A Case Study for Performance Portability using OpenMP 4.5

Rahulkumar Gayatri, Charlene Yang, Thorsten Kurth, and Jack Deslippe

National Energy Research Scientific Computing Center (NERSC)
Lawrence Berkeley National Laboratory (LBNL)
Berkeley, CA, USA
{rgayatri, cjyang, tkurth, jrdeslippe}@lbl.gov

Abstract. In recent years, the HPC landscape has shifted away from traditional multi-core CPU systems to energy-efficient architectures, such as many-core CPUs and accelerators like GPUs, to achieve high performance. The goal of performance portability is to enable developers to rapidly produce applications which can run efficiently on a variety of these architectures, with little to no architecture specific code adoptions required. We implement a key kernel from a material science application using OpenMP 3.0, OpenMP 4.5, OpenACC, and CUDA on Intel architectures, Xeon and Xeon Phi, and NVIDIA GPUs, P100 and V100. We will compare the performance of the OpenMP 4.5 implementation with that of the more architecture-specific implementations, examine the performance of the OpenMP 4.5 implementation on CPUs after back-porting, and share our experience optimizing large reduction loops, as well as discuss the latest compiler status for OpenMP 4.5 and OpenACC.

Keywords: OpenMP 3.0 · OpenMP 4.5 · OpenACC · CUDA · Parallel Programming Models · P100 · V100 · Xeon Phi · Haswell.

1 Introduction

The TOP500 list [1] is dominated by systems that employ accelerators and energy-efficient architectures in order to reach their quoted performance numbers. This trend is expected to continue and intensify on the road to exascale, and has increased the emphasis on "X" in "MPI + X", where "X" is an on-node programming framework, which allows for code parallelization over threads and/or vector lanes of a CPU and an accelerator. While "MPI" has established itself as the preferred choice for distributed programming by many, there is not yet a consensus choice for the on-node programming model. There are several options for "X" and they can be loosely categorized into the following.

1. Directive based approaches such as OpenMP, and OpenACC.
2. Architecture specific approaches such as POSIX Threads (pthreads), and CUDA.
3. Abstraction layers of data/task parallelism such as Intel Thread Building Blocks (TBB), OpenCL, Kokkos [2], and Raja [3].

Architecture specific programming models usually require significant code changes and development efforts. The approach of using an abstraction layer for data/-task parallelism on the other hand can add an extra dependency to the code. These models also commonly only support C/C++. In this paper, we focus on the viability of directive based on-node programming models, OpenMP in particular, with performance portability in mind.

OpenMP has been a prevailing programming model for years, especially for first time HPC programmers. Its ease of use and support from major compiler vendors has aided in its adoption as the first step in the parallelization of a sequential code. With version 4.0/4.5, the OpenMP standard has been extended to include support for accelerators. This means that one of the widely used programming models can now support parallelization over heterogeneous architectures via a single framework. That said, the implementation of OpenMP 4.5 by compiler vendors is still at an early stage, which we will have a close look at in this paper.

We investigate porting a relatively simple material science kernel that has been optimized on CPUs using OpenMP 3.0 [7]. We then implement it using OpenMP 4.5 [8] on the GPUs. We will compare the performance of the OpenMP 4.5 implementation with that of its OpenACC [9] counterpart, in terms of their kernel generation capabilities such as registers used and data moved, when different grid and thread dimensions are configured. We will discuss the challenges we faced when implementing the kernel with these frameworks and the techniques we used to improve the performance of each implementation. After an examination of our GPU implementations, we will discuss the performance of the GPU code back on the CPUs and provide an analysis of how portable and more specifically, performance portable it is.

Overall, this paper is structured around the discussions of

1. The optimization strategies for writing OpenMP 3.0/4.5 codes on CPU
2. Early experiences of OpenMP 4.5 on GPUs compared to other options
3. The portability of OpenMP 4.5 codes back on CPU,

with a goal that is two-fold:

1. To demonstrate that a single code can run across multiple (CPU and GPU) architectures using OpenMP 4.5, and
2. To demonstrate that such a code can give an acceptable level of performance compared to the optimized architecture-specific implementations.

The platforms we run on are: the Cori supercomputer [15] at the National Energy Research Scientific Computing Center (NERSC), Lawrence Berkeley National Laboratory (LBNL), for its Intel Haswell and Xeon Phi (Knights Landing (KNL)) architectures, and the Summit supercomputer [16] and the Summitdev testbed at the Oak Ridge Leadership Computing Facility (OLCF), Oak Ridge National Laboratory (ORNL), for their NVIDIA P100 and V100 GPUs, as well as Power CPUs. At the time of writing, **xlc** and **xlc++** from IBM and gcc are the primary compilers which support GPU offloading via OpenMP 4.5 directives on these systems, and we have experimented with both. For OpenACC, the compilers we used are from PGI, the **pgc** and **pgc++** compilers.

The rest of the paper is organized as follows, Section 2 presents a basic introduction of the kernel and the application from which the kernel was extracted. In the same Section, from Subsection 2.2, we present a baseline CPU implementation of the kernel which we will use as a reference for our GPU implementation. Section 3 presents our GPU implementations. In this Section, we will present our experiences with OpenMP4.5 directives and their effective use to optimize performance on a GPU. We compare our OpenMP implementation for GPUs with OpenACC and CUDA. In Section 4 we discuss our efforts in porting the GPU implementations back to the CPU. In Section 6, we talk about our final conclusions and plans for the future.

2 The GPP kernel and its baseline CPU implementation

In this section, we will introduce the General Plasmon Pole (GPP) kernel [6], which is a mini-application representing a single MPI rank's work extracted from a material science code BerkeleyGW [4] [5]. BerkeleyGW itself can be used to compute the excited state properties of complex materials and its main computational bottlenecks are FFTs, dense linear algebra and large reductions, out of which, large reductions can take up 30% of the whole runtime for certain common execution paths. GPP represents the node level work of one of these reductions, and if optimized, can bring significant benefit to the performance of the whole code.

2.1 GPP kernel

The GPP kernel computes the electron self-energy using the General Plasmon Pole approximation. Listing 1.1 shows the most basic pseudo code of this kernel in C++.

Listing 1.1. GPP pseudo code

```
for(X){          // X = 512
    for(N){      // N = 32768/20
        for(M){ // M = 32768
            for(int iw = 0; iw < 3; ++iw){
                Some computation
                output[iw] += ...
            }
        }
    }
}
```

The code was originally written in FORTRAN and employs OpenMP for on-node parallelization. However, in order to apply a large variety of performance portable programming approaches, we created a C++ port for the kernel. The main computational work in the kernel is to perform a series of tensor-contraction-like operations (involving a non-trivial series of multiply, add and divide instructions) for a number of pre-computed complex double-precision arrays, and eventually reduce them to a small 3x3 matrix. The code uses a double-complex number as its primary data type. The problem discussed in this paper consists of 512 electrons and 32768 plane wave basis vectors and corresponds to a medium sized molecule such as Chlorophyll or a small piece of a surface. This choice of size leads to the following characteristics for the kernel:

1. The overall memory footprint is around 2GB.
2. The first and second loop are closely nested and can be collapsed, with a resultant trip count of $\mathcal{O}(800K)$, which could be a target for thread parallelization on the CPUs or threadblock distribution on the GPUs.
3. The third loop has a fairly large trip count too and can be vectorized on the CPUs or parallelized with the threads within a threadblock on the GPUs.
4. The innermost loop has a small, fixed trip count and can be unrolled to facilitate SIMD/SIMT parallelization.

Despite the apparent simplicity, this kernel has a set of very interesting characteristics. For example, the reduction over a series of double-complex arrays that involves multiply add and divide instructions, which are left out of the paper to simplify the discussion. Also, the innermost `iw` loop has significant data-reuse potential whose dimension is problem size dependent (fixed as 3 for our purposes in this paper). For typical calculations, this leads to an arithmetic intensity of the kernel which is between 1-10, which implies that the kernel has to be optimized for both memory locality as well as thread and vectororization efficiency.

2.2 Baseline CPU implementation

The shared memory parallel programming framework OpenMP [7] is a very attractive option for incremental parallelization of codes due to its ease of use and extensive support from compilers. To explicitly address parallelism on hardware of contemporary CPUs, the least version required is OpenMP 3.0, which supports common parallelization paradigms such as vectorization and code transformation features such as loop collapsing, but not the offloading features as in OpenMP 4.x.

Listing 1.2 shows our initial implementation of GPP on CPUs, using OpenMP 3.0, where we spread the "X" loop over threads and "M" loop over SIMD vector lanes. As written, vectorization is automatic by the Intel compiler, without any use of a pragma because the compiler chooses to fully unroll the inermost `iw` loop. In the general case it is necessary to insert an `omp simd` pragma outside the `M` loop.

To represent a complex number, we built an in-house customized complex class that mimics the thrust-complex class available in the CUDA [17] framework.

Complex number reduction is not supported by OpenMP in C/C++ (but it is in Fortran). To work around this, we divide the `output` array into two separate data structures, `output_re[3]` and `output_im[3]`, for their respective real and imaginary components, and apply `reduction` to these data structures. We could potentially have utilized a user defined reduction but experienced performance issues from certain compilers in the past. Similar compiler compatibility issues apply to array reductions in general. To avoid further problems with compiler support for array reduction, in this paper we will always split the reductions up into 6 real-number reductions.

Listing 1.2. GPP + OpenMP 3.0 for CPU

```
#pragma omp parallel for
    reduction(+:output_re[0:3], output_im[0:3])
for(X){
    for(N){
        for(M){
            for(int iw = 0; iw < 3; ++iw){
                //Compute and Store in local variables
            }
        }
        for(int iw = 0; iw < 3; ++iw){
            output_re[iw] += ...
            output_im[iw] += ...
        }
    }
}
```

We execute the code in Listing 1.2 on IBM Power 8 [12], Power 9 [13] and on Intel Haswell [11] and Xeon Phi [10] architectures. As shown in Figure. 1, there is a significant performance difference on the Power processors and Intel architectures. While Power-9 performance is not the focus of this paper, we are investigating the performance gap with Haswell and will add an explanation to the camera-ready paper. Our Xeon- Phi timings for GPP is approximately 2.5 seconds and we use this number as a reference benchmark when porting the application to GPUs.

3 GPU implementations of the GPP kernel

A GPU consists of thousands of cores and they can be abstracted into two layers of parallelization from a programmer's point of view, thread blocks and threads. Thread blocks can form a 1D, 2D or 3D grid, each consisting of the same number of threads within the block. From a hardware perspective, the threads in a thread blocks are also grouped into warps of 32 threads, all executing the same instruction at any time (Volta supports independent thread scheduling). Several warps constitute a thread block, several thread blocks are assigned to a streaming multiprocessor (SM), and several tens of SMs make up the whole GPU card.

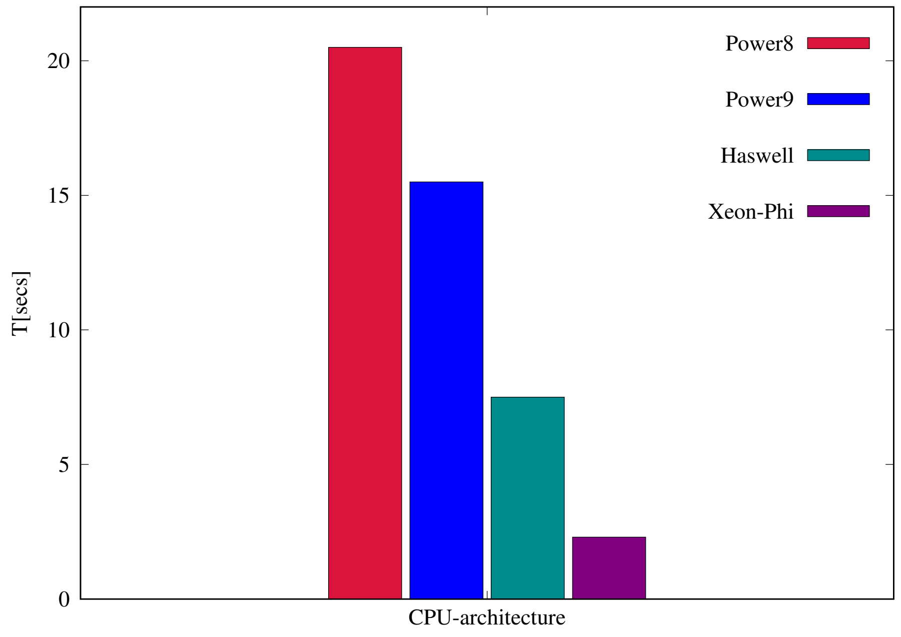

Figure 1. Performance of GPP on multicores

Given the massive parallelism available, it is the programmer's responsibility to match the appropriate data or task parallelism in the code onto the thread blocks or threads within a block on the hardware, in order to take full advantage of the compute power of the card.

We will investigate three programming models on the GPU in this section, OpenMP 4.5, OpenACC and CUDA, with a focus on OpenMP 4.5. We will lay out the compilation configurations and implementation details of GPP using these models, and will also compare them on aspects such as code generation capability, ease of use, compiler support, and code performance.

3.1 Implementation groundwork

In this subsection we describe the settings and software used for compiling and running the GPP kernel. On the Summitdev system only **gcc** and **xl** compilers support accelerator offloading via OpenMP 4.5 (recent versions of clang were not tested), and the compilation flags used for these compilers are shown in Listing 1.3 and Listing 1.4 respectively.

Listing 1.3. Compile flags for **gcc** for OpenMP offloading

```
CXXFLAGS  = -g -O3 -std=c++11 -fopenmp
CXXFLAGS += -foffload=nvptx-none
LINKFLAGS = -fopenmp -foffload=nvptx-none
```

Listing 1.4. Compile flags for **xlc** for OpenMP offloading

```
CXXFLAGS=-O3 -std=gnu++11 -g -qsmp=noauto:omp
CXXFLAGS+=-qoffload #offload target regions to a GPU
CXXFLAGS+=-Xptxas -v #generate report, a CUDA flag
LINKFLAGS+=-qsmp=noauto #disable auto parallelization
LINKFLAGS+=-qsmp=omp -qoffload
```

In our experiments, we observed that the performance of kernels generated by **gcc** compiler was considerably worse than that of the **xl** compilers. Therefore, if not otherwise stated, we will use **xl/20180223-beta** and **xl/20180502** on Summitdev and Summit respectively for all OpenMP 4.5 enabled GPU offloading experiments in this paper. For OpenACC offloading, we employed version 18.4 of the **pgi** compiler available on Cori, SummitDev and Summit machines. **pgi** compile flags for Summitdev and Summit are shown in Listing 1.5.

Listing 1.5. Compile flags for **pgi** for OpenACC offloading

```
CXXFLAGS  = -fast -std=c++11 --gnu_extensions -Munroll
CXXFLAGS += -acc -ta=tesla:cc70 #CUDA kernels for V100
#CXXFLAGS += -acc -ta=tesla:cc60 #CUDA kernels for P100
CXXFLAGS += -Minfo=accel #generate report
LINKFLAGS = -acc -ta=tesla:cc70
#LINKFLAGS = -acc -ta=tesla:cc60
```

3.2 OpenMP 4.5

In order to map the GPP kernel efficiently to the GPU, we need to exploit the different levels of hardware parallelism described above. More precisely, we want to distribute the `X`, `N` and `M` loops across threadblocks and threads within a block in a way that it takes the maximum advantage of the available GPU resources.

In our initial experiments, as a naive CPU parallel programmer, we followed the idea of distributing `X` loop of Listing 1.1 across threadblocks and the `N`, `M` loops across threads within a threadblock. This is shown in Listing 1.6.

Listing 1.6. GPP with OpenMP 4.5 directives

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

The `target` directive on line 1 offloads the code block that follows the directive onto the accelerator. The `teams distribute` directives divide the loop iterations into teams and distribute them among the threadblocks. The `parallel for` on line 4 will distribute the loop iterations among the warps of a threadblock and `simd` directive on line 6 will divide the loop that follows it among the threads in a warp. We inline all function calls from inside the kernel region to avoid additional overhead caused by kernel calls from the device. Array reductions are not supported inside `target` regions by the **xl** compilers and therefore we resorted to our atomic update approach. We furthermore need to manage the data accessed inside the target region. This information is passed to the framework via the `map` clauses and their use is shown in Listing 1.6:

- `map(to:input[0:N])` - copy the values in the data structure to the device at the start of the `target` region.
- `map(tofrom:input-output[0:N])` - copy the values in the data structure to-and-from the device
- `map(from:output[0:N])` - copy the values in the data structure from the device at the end of the `target` region
- `map(alloc:input[0:N])` - A corresponding storage space for `input` is created on the device
- `map(delete:input[0:N])` - Delete the allocated data on the device

Our initial implementation shown in listing 1.6 did not make the optimal use of available resources. It generated a kernel with 1280 threadblocks and 354 threads per block. Even though the `X` loop has only 512 iteration space, **xl** implementation of OpenMP 4.5 directives generated approximately twice the necessary threadblocks. The **xl** compilers distributed the `N` loop following the `parallel for` directive among the threads of a threadblock. Based on the iteration space and assuming all iterations take similar runtime, every thread would execute 4 iterations of the `N` loop and in every iteration the `M` loop is executed sequentially. This implies that the 3^{rd} loop which has an iteration space of O(33K) are not parallelized. After experimentation with different combinations of work distribution Listing 1.7 shows our best implementation (without replacing atomic) of GPP.

Listing 1.7. Optimized GPP with OpenMP 4.5 with atomic

```
#pragma omp target enter data map(alloc:input[0:X]))
#pragma omp target teams distribute parallel for collapse(2)
map(tofrom:output_re[0:3], output_im[0:3])
```

```
for(X){
    for(N){
        for(M){
            for(int iw = 0; iw < 3; ++iw){
                //Store in local variables
                }
        }
        #pragma omp atomic
        output_re* += ...
        #pragma omp atomic
        output_im* += ...
    }
#pragma omp target exit data map(delete:input[0:X]))
}
```

OpenMP provides clauses(`alloc`) to allocate data on the device. As mentioned in Section 2, the memory usage of GPP is approximately 2GB and hence we can allocate all the necessary data on the device. The use of this clause improved the performance of the kernel by 10%, however, the total runtime of the application remained constant. This implies that prior to the usage of `alloc` clause, the kernel time evaluated also included the time taken for data transfers.

In Listing 1.7, we collapse the outer two loops and distribute the resulting iterations among threadblocks and threads within a block. This generates 6552 threadblocks and 128 threads per block. Even in this case all the iterations in the `M` loop are run sequentially by each thread. Distributing them among threads for parallelization increases the number of `atomic` updates relative to the loop iteration space i.e., $\mathcal{O}(33K)$. The benefits of parallelizing the `M` loop are overshadowed by the overhead incurred due to `atomic` updates which are necessary to maintain correctness.

To avoid the use of `atomic` and take advantage of parallelizing the `M` loop, we assign scalar variables to each of the three real and three imaginary components of `output` and pass them into the `reduction` clause. This optimization gave us a performance boost of 3×. `output_re`* and `output_im`* in Listing 1.8 represent these variables.

Listing 1.8. GPP + OpenMP 4.5 with `reduction` for GPU

```
#pragma omp target enter data map(alloc:input[0:X])
#pragma omp target teams distribute parallel for collapse(2)
    reduction(+:output_re*, output_im*)
for(X){
    for(N){
        #pragma omp parallel for
        reduction(+:output_re*, output_im*)
        for(M){
            for(int iw = 0; iw < 3; ++iw){
                //Store in local variables
```

```
                  }

             output_re* += ...
             output_im* += ...
         }

     }
#pragma omp target exit data map(delete:input[0:X])
}
```

Listing 1.8 shows the pseudo code for our most optimized implementation of GPP via OpenMP 4.5 directives. In this code we collapse the outer two loops, i.e., `X` and `N` and distribute them over threadblocks while the `M` loop is parallelized over the threads in a thread block. Since the values are updated inside the `M` loop we have a **`reduction`** clause with the **`teams distribute`** and **`parallel for`** directives. Unlike Listing 1.6 where **`output_re`** and **`output_im`** are passed to the **`map(from:...)`** clause, variables passed into the **`reduction`** clause need not be passed in any other clauses. Listing 1.8 generates 1280 threadblocks and 512 threads per block.

OpenMP 4.5 also provides clauses to control the grid and thread dimension generated by the framework. A programmer can use **`num_teams`** and **`thread_limit`** clauses to inform the framework about the number of threadblocks and threads per block with which the CUDA kernel should be launched. In our case we realized that the default kernel dimensions chosen by the compiler were optimal. Figure 2 shows the performance comparison between **`atomic`** and **`reduction`** on P100 and V100. Both the implementations use different parallelization techniques as shown in Listing 1.7 and 1.8 respectively. This implies that the use of **`atomic`** or **`reduction`** with **xl** compilers to maintain correctness might lead to different optimal parallelization strategies on a GPU.

xl vs gcc implementation of OpenMP 4.5 Although, in the disucssion above and for the rest of the paper, we focus on **xl** implementation of OpenMP 4.5, we were also successful in porting GPP with **gcc** compiler using the accelerator directives. In this section we present three major differences between the compiler implementations of OpenMP 4.5 directives that complicated the use of OpenMP 4.5 when targetting a code that is intended to support multiple compilers.

1. **`simd`** in the case of **xl** compilers is optional but mandatory for **gcc** to make use of all the threads in a warp.
2. The use of **`map`** clauses is mandatory for **xl** compilers. Every memory location accessed inside **`target`** region has to pass through one of the directionality clauses. In case of **gcc**, this condition is not enforced.
3. In practice it has been our observation that dynamic allocation of data structures inside the **`target`** directives fail with the **xl** compilers. This constraint is not applicable to **gcc** compilers.

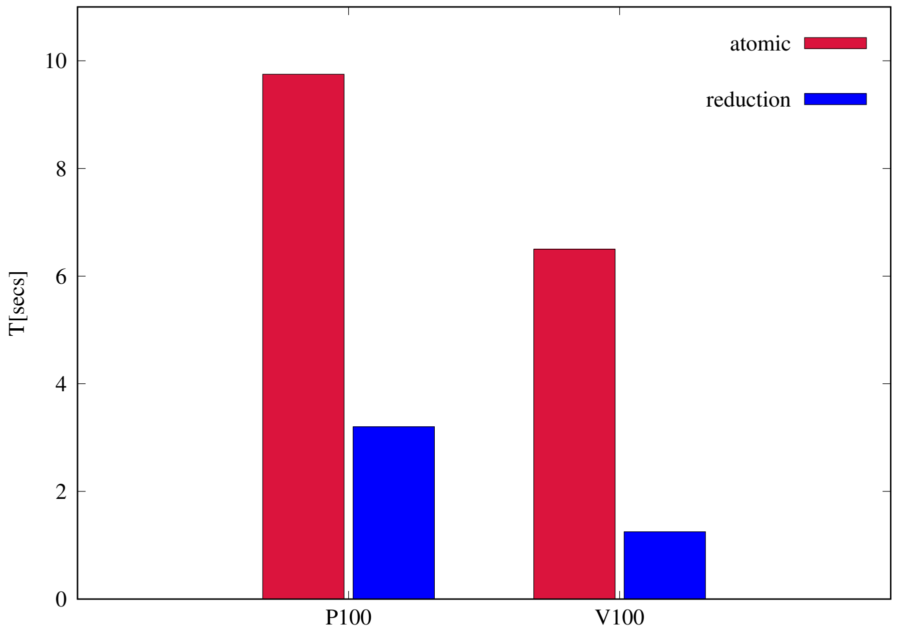

Figure 2. `atomic` vs. `reduction` clauses for OpenMP 4.5 directives on P100 GPU.

3.3 OpenACC

Similar to OpenMP 4.5, OpenACC has its own directives to distribute loops across the threads of a GPU. OpenACC directives for work distribution across GPU threadblock and threads are `gang` and `vector` respectively.

The experiences gained in OpenMP offloading experiments helped us in tuning the OpenACC implementations. The best performance of GPP with OpenACC directives was achieved with the `reduction` implementation of OpenACC, similar to OpenMP 4.5 as shown in Listing 1.9.

Listing 1.9. GPP + OpenACC for GPU

```
#pragma enter data create(input[0:X]))
#pragma acc parallel loop gang collapse(2)
    present(input[0:X]))
    reduction(+:output_re*, output_im*)
for(X){
    for(N){
    #pragma acc loop vector\
    #reduction(+:output_re*, output_im*)
        for(M){
            for(int iw = 0; iw < 3; ++iw){
                //Store in local variables
```

```
                    }
          }
          output_re* += ...
          output_im* += ...
     }
#pragma exit data delete(input[0:X])
}
```

In Listing 1.9, the directives in line 2 collapse the X and N loop and distribute them among the threadblocks, while the directives in line 7 distribute the M loop among the threads in a threadblock.

The `reduction` versions of OpenACC and OpenMP 4.5 give equivalent performance. However the **pgi** compiler generates 65535 threadblocks and 128 threads per block for this parallelization which is significantly different than the 1280 threadblocks and 512 threads per block generated by the **xl** compiler in response to the OpenMP offload directives. Section 3.5 discusses the reasons for similar performance with different kernel configuration in greater detail.

The optimal `atomic` version with OpenACC occurs when we distribute the X loop among the threadblocks and N loop among the threads per block. For this version the compiler generated a kernel with 512 threadblocks and 128 threads per block and its performance is 2× faster than the `atomic` version of OpenMP 4.5. However, when we back-ported these changes to the **xl** implementations of OpenMP 4.5, we were unable to replicate the performance.

Line 1 and line 15 of Listing 1.9. are the data allocation directives of OpenACC. The `present` directive in line 3 informs the compiler that the data passed to this clause is available on the device. Otherwise `copyin` and `copyout` clauses are necessary to map the necessary data on-to the device. During our OpenACC implementation, we learned that the **pgi** compiler does not copy the data on to the device when encountered with the `data create` directives. The actual copy occurs when the corresponding data is encountered inside the kernel. In order to overcome this issue, we initialized the data on the device to guarantee its availability during kernel launch. This optimization was performed in order to avoid the inclusion of memory transfer time in kernel computation timing.

Figure 3 shows the comparison of atomic versus reduction versions of GPP on both the GPU architectures. The atomic version in the case of OpenACC is only 5 % slower than the `reduction` version which is significantly lower than the difference between similar implementations of OpenMP 4.5.

3.4 CUDA

CUDA [17] is an extension of the C and C++ programming language, developed by NVIDIA to offload parallel kernels onto a GPU. In version 1, we implement a single dimension grid with the X loop being distributed across the threadblocks and N loop between the threads of a threadblock. In version 2 we generate a two dimensional grid. The 1st dimension is the outermost X loop and 2nd dimension is the N loop. The innermost M loop is distributed among the threads of a threadblock. Both the kernel launch parameters are shown in Listing 1.10.

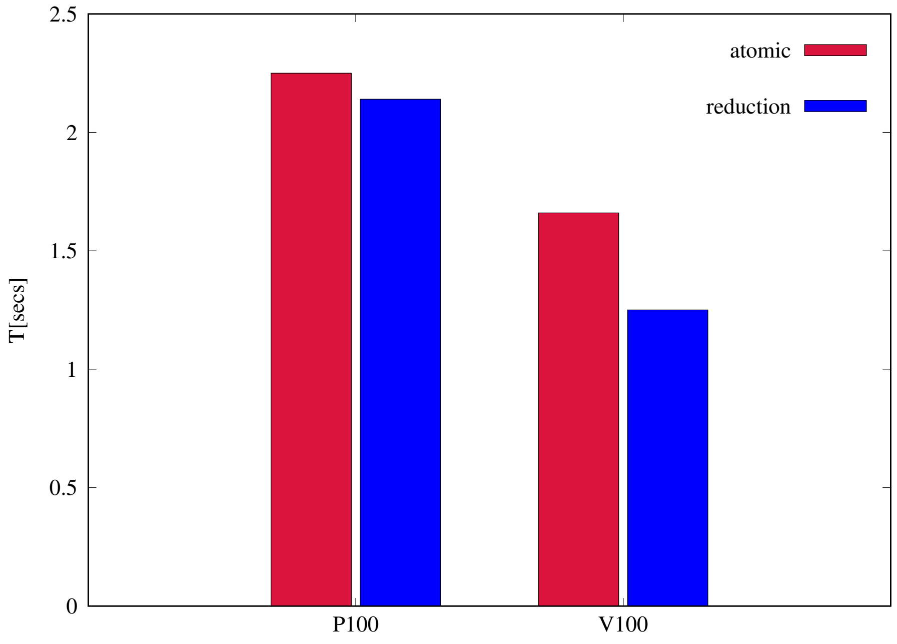

Figure 3. `atomic` versus `reduction` in OpenACC on P100 and V100

Listing 1.10. Kernel parameters for CUDA version 1 and version 2

```
dim3 numBlocks(X,1,1);
dim3 numThreadsPerBlock(64,1,1);

//Version 2
dim3 numBlocks(X,N,1);
dim3 numThreadsPerBlock(32,1,1);

//Kernel Launch
gpp_Kernel<<<numBlocks, numThreadsPerBlock>>> (...);
```

We launch the kernels with 64 and 32 threads per threadblock for version 1 and version 2 respectively. In our experiments these proved to be the ideal kernel launch parameters for the respective versions. We use the `atomcAdd` routine in CUDA to maintain correctness of our updates.

While version 1 gave similar performance as the corresponding OpenACC `atomic` implementation, version 2 was approximately 2× faster compared to version 1. But as shown in Figure 3, the difference between the `atomic` and `reduction` implementations of OpenACC was only 5%. This shows that the benefits of CUDA version 2 implementation were unavailable in the corresponding OpenACC or the OpenMP 4.5 versions which give similar performance.

3.5 Performance comparison among GPU implementations

In this section we perform a detailed comparison among the available GPU implementations. We specifically focus on the differences between OpenMP 4.5 and OpenACC implementations of GPP.

OpenMP 4.5 vs OpenACC The optimized `reduction` version of OpenMP and OpenACC collapses the `X` and `N` loops and distributes them across threadblocks whereas the `M` loop is parallelized over the threads. Table 1 presents a comparison of the kernels generated by both these versions.

V100	runtime	grid-dim	thread-dim	registers
OpenACC(pgi/18.4)	1.24	(65535,1,1)	(128,1,1)	136
OpenMP(xlc/20180502)	1.25	(1280,1,1)	(512,1,1)	114

Table 1. OpenACC vs OpenMP 4.5 kernel configuration on V100 reduction version

Collapsing of `X` and `N` loops generates $\mathcal{O}(800K)$ iterations that can be distributed across the available threadblocks. From the details presented in Table 1, we observe that even though OpenACC generates $50\times$ more threadblocks (and 4x fewer threads per block) than OpenMP, both the frameworks give us approximately the same runtime. A summary of the hardware metrics and their comparison is shown in Table 2. While dram-utilization and warp-efficiency in both the implementations are similar, OpenMP has a 30% higher global-hit-rate, i.e., hit rate for global load and store in L1 cache and a somewhat higher occupancy, i.e., the ratio of active warps to the maximum number of warps per cycle. We expect on OpenACC the latency of the misses is effectively hidden by the additional overall threads available and there is a high enough arithmetic intensity to avoid saturating memory bandwidth.

V100	dram-utilization	global-hit-rate	Warp-efficiency	occupancy
OpenACC(pgi/18.4)	8 (high)	54.05%	99.92%	0.19
OpenMP(xlc/20180502)	7 (high)	84.6	99.86%	0.27

Table 2. OpenACC vs OpenMP 4.5 kernel configuration on V100 reduction version

GPP performance on contemporary GPUs In this section we perform a general comparison among all the available GPU implementations of the kernel.

The horizontal dash line in Figure 4 represents the performance of GPP on Xeon Phi against which we compare our GPU implementations. As observed the Figure 4, apart from the OpenMP 4.5 version on P100, all other implementations perform better than OpenMP 3.0 implementation on Xeon Phi. OpenMP 4.5 in

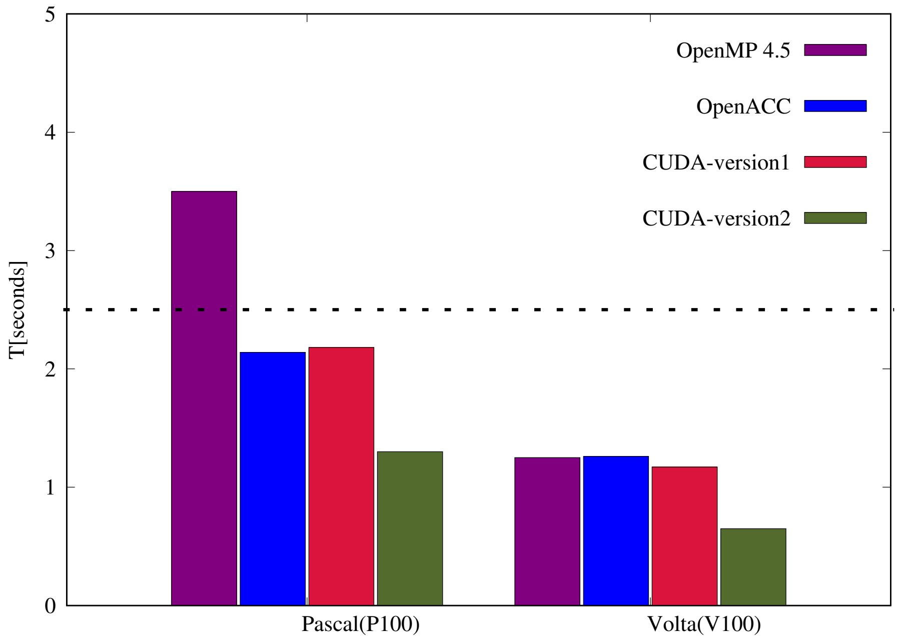

Figure 4. Performance on P100 and V100

particular, shows a drastic improvement in its performance relative to other implementations on Volta compared to Pascal. The main reason for this is the use of new compiler on the Summit machine, which is unavailable on Summitdev. This shows the importance of compiler maturity in generating optimal CUDA kernels via the offload directives.

As mentioned in Section 3.4, our CUDA version 2 implementation of GPP is similar to the `reduction` version of OpenMP and OpenACC, however its runtime is approximately 2× faster. CUDA version 2 generates a grid of `(512,1638,1)` with 32 threads per threadblock, i.e., 50× more than OpenACC. Additionally its dram-utilization relative to peak utilization is 9, which is higher than OpenMP and OpenACC implementations, 7 and 8 respectively. Back-porting the kernel configuration onto OpenACC and OpenMP implementations using the clauses provided did not result in the same performance benefits. Additionally the GPU occupancy in case of CUDA version 2 is approximately 0.5 which is higher compared to 0.27 and 0.19 for OpenMP and OpenACC respectively.

Another important observation in Figure 4 is the approximately 2× difference between P100 and V100 performance. We also want to assert that the code implementations were consistent for every framework on both the GPUs. Further investigation is required (to be resolved before camera-ready date), but we are initially attributing the faster performance on Volta GPU to the following: 1) Compared to Pascal, Volta has a bigger and faster L1 cache; 2) global atomic is 2× faster, which may be important for reductions in GPP; 3) lower instruction latency.

4 Porting GPU implementations back to CPU

As mentioned in Introduction of this paper, our aim is to evaluate the status of the current programming frameworks and their ability to create a single code implementation that can be executed across architectures. In this section we discuss the performance of OpenMP and OpenACC GPU implementations on CPUs, especially Intel's Haswell and Xeon Phi.

4.1 OpenACC

Initially, **pgi** compilers were unable to parallelize GPP loops annotated with OpenACC directives on CPUs. The compiler appeared to assume dependencies between variables declared inside the loops which should technically be considered thread-private. The **pgi** compiler as a part of its aggressive optimization hoists variables declared inside the loop. This implies that the OpenACC directives would annotate these variables as shared, if stated otherwise, and prevent the parallelization of these loops. To avoid these problems, we declared the said variables outside the loops and marked them `private` to avoid dependency assumptions by the compiler.

Even after the said changes, **pgi** compiler was unable to vectorize the code for CPUs. Hence OpenACC implementation on CPUs for GPP is 4× slower than the OpenMP implementation.

4.2 OpenMP 4.5

For the compilation of OpenMP 4.5 on Haswell and Xeon Phi, we used the `intel/18.0.1.163` compiler on the Cori machine. Listing 1.11, shows the flags used in order to compile the code on CPUs.

Listing 1.11. Intel flags for OpenMP 4.5

```
    CXXFLAGS=-O3 -std=c++11 -qopenmp -qopt-report=5
    CXXFLAGS+=-qopenmp-offload=host #For offloading
    #CXXFLAGS+=-xCORE-AVX2 #For Haswell
    CXXFLAGS+=-xMIC-AVX512 #For Xeon Phi
    LINKFLAGS=-qopenmp
    LINKFLAGS+=-qopenmp-offload=mic #For Xeon Phi
```

Intel compiler, when encountered the with the `target teams` directive of OpenMP 4.5 on a CPU, generates a single team and assigns all the threads available to the team. Hence when the corresponding `parallel for` is encountered, the loop following the directive is parallelized among the available threads. In case of our best OpenMP 4.5 `reduction` implementation, this translates to the outermost `X` and `N` loop running sequentially, but the innermost `M` loop is distributed among the threads. In the case of GPP, this interpretation of OpenMP offload directives lead to a "portable" but not "performance portable" code since such a parallelization increases the GPP runtime by 25× compared to the optimized OpenMP 3.0 implementation. In order to optimize GPP with OpenMP 4.5 on CPUs, we modified the implementation by moving the `parallel for` directives on `X` loop as shown in Listing 1.12.

Listing 1.12. GPP + OpenMP 4.5 on CPU

```
#pragma omp target teams distribute parallel for
    reduction(+:output_re*, output_im*)
for(X){
    for(N){
        for(M){
            for(iw = 0; iw < 3; ++iw)
                {//Store in local variables}
        }
        output_re* += ...
        output_im* += ...
    }
}
```

This creates a team of 272 or 64 threads for Xeon Phi or Haswell and distributes the `X` loop across the available threads. This is similar to the OpenMP 3.0 implementation of GPP.

Figure 5 shows a comparison of executing optimal GPU and CPU implementations with OpenMP 4.5 directives on Xeon Phi and Volta. As can be observed the performance of CPU optimized OpenMP 4.5 implementation is similar to the optimized OpenMP 3.0 runtime. However, on GPU the implementation is 5× slower than the optimized OpenMP 4.5 implementation for GPUs. Conversely, the optimized GPU implementation with OpenMP 4.5 is 25× slower than the optimized implementation on Xeon Phi. This experiment shows that the assumption by the **intel** compilers where they equate the threads on a CPU and threads in a GPU's threadblock did not result in a "performance portable" code in the case of GPP .

5 Related Work

Since OpenMP 4.5 began gaining traction with compilers, there have been a number studies performed in the community (we discuss a number below) to compare its performance and cost of implementation to other GPU programming

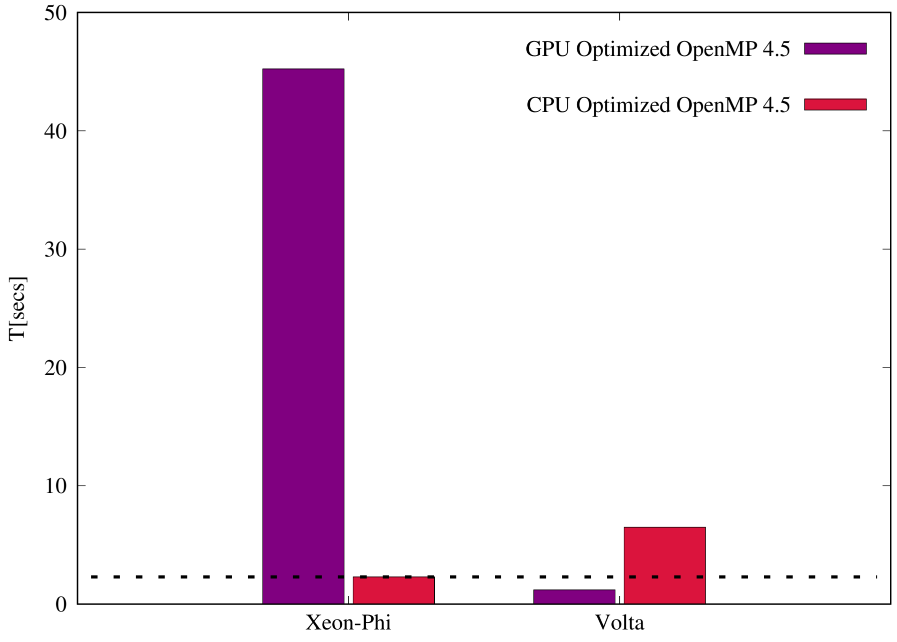

Figure 5. OpenMP 4.5 on CPU

models. Our effort is intended to be complimentary to these published works, represent a specific case-study of importance to the Material Science community and document a snapshot of the current state of support under rapid OpenMP 4.5 development and optimization over the past few years.

One of the early works to evaluate OpenMP 4.0 on Xeon Phi and GPUs is published in [20]. In this they chose a kernel representative of regular workloads on Titan and attempt to port it across widely used HPC architectures such as CPU, CPU+accelerator and self hosted coprocessor using OpenMP 4.0. In [19], the authors evaluate and analyze OpenMP 4.X benchmarks on Power8 and NVIDIA Tesla K80 platform. They perform an analysis of hand written CUDA code and the automatic GPU code generated using IBM xl compilers and clang/LLVM compilers from high level OpenMP 4.x programs. Our work differs from the paper in two major areas: 1) The kernel we ported is more complicated and uses a template class to represent a complex number, and; 2) We back-port the GPU implementations of OpenMP 4.5 onto CPUs.

In [18], the authors evaluate the "state of the art" for achieving performance portability using compiler directives. The paper performs an in-depth analysis of the how OpenMP 4.0 model performs on K20X GPU and in Xeon Phi architectures for commonly used kernel such as "daxpy" and "dgemm". However unlike this paper, they do not discuss the kernel configurations generated by the frameworks and their impact on the various parallel-loops inside the kernel.

6 Summary and future work

In this paper, we presented an analysis on the effort needed to port a kernel onto CPUs and GPUs using OpenMP in comparison to other approaches with focus on OpenACC. We discussed the configurations of the kernels generated and the methods to tune them in order to optimize the use of available hardware resources.

We were successful in porting our best implementation of OpenMP 4.5 onto CPUs with some important changes to the implementation. The performance of this version is equivalent to our best OpenMP 3.0 version. But, an exactly unchanged OpenMP 4.5 version optimized for GPUs is ill-suited for CPU execution.

OpenACC and OpenMP, both lack the ability to fully generate a customized multidimensional grid and threads for GPUs. While this drawback can be overcome by flattening the loop via the `collapse` clause, our experiments have shown that the performance of such an optimization might still be slower than the CUDA 2-dimensional grid, where we achieved our best performance result.

6.1 Future work

The memory footprint of this kernel is approximately 2GB and hence can be completely allocated in the HBM of P100 and V100. In the future we want to evaluate the practical amount of work required to port kernels which exceed the memory space that can be allocated on the device. Although there are plans to include UVM support from OpenMP 5 version, the **xlc** compilers allow passing of device pointers to the framework. The memory on the device could be allocated via `cudaManagedMalloc` and the corresponding pointer would be passed to the OpenMP 4.5 via `is_device_ptr` clause - allowing such studies to be done today.

7 Acknowledgement

This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research also used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

8 Reproducibility

Due to the nature and the content of our work, we have included the information about the hardware and software frameworks associated with our results in the paper. The authors will provide full access to the GPP kernel and its implementations on GitHub if this paper is accepted for publication.

References

1. TOP500 Supercomputers list, `https://www.top500.org/lists/2018/06/`.
2. H. C. Edwards and C. R. Trott and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns", Journal of Parallel and Distributed Computing, 2014.
3. R. D. Hornung and J. A. Keasler, "The RAJA Poratability Layer: Overview and Status", Tech Report, LLNL-TR-661403, Sep. 2014.
4. J. Deslippe, G. Samsonidze, D. A. Strubbe, M. Jain, M. L. Cohen, and S. G. Louie. "BerkeleyGW: A massively parallel computer package for the calculation of the quasiparticle and optical properties of materials and nanostructures". Computer Physics Communications, Volume 183, Issue 6, June 2012, Pages 1269-1289
5. BerkeleyGW Code. `https://berkeleygw.org`
6. J. Soininen, J. Rehr, and E. Shirley, "Electron self-energy calculation using a general multi-pole approximation", Journal of Physics: Condensed Matter, 15(17), 2003.
7. https://www.openmp.org/
8. https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf
9. https://www.openacc.org/
10. Intel Knights Landing Processor. `https://ark.intel.com/products/94034/Intel-Xeon-Phi-Processor-7230-16GB-1_30-GHz-64-cor`
11. Intel Haswell Processor. "Haswell: The Fourth-Generation Intel Core Processor", in IEEE Micro, vol. 34, no. 2, pp. 6-20, Mar.-Apr. 2014.
12. B. Sinharoy et al., "IBM POWER8 processor core microarchitecture", in IBM Journal of Research and Development, vol. 59, no. 1, pp. 2:1-2:21, Jan.-Feb. 2015.
13. S. K. Sadasivam, B. W. Thompto, R. Kalla and W. J. Starke, "IBM Power9 Processor Architecture", in IEEE Micro, vol. 37, no. 2, pp. 40-51, Mar.-Apr. 2017.
14. NVIDIA V100 GPU Whitepaper. `http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`
15. `http://www.nersc.gov/users/computational-systems/cori/`
16. `https://www.olcf.ornl.gov/summit/`
17. J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA", Queue 6, 2 (March 2008), 40-53. DOI: https://doi.org/10.1145/1365490.1365500
18. M. G. Lopez, L. V. Vergara, W. Joubert, O. Hernandez, A. Haidar, S. Tomov, and J. Dongarra, "Towards achieving performance portability using directives for accelerators", 2016 Third Workshop on Accelerator Programming Using Directives (WACCPD).
19. A. Hayashi, J. Shirako, E. Tiotto, R. Ho, and V. Sarkar, "Exploring Compiler Optimization Opportunities for the OpenMP 4.× Accelerator Model on a POWER8+GPU Platform", 2016 Third Workshop on Accelerator Programming Using Directives (WACCPD).
20. L. V. G. Vergara, J. Wayne, M. G. Lopez and O. Hernández, "Early Experiences Writing Performance Portable OpenMP 4 Codes", Proc. Cray User Group Meeting, London, England. Cray User Group Incorporated, May 2016.