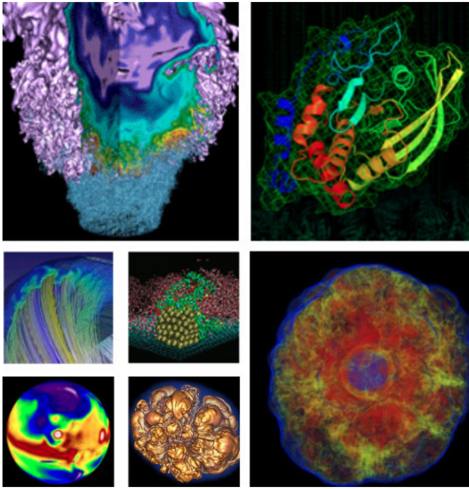
Optimizing Large Reductions in BerkeleyGW on GPUs

Using OpenMP and OpenACC



National Energy Research Scientific Computing Center

Rahulkumar Gayatri, Charlene Yang

National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory

March 8, 2019

✉ rgayatri@lbl.gov, cjyang@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

BerkeleyGW

- 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

```
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

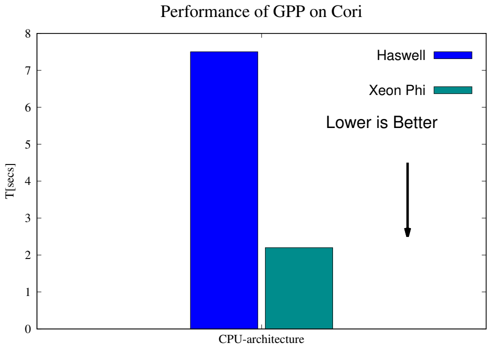
GPP On CPU

OpenMP 3.0 parallelization of GPP

```
#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

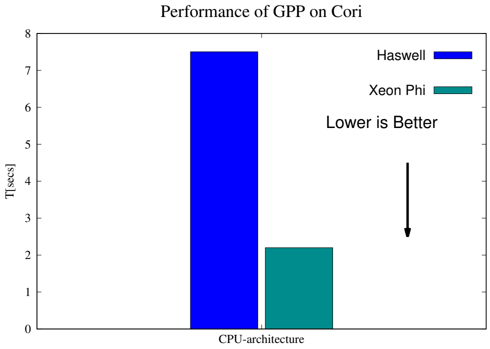
Runtime of GPP on Cori



- 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

Xeon Phi - 2.2 seconds

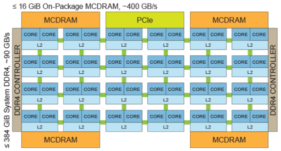


- Performance numbers from Cori at LBNL
  - Haswell
  - Xeon Phi
- intel/2018 compilers
- A perfect scaling would allow a KNL execution to be 4× faster than Haswell
  - KNL implementation of GPP is 3× faster than Haswell

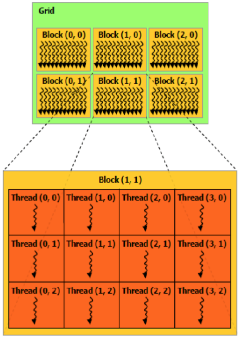
GPP On GPU

GPU Hardware

KNL


≤ 16 GiB On-Package MCDRAM, ~400 GB/s
MCDRAM
PCIe
CORE
L2
DDR4 CONTROLLER
≤ 384 GiB System DDR4, ~90 GB/s

GPU



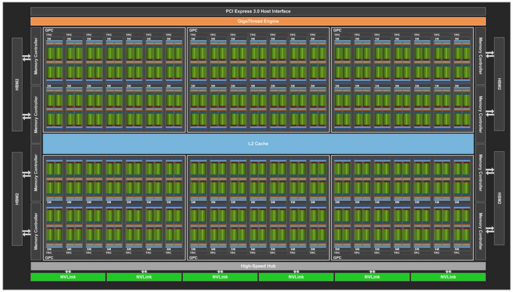
- Going from 272 to 164K threads
- 164k threads
  - 80 SMs
  - 2048 threads within a SM

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

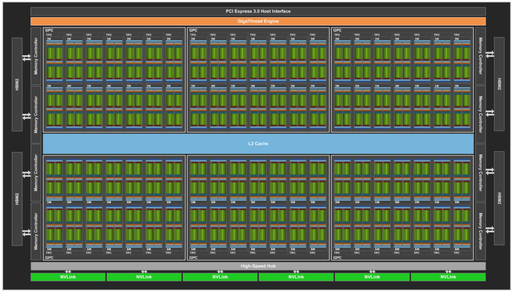


OpenMP offloading to GPU

- **OpenMP 4.5**
  - **Cray**
  - **XL(IBM)**
  - **Clang**
  - **GCC**
- OpenACC
  - PGI
  - Cray
- CUDA

Volta GPU available on Cori and Summit

Target architecture - Volta



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

```
#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

```
#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

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

Clauses to use with **target** directives

```
map(to:...)     map(from:...)     map(tofrom:...)
```

OpenMP 4.5 directives to offload routines on the device

Routines

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

Not necessary if routines are inlined

Naive OpenMP 4.5 implementation of GPP

```
#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

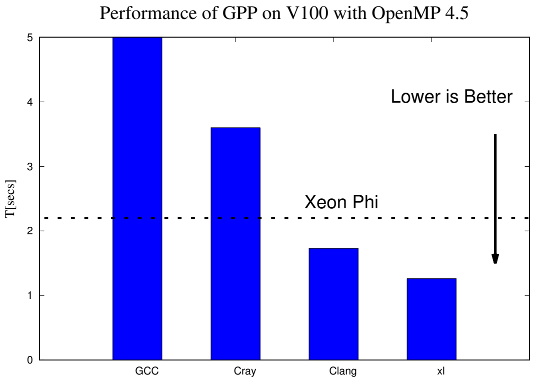
Optimized implementation with OpenMP 4.5

```
#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× 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

OpenMP 4.5 directives map onto hardware

	Grid	**Thread**
GCC	teams distribute	parallel for
XL	teams distribute	parallel for
Clang	teams distribute	parallel for
Cray	teams distribute	simd

Table 1: **OpenMP 4.5 mapping onto GPU hardware**

Optimized implementation with XL

```
#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 **reduction** clause should not be passed to any other clause in the same directive
- All data accessed inside the **target** region has to be passed via a **map** clause
- **simd** has no effect

Optimized implementation with Clang

```
#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(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)
```

- 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

Optimized Cray implementation

```
#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**

Optimized GCC implementation

```
#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

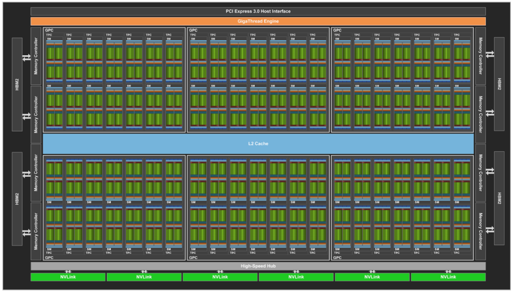
Cheat Sheet of Do's and Dont's

- **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

Target architecture - Volta



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

```
#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} += ...
  }
}
```

- Collapse X and N loops to distribute across threadblocks
- Distribute M loops across threads of a threadblock
- **reduction** required at **gang** and **vector** level since the output variables are updated by every thread.

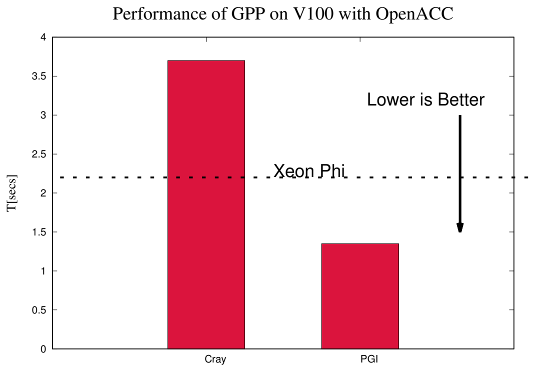
Optimized GPP implementation with Cray OpenACC

```
#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} += ...
  }
}
```

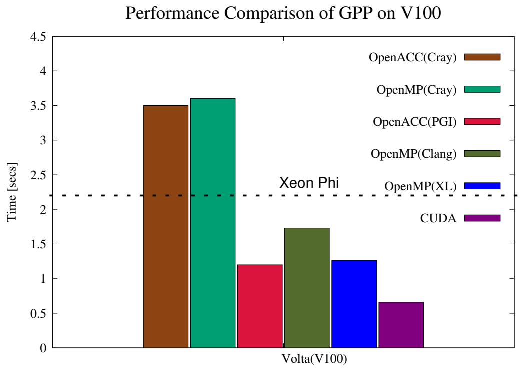
- Collapse Distribute X and N loops to distribute across threadblocks and threads within a block
- Dimensions of the data structures have to be passed to the **present** clause

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$\times$ faster than the 2nd best implementation

CUDA Implementation of GPP

CUDA

```
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
     Add
    output_im{0,1,2} += ... //Atomic
     Add
  }
}
```

- 2-dimensional grid for X and N loops
- Distribute M-loop across threads in a threadblock
- CUDA atomics to maintain correctness

```
dim3 numBlocks(X,N,1);
dim3 numThreads(64,1,1);
gpp_kernel<<<numBlocks, nunThreads>>>;
```

OpenMP loop re-reordering to match CUDA implementation

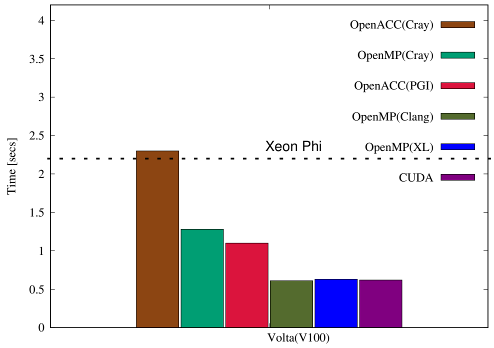
CUDA

```
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

```
#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

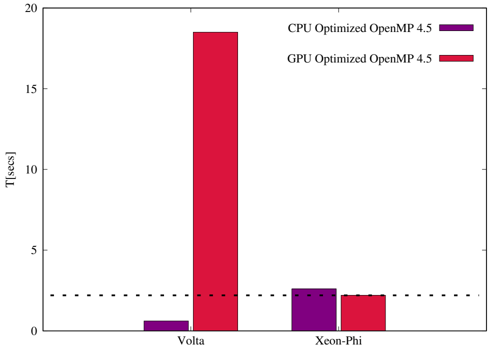
Interpretation of OpenMP 4.5 dierctives on CPU

```
#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 of GPU implementations on CPU



GPU - clang compiler

CPU - intel/2018 compilers

- GPU optimized OpenMP is 10% slower than optimized Xeon Phi
- CPU optimized OpenMP is 30× 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