OpenMP 4.5**
  - Cray
  - XL (IBM)
  - Clang
  - GCC

- **OpenACC**
  - PGI
  - Cray

- **CUDA**

Volta GPU available on Cori and Summit
OpenMP directives to offload code-blocks onto GPUs

**Directives to distribute work across GPU threads**

- **target** — offload the code-block on to the device
- **teams** — spawn one or more thread team
- **distribute** — distribute iterations of the loops onto master threads of the team
- **parallel for** — distribute loop iterations among threads in a threadblock
- **simd** — implementation dependent on compilers

```c
#pragma omp target teams distribute
for() //Distribute the loop across threadblocks
#pragma omp parallel for
for() //Distribute the loop across threads within a threadblock
```
OpenMP 4.5 directives to move data from device to host

Allocate and delete data on the device

```c
#pragma omp target enter data map(alloc: list-of-data-structures[:])
#pragma omp target exit data map(delete: list-of-data-structures[:])
```

Update data on device and host

```c
#pragma omp target update to/from (list-of-data-structures[:])
to  - HostToDevice
from - DeviceToHost
```

Clauses to use with target directives

```c
map(to:...)   map(from:...)   map(tofrom:....)
```
OpenMP 4.5 directives to offload routines on the device

```c
#pragma omp declare target
void foo();
#pragma omp end declare target
```

Not necessary if routines are inlined
Naive OpenMP 4.5 implementation of GPP

```c
#pragma omp target teams distribute
    map(to:...)
    map(tofrom:output_re[0-2], output_im[0-2])
for(X){
    #pragma omp parallel for
    for(N){
        for(M){
            for(int iw = 0; iw < 3; ++iw){
                //Store local
            }
        }
        for(int iw = 0; iw < 3; ++iw){
            #pragma omp atomic
            output_re[iw] += ...
            #pragma omp atomic
            output_im[iw] += ...
        }
    }
}
```

- Distribute **M-loop** across threadblocks
- Distribute **N-loop** among threads in a threadblocks
- No array reduction with OpenMP 4.5 directives. Hence use atomic to maintain correctness
- Parallelizing **M-loop** increases overhead of synchronization
Optimized implementation with OpenMP 4.5

```c
#pragma omp target enter data
map(alloc:input[0:X])
#pragma omp target update input[0:X])

#pragma omp target teams distribute \ 
parallel for collapse(2) \ 
reduction(+:output_re(0,1,2), output_im(0,1,2))
for(X){
    for(N){
        for(M){
            for(int iw = 0; iw < 3; ++iw){
                //Store local
            }
        }
        output_re(0,1,2) += ...
        output_im(0,1,2) += ...
    }
}
#pragma omp target exit data map(delete:input)
```

- XL, Clang, Cray and GCC gave the best performance with the same parallelization technique
  - Collapse N and M loops and distribute them across threadblocks and threads within a block
- Memory allocation improved the performance of the kernel by 10%
  - `#pragma omp target enter/exit data`
- Reduction gave a $3 \times$ boost in the performance
  - Flatten arrays to scalars
Performance of GPP on V100 with OpenMP 4.5

- Cray is 3× slower than XL
- Clang is 30% slower than XL
- GCC implementation takes 26 seconds
### Table 1: OpenMP 4.5 mapping onto GPU hardware

<table>
<thead>
<tr>
<th>Grid</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>teams distribute</td>
</tr>
<tr>
<td>XL</td>
<td>teams distribute</td>
</tr>
<tr>
<td>Clang</td>
<td>teams distribute</td>
</tr>
<tr>
<td>Cray</td>
<td>teams distribute</td>
</tr>
</tbody>
</table>
#pragma omp target enter data
     map(alloc:input[0:X])

#pragma omp target teams distribute \ 
     parallel for collapse(2) \ 
     map(to:input[0:X]) \ 
     reduction(+:output_re(0,1,2), output_im(0,1,2))
for(X){
    for(N){
        for(M){
            for(int iw = 0; iw < 3; ++iw){
                //Store local
            }
        }
        output_re(0,1,2) += ...
        output_im(0,1,2) += ...
    }
}
#pragma omp target exit data map(delete:input)

- Did not support class operators in older versions.
- Variables passed to the \texttt{reduction} clause should not be passed to any other clause in the same directive.
- All data accessed inside the \texttt{target} region has to be passed via a \texttt{map} clause.
- \texttt{simd} has no effect.
Optimized implementation with Clang

```c
#pragma omp target enter data
    map(alloc:input[0:X])
#pragma omp target update input[0:X])

#pragma omp target teams distribute \\    parallel for collapse(2) \\    map(\text{tofrom:output}_\text{re}(0,1,2), output\_\text{im}(0,1,2)) \\    reduction(+:output\_\text{re}(0,1,2), output\_\text{im}(0,1,2))
for(X){
    for(N){
        for(M){
            for(int iw = 0; iw < 3; ++iw){
                //Store local
            }
        }
        output\_\text{re}(0,1,2) += ...
        output\_\text{im}(0,1,2) += ...
    }
}
#pragma omp target exit data map(delete:input)
```

- Data allocated on the device using OpenMP 4.5 directives need not be passed via `map` clauses.
- Variables passed to the `reduction` clause have to also be passed to `map` clauses.
#pragma omp target enter data
map(alloc:input[0:X])
#pragma omp target update input[0:X])

#pragma omp target teams distribute \ 
    simd collapse(2) \ 
map(tofrom:output_re(0,1,2), output_im(0,1,2))
reduction(+:output_re(0,1,2), output_im(0,1,2))
for(X){
    for(N){
        for(M){
            for(int iw = 0; iw < 3; ++iw){
                //Store local
            }
        }
        output_re(0,1,2) += ...
        output_im(0,1,2) += ...
    }
}
#pragma omp target exit data map(delete:input)

- **parallel for** is executed sequentially inside the **target** region
- **simd** distributes loop across threads of a threadblock
- **reduction** variables have to be passed to the **map** clauses
- Previously allocated data allocated need not be passed via the **map** clauses
- **printf** is not supported inside routines annotated with **declare target**
#pragma omp target enter data
map(alloc:input[0:X])

#pragma omp target teams distribute \  
  parallel for collapse(2) \  
  map(tofrom:output_re(0,1,2), output_im(0,1,2)) \ 
  reduction(+:output_re(0,1,2), output_im(0,1,2))  
for(X){
  for(N){
    for(M){
      for(int iw = 0; iw < 3; ++iw){
        //Store local
      }
      output_re(0,1,2) += ...
      output_im(0,1,2) += ...
    }
  }
}  
#pragma omp target exit data map(delete:input)

- **simd** gives compiler error

- If data is allocated beforehand using **data map (alloc:...)** clauses, they need not be passed to **map** clauses again

- Variables passed to the **reduction** clause have to also be passed to **map** clauses
Cheat Sheet of Do’s and Don’t’s

- **XL**
  - Everything accessed inside the `target` region has to be mapped explicitly via `map` clauses
    - Even if they are allocated on the device beforehand
  - Do not pass the same data to two different clauses in the same directive
    - Even if one of them is a `reduction` clause

- **Clang, GCC, Cray**
  - Always pass the directionality information to the `reduction` variables via `map` clauses

- **GCC** - Do not use `simd`
OpenACC offloading to GPU

- OpenMP
  - Cray
  - XL (IBM)
  - Clang
  - GCC

- OpenACC
  - PGI
  - Cray

- CUDA
OpenACC directive map on GPU

OpenACC

- **gang** — threadblock
- **vector** — Threads in a threadblock
- **worker** — y dimension inside a threadblock (PGI compiler)

OpenMP

- **teams distribute**
- **parallel for**
- **simd**

#pragma acc parallel loop gang

#pragma acc loop vector

#pragma acc loop worker
OpenACC directives for memory movement

#pragma acc enter data copyin
#pragma acc enter data copyout
#pragma acc enter data copy

#pragma acc enter data create(…)
#pragma acc exit data delete(…)
Optimized GPP implementation with PGI OpenACC

```c
#pragma acc enter data create
copyin(input[0:X])
#pragma acc enter data update
device(input[0:X])

#pragma acc parallel loop gang collapse(2)
present(input) \ 
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(int iw = 0; iw < 3; ++iw){
            //Store local
        }
    }
    output_re{0,1,2} += ...
    output_im{0,1,2} += ...
}
```
#pragma acc enter data create copyin(input[0:X])
#pragma acc enter data update device(input[0:X])

#pragma acc parallel loop gang vector collapse(2)
present(input[0:X]) \
reduction(+:output_re(0,1,2), output_im(0,1,2))

for(X){
    for(N){
        for(M){
            for(int iw = 0; iw < 3; ++iw){
                //Store local
            }
        }
        output_re{0,1,2} += ...
        output_im{0,1,2} += ...
    }
}
Cray and PGI implementations of GPP using OpenACC

- Cray is 3× slower than PGI
- Cray is 50% slower than optimized Xeon Phi runtime
Performance comparison of all GPU implementations

- Dashed line is Xeon Phi reference time
- Cray OpenMP and OpenACC give similar performance and is slower than Xeon Phi
- CUDA is 2× faster than the 2nd best implementation
CUDA Implementation of GPP

• 2-dimensional grid for X and N loops

• Distribute M-loop across threads in a threadblock

• CUDA atomics to maintain correctness

```cpp
def gpp_kernel<<<numBlocks, numThreads>>>(X, N):
    for(X)
        for(N)
            for(M)
                for(int iw = 0; iw < 3; ++iw)
                    // Store local
                output_re{0,1,2} += ... // Atomic Add
                output_im{0,1,2} += ... // Atomic Add
```
OpenMP Implementation to match CUDA

OpenMP loop re-reordering to match CUDA implementation

CUDA

```c
for(X){ // blockIdx.x
  for(N){ // blockIdx.y
    for(M){ // threadIdx.x
      for(int iw = 0; iw < 3; ++iw){
        //Store local
      }
    }
    output_re{0,1,2} += ... //Atomic
    output_im{0,1,2} += ... //Atomic
  }
}
```

OpenMP

```c
#pragma omp target teams distribute \
parallel for collapse(2) \
map(to:...) \
reduction(+:output_re0,1,2, output_im0,1,2)
for(N){
  for(X){
    for(M){
      for(int iw = 0; iw < 3; ++iw){
        //Store local
      }
      output_re{0,1,2} += ...
      output_im{0,1,2} += ...
    }
  }
}
```
Performance of GPP implementations after loop reordering

- OpenMP(XL and Clang) are 2× faster after loop re-ordering
- OpenACC(PGI) is 30% faster
- OpenACC(Cray) is 3× faster
- XL and Clang OpenMP similar to optimized CUDA
Performance Portability
Interpretation of OpenMP 4.5 directives on CPU

```
#pragma omp target enter data
    map(alloc:input[0:X])
#pragma omp target update input[0:X])

#pragma omp target teams distribute /
    parallel for collapse(2) /
    map(tofrom:output_re(0,1,2), output_im(0,1,2)) /
    reduction(+:output_re(0,1,2), output_im(0,1,2))
for(N){
    for(X){
        for(M){
            for(int iw = 0; iw < 3; ++iw){
                //Store local
            }
        }
    output_re(0,1,2) += ...
    output_im(0,1,2) += ...
}
#pragma omp target exit data map(delete:input)
```

- **intel/2018 compilers**
- **teams** - creates a single team and associates all threads to that team
  - Reverse the order of `X` and `N` loops and distribute them across threads
- Ignores other OpenMP 4.5 related directives, for example device memory allocation directives
Performance Portability

Performance of GPU implementations on CPU

- GPU optimized OpenMP is 10% slower than optimized Xeon Phi
- CPU optimized OpenMP is $30 \times$ slower on Volta
Summary of the Presentation

- Multiple implementations of OpenMP offloading gave us close to optimized CUDA performance
  - Differences in Compiler interpretations of OpenMP 4.5 offload directives

- Loop reordering might provide benefits due to change in data access patterns

- OpenACC had difficulty in CPU-vectorization

- Portable code but not performance portable