Comparing High Performance Computing Accelerator Programming Models

Swaroop Pophale(✉), Swen Boehm, and Verónica G. Vergara Larrea

Oak Ridge National Laboratory, Oak Ridge, USA
{pophaless,boehms,vergaravg}@ornl.gov

Abstract. Accelerator devices are becoming a norm in High Performance Computing (HPC). With more systems opting for heterogeneous architectures, portable programming models like OpenMP and OpenACC are becoming increasingly important. The SPEC ACCEL 1.2 benchmark suite consists of comparable benchmarks in OpenCL, OpenMP 4.5, and OpenACC 2.5 that can be used to evaluate the performance and support for programming models and frameworks on heterogeneous platforms. In this paper we go beneath the normative metric of performance times and look at the individual kernels to study the usage, strengths, and weaknesses of the two prevalent portable heterogeneous programming models, OpenMP and OpenACC. From our analysis we identify that benchmarks like MRI-Q, SP and BT have better performance using OpenACC, while benchmarks like MiniGhost, LBM and LBDC do consistently better with the OpenMP programming model across super-computers like Titan, and Summit. We deep dive into the kernels of select four benchmarks to answer questions like: Where does the benchmark spend most of its cycles? What is the parallelization strategy used? Why is one programming model more performant than the other? By identifying the similarities and differences we want to contrast between the benchmark implementation strategies in the SPEC ACCEL 1.2 benchmarks and provide more insights into the OpenMP and OpenACC programming models.

1 Introduction

The SPEC ACCEL benchmarks are written and maintained by members of Standard Performance Corporation (SPEC) High Performance Group (HPG) and are written in a performance portable manner. The SPEC ACCEL 1.2 suite

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

M. Weiland et al. (Eds.): ISC 2019 Workshops, LNCS 11887, pp. 155–168, 2019.
https://doi.org/10.1007/978-3-030-34356-9_14

includes a collection of benchmarks that cover a variety of common HPC algorithms. SPEC ACCEL consists of 19 OpenCL benchmarks that are based on the Parboil Benchmark (University of Illinois at Urbana-Champaign) and the Rodinia benchmark (University of Virginia) and 15 benchmarks for OpenMP 4.5 and OpenACC 2.5, that are based on the NAS Parallel benchmarks, SPEC OMP 2012 and benchmarks derived from HPC applications. The benchmarks in the suite can provide insights about the quality of different implementations of the OpenMP and OpenACC compilers and runtime environments. We have tested them to evaluate the extent of support available for new OpenMP 4.5 features on leadership computing systems like Titan [7] and Summit [2]. Comparing performance portability across different architectures and implementations provides insight to the application programmers/users as to the readiness of the systems. This is especially true for Summit where the implementations are still under development. Although programming models like OpenMP are designed to be platform agnostic, architectural differences can have a profound effect on performance. Users can then compare functionality and performance across a range of architectures and implementations of OpenMP and OpenACC.

In this paper, we document results from running the SPEC ACCEL 1.2 benchmark suite on Titan and Summit to see the current status of support and performance afforded by current OpenMP and OpenACC implementations. We perform experiments to capture the changing landscape of OpenMP 4.5 support and look deeper into the specific kernels that are the key performance bottlenecks. We also take a closer look at that subset of SPEC ACCEL benchmark kernels to determine which factors account for the performance difference. We look at the performance profiles and focus on the kernels/sub-routines that take the most time. Understanding the different strategies used by OpenMP and OpenACC is an exercise in finding equivalence, analyzing productivity and understanding the level of user intervention required to gain most of the benefits afforded by the programming model.

2 Motivation

In this study, we look into the different benchmark kernels with the objective of highlighting and investigating the differences and similarities between the two programming models, OpenMP and OpenACC. Fundamentally, OpenMP has been identified as prescriptive while OpenACC claims to be descriptive in their approach. Prescriptive model of programming requires very tight semantics and implementations must provide the exact behavior promised. While descriptive models describe the objective and leave more room for the implementations to work towards this objective. Looking at the benchmark kernels allows us to investigate real cases and analyze if the differences stemming from the specification are only in the semantics or the actual implementations. If a lot of implementation defined features are in play, the behavior of the kernels and the performance changes accordingly. For example, the maximum number of threads created per team is implementation defined in OpenMP. The user has the option to specify

Table 1. Successes and failures of running the SPEC ACCEL 1.2 benchmarks on different architectures with OpenMP 4.5 and OpenACC. The compiler versions used are: On Summit: PGI 18.3, XL V16.1.0, Clang/LLVM (ykt branch), GCC 7.2 (gomp branch), on Titan Cray CCE 8.7.0, PGI 18.4

	Summit (NV100 GPU)					Titan (K20X GPU)	
	XL	PGI	GCC		Clang	PGI	CCE
	OMP	ACC	ACC	OMP	OMP	ACC	OMP
Stencil	✓	✓	✓	✓	✓	✓	✓
LBM	✓	✓	✓	✓	✓	✓	✓
MRI-Q	✓	✓	✓	✗RE	✓	✓	✓
MD	✓	✓	✓	✗RE		✓	✗RE
PALM	✗RE	✓	✓	✗RE		✗CE	✗CE
EP	✓	✓	✓	✗VE	✓	✓	✓
CLVRLEAF	✓	✓	✓	✗RE		✓	✗VE
CG	✓	✓	✓	✗VE	✗RE	✓	✓
SEISMIC	✓	✓	✓	✗RE		✓	✗RE
SP F	✓	✓	✓	✗RE		✓	✗RE
C	✓	✓	✓	✗RE	✓	✓	✓
MiniGhost	✓	✓	✓	✗RE		✗CE	✗RE
LBDC	✓	✓	✓	✓		✓	✗RE
Swim	✓	✓	✓	✗RE		✓	✗RE
BT	✓	✓	✓	✗RE	✓	✓	✓
Passed	14	15	15[a]	3	6	13	7

[a]GCC/OpenACC only offloads 4 out of the 15 benchmarks, the remaining 11 benchmarks utilize the CPU.
VE: Verification error
RE: Runtime error
CE: compile error

a `thread_limit` clause that gives an upper bound to the implementation defined value for the number of threads per team. A user can request a given number of threads for a parallel region via the `num_threads` clause. Another example of an implementation dependent behavior can be observed in the LLVM compiler, which defaults to `schedule(static,1)` for the parallel loops when executed inside a target region that is offloaded to a GPU.

On Summit, the world's fastest supercomputer [8], vendors are still in the process of providing full support for the OpenMP 4.5 programming model. Through this work we want to also provide a temporal snapshot of the programming models support on Summit. Table 1 shows the number of benchmarks that compile and execute correctly with different OpenMP and OpenACC implementations. Figure 1 compares the best performance time for OpenACC vs. OpenMP on Summit and Titan with latest versions of the OpenMP implementations from IBM.

Fig. 1. OpenMP's performance improvement over OpenACC.

As it happens, there is not a single vendor or compiler implementation that provides both OpenMP and OpenACC implementation with the same degree of success and, as such, the comparisons across different vendors may, at first sight, seem unfair. But it is our experience that applications will choose the fastest implementation and in that respect comparing the best of OpenMP and OpenACC gives a fair assessment, as we expect these implementations on the same platform to have exploited similar architectural features.

For this work the relative speed up is calculated by dividing the best OpenACC timing by the best OpenMP for individual benchmarks on a particular platform. The benchmarks scoring above the threshold line (at 1) indicate better performance with OpenMP programming model, while those scoring in the negative Y axis direction indicate that they perform better with OpenACC programming model. For Titan we use PGI's OpenACC and Cray's OpenMP implementations while for Summit (Power9 + NVIDIA V100 GPU) we compare PGI's OpenACC 2.5 with XL's OpenMP 4.5.

We see that the MRI-Q, SP (C version) and BT benchmark have better performance using OpenACC, while the LBM, MiniGhost, and LBDC benchmark do consistently better with the OpenMP programming model across Titan and Summit. Based on the analysis in Fig. 1 we take a more detailed look into benchmarks BT, SP, LBM, and LBDC as they show distinct and pronounced performance advantage with one of the programming models.

3 Related Work

Previous work has compared the performance of the SPEC ACCEL benchmark suite codes when using different programming models including OpenCL, OpenACC, and OpenMP 4.x. In [4], the three different programming models are used to compare performance of OpenACC on two different GPU devices, and OpenMP on the Intel Xeon Phi coprocessor. At the time, only the Intel compiler provided support for the OpenMP 4.0 accelerator model. Since then,

GNU, LLVM, and XL compilers have added support for this model. In addition, the PGI compiler has added support to self-offload using OpenACC which has enabled testing of the PGI compiler on Intel Xeon Phi based architectures.

In [5], Juckeland et al., provide a detailed overview of the effort required to port the SPEC ACCEL benchmark suite from the OpenACC programming model to the OpenMP 4.5 accelerator programming model. The work highlights the differences between each programming model. For example, in OpenACC, the developer can briefly describe the intended parallelism of a region and the runtime takes care of executing it. In OpenMP, however, the developer explicitly specifies the type of parallelism and those choices often have a measurable impact on the performance of the code. Converting a code from one programming model to another can be a fairly straightforward change [5,9]. However, porting a code to achieve the best performance can be a challenging task.

This work builds upon the results observed in [3], which includes an evaluation of the SPEC ACCEL benchmark suite across five compilers on three distinct architectures including Percival [1], Titan [7], and Summit [2].

4 Analysis

Here we take a closer look at the SPEC ACCEL benchmark kernels to determine what factors account for the performance difference. Since the benchmarks claim that they were created with performance portability in mind, the created kernels are functionally equivalent. Here we first present the profiling results as analyzed and displayed by the NVIDIA Visual Profiler [6]. From these profiles we pick the kernels that the most time to see how they differ in the two programming models. There exists a large number of variables in the determination of the exact cause of the performance difference, hence we follow the standard performance analysis criteria and analyze the kernels taking the maximum wall-clock time as they have the most impact on the performance of the benchmark. Figure 2 shows the timing profile of the GPU for the OpenMP version of the BT benchmark. We see that the kernels that take the maximum time for BT OpenMP version are from functions `x_solve`, `y_solve`, and `z_solve`, which account for 24% each of the total GPU time. Similarly, Fig. 3 shows the timing profile of the GPU for

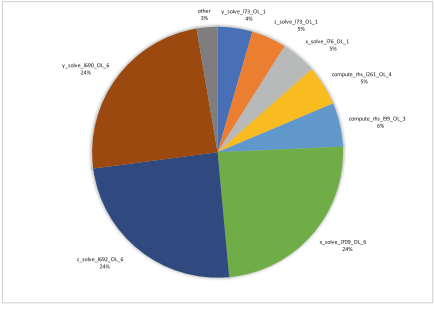


Fig. 2. BT OpenMP calls profiled.

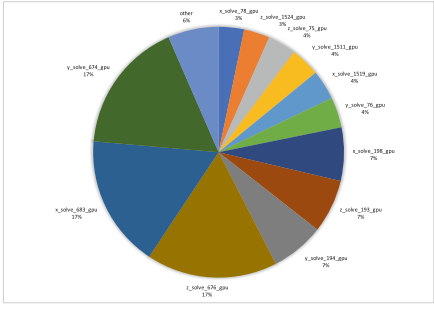


Fig. 3. BT OpenACC calls profiled.

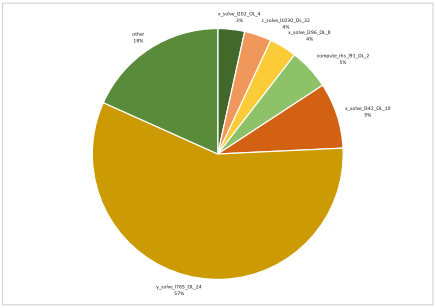


Fig. 4. SP OpenMP calls profiled.

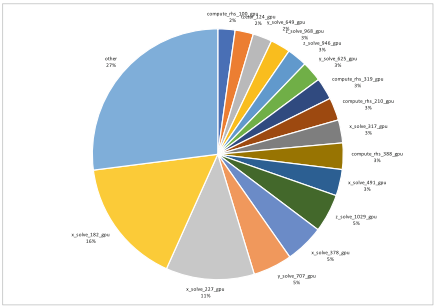


Fig. 5. SP OpenACC calls profiled.

the OpenACC version of the benchmark. The 51% of the total GPU processing time is evenly spread across `x_solve`, `y_solve`, and `z_solve` functions. The *other* category includes cumulative timings of kernels that take less than 1% of the total time. Figures 4 and 5 show the GPU profiles for SP benchmark's OpenMP and OpenACC versions. For OpenMP version of the benchmark we see that 57% of the GPU time is utilized by one invocation of the kernel from `y_solve` function while for the OpenACC version we see a trend of little contributions from all calls take relatively uniform times except kernels from the function `x_solve`. The *other* category includes cumulative timings of kernels that take less than 2% of the total time.

For the LBC and LBDC benchmarks we see that all of the GPU time is spent on a single invocation of a kernel. The details are presented in the Table 2.

Table 2. GPU profile for LBC and LBDC benchmarks.

Benchmark	Kernel	No. of invocations	OpenMP Avg. duration (μs)	OpenACC Avg. duration (μs)
LBM	StreamCollide	5000	3.100844	6.120061
LDBC	relax_collstream	5000	1.903010	2.003094

5 Discussion

In the following section we will discuss the kernels identified in Sect. 4 for the different benchmarks. We compare and contrast the differences in OpenMP and OpenACC constructs used in these kernels and throw some light on the relative performance based on additional profiles collected for these specific kernels.

5.1 BT Benchmark

For BT benchmark we look at the `x_solve` kernel and `compute_rhs`. Since `y_solve` and `z_solve` are very similar to `x_solve` our analysis on `x_solve` is applicable for the other two. Listing 5.1.1 and 5.1.2 lists the kernel for `x_solve`.

The OpenMP version the directive `target teams distribute parallel for` is short for `target` followed by `teams distribute parallel for`. The `teams` construct creates a league of thread teams and the master thread of each team executes the region. The `distribute parallel` loop construct specifies that the for loop with iterator "j" can be executed in parallel by threads from teams from different contention groups. The `for` loop enclosed by `omp simd` indicates that the loop can be lowered where multiple iterations of the loop can be executed by multiple SIMD lanes.

Listing 5.1.1. BT Kernel for x_solve (__xl_x_solve_l709_OL_6)

```
707  ...
708  #pragma omp target teams
          distribute parallel for
          private(i,k)
709  for (j = 1; j <= gp12; j++) {
710      for (i = 1; i <= isize-1;
               i++) {
711          #pragma omp simd
                   private(pivot,coeff)
712          for (k = 1; k <= gp22;
                   k++) {...}
713      }
714  }
715  ...
```

Listing 5.1.2. BT Kernel for x_solve

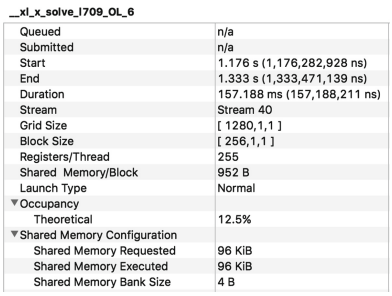
```
679  ...
680  #pragma acc kernels loop
681  for (k = 1; k <= gp22; k++) {
682      for (j = 1; j <= gp12; j++)
               {
683              for (i = 1; i <=
                       isize-1; i++) {...}
684      }
685  }
686  ...
```

Listing 5.1.3 shows the parallelization strategy implemented by the PGI compiler. The OpenACC version marked the loop nest with the kernel directive and leaves it to the compiler to analyze the loop and pick the right schedule for the loops. We see that OpenACC is more descriptive, there is more freedom for the compilers to apply parallelization techniques. In this case the PGI compiler decided to pick a gang and vector schedule of the "k" loop, a gang schedule for the "j" loop and a sequential schedule for the "i" loop.

Listing 5.1.3. PGI Compiler Parallelization Strategy for x_solve

```
1   681, Loop is parallelizable
2   682, Loop is parallelizable
3   683, Loop carried dependence of rhs,lhsX prevents parallelization
4            Loop carried backward dependence of rhs,lhsX prevents
                 vectorization
5            Inner sequential loop scheduled on accelerator
6            Accelerator kernel generated
7            Generating Tesla code
8        681, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
9        682, #pragma acc loop gang /* blockIdx.y */
10       683, #pragma acc loop seq
```

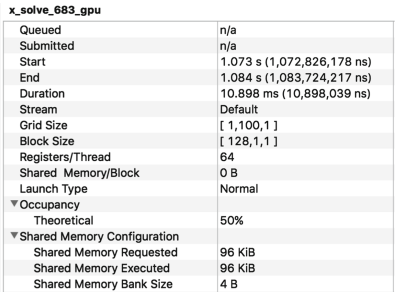
More insights can be obtained from the profiles in Figs. 6 and 7. The key parameters to look at there are the Grid Size and the Block Size as they together indicate the level of parallelism achieved. In addition the number of registers per thread and shared memory affects the performance, as threads share a finite number of registers and shared memory. The performance gain from increased occupancy (block size) may be outweighed by the lack of registers per thread. Inadequate registers will mean access to local memory more often, which is more expensive.



__xl_x_solve_l709_OL_6

Queued	n/a
Submitted	n/a
Start	1.176 s (1,176,282,928 ns)
End	1.333 s (1,333,471,139 ns)
Duration	157.188 ms (157,188,211 ns)
Stream	Stream 40
Grid Size	[1280,1,1]
Block Size	[256,1,1]
Registers/Thread	255
Shared Memory/Block	952 B
Launch Type	Normal
▼Occupancy	
Theoretical	12.5%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Fig. 6. BT benchmark x_solve OpenMP calls profiled.



x_solve_683_gpu

Queued	n/a
Submitted	n/a
Start	1.073 s (1,072,826,178 ns)
End	1.084 s (1,083,724,217 ns)
Duration	10.898 ms (10,898,039 ns)
Stream	Default
Grid Size	[1,100,1]
Block Size	[128,1,1]
Registers/Thread	64
Shared Memory/Block	0 B
Launch Type	Normal
▼Occupancy	
Theoretical	50%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Fig. 7. BT benchmark x_solve OpenACC calls profiled

For the OpenMP version, the GPU schedule is 1280 for the grid size and 256 for the thread block size. The register usage was 255. Overall this loopnest achieved a total of 12.5% GPU occupancy. On the other hand, for the OpenACC version, the GPU schedule for the loop nest was 100 grid size and 128 for the thread block size. The register usage per thread was 64 with no shared memory per file. This scheduled achieved a higher GPU occupancy of 50% than the OpenMP version. This is one of the primary reasons that the OpenACC version of the `loopnest` performed 14.4x faster than the OpenMP version. Another reason from the programming models point of view is that the OpenMP SIMD construct is not able to vectorize the loop iterations and serial execution further reduces performance. The OpenMP benchmark would benefit from having architecture specific code paths for further performance gain.

Listing 5.1.4. BT Kernel for compute_rhs (__xl_compute_rhs_l261_OL_4)

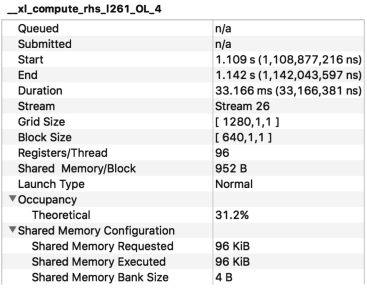
```
259   ...
260   #pragma omp target teams
          distribute parallel for
          private(vijk,vp1,vm1,i,j,k)
261   for (k = 1; k <= gp22; k++) {
262       for (j = 1; j <= gp12; j++)
              {
263           #pragma omp simd
                  private(vijk,vp1,vm1)
264           for (i = 1; i <= gp02;
                  i++) {...}
265       }
266   }
267   ...
```

Listing 5.1.5. BT Kernel for compute_rhs

```
262   ...
263   #pragma acc kernels loop
264   for (k = 1; k <= gp22; k++) {
265       for (j = 1; j <= gp12; j++)
              {
266           for (i = 1; i <= gp02;
                  i++) {...}
267       }
268   }
```

Listing 5.1.6. PGI Compiler Parallelization Strategy for compute_rhs

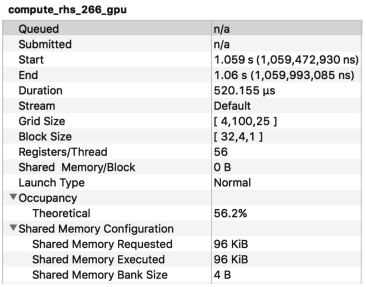
```
1  264, Loop is parallelizable
2  265, Loop is parallelizable
3  266, Loop is parallelizable
4           Accelerator kernel generated
5           Generating Tesla code
6        264, #pragma acc loop gang /* blockIdx.y */
7        265, #pragma acc loop gang, vector(4) /* blockIdx.z threadIdx.y */
8        266, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
```



__xl_compute_rhs_l261_OL_4

Queued	n/a
Submitted	n/a
Start	1.109 s (1,108,877,216 ns)
End	1.142 s (1,142,043,597 ns)
Duration	33.166 ms (33,166,381 ns)
Stream	Stream 26
Grid Size	[1280,1,1]
Block Size	[640,1,1]
Registers/Thread	96
Shared Memory/Block	952 B
Launch Type	Normal
▼Occupancy	
Theoretical	31.2%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Fig. 8. BT benchmark compute_rhs OpenMP calls profiled.



compute_rhs_266_gpu

Queued	n/a
Submitted	n/a
Start	1.059 s (1,059,472,930 ns)
End	1.06 s (1,059,993,085 ns)
Duration	520.155 µs
Stream	Default
Grid Size	[4,100,25]
Block Size	[32,4,1]
Registers/Thread	56
Shared Memory/Block	0 B
Launch Type	Normal
▼Occupancy	
Theoretical	56.2%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Fig. 9. BT benchmark compute_rhs OpenACC calls profiled

Listing 5.1.4 and 5.1.5 shows the OpenMP and OpenACC version of another `loopnest` in the `rhs` kernel of BT. We look at this kernel specifically because it takes 6% of the total time in the OpenMP version but about 1% in the OpenACC benchmark. Here both versions have the same code structure. No loop interchanged was done by the programmer. All the loops are parallel. The benchmark employs the OpenMP SIMD directive to the innermost loop. The OpenACC version of the loop uses the kernels directive and lets the compiler apply the loop schedules (Figs. 8 and 9).

Listing 5.1.6 is the output from the PGI compiler for the OpenACC `loop nest`. We can see that OpenACC applies gang and vector schedules for the three loops in the loopnest. As a result it gets a $4 \times 100 \times 25$ schedule for the grid and 32×4 schedule for the `threadblock` size. The occupancy is of 56.2%. The OpenMP version, on the other hand, has a schedule of 1280×1 for the grid and 640×1 for the same `threadblock`. The occupancy for OpenMP version is 31.2%. Low occupancy results in poor instruction issue efficiency and since there are not enough eligible warps, the latency between dependent instructions is more obvious. As a result, using default settings for both the versions of the benchmark, more threads were spawned in the OpenACC version leading to 63x better performance. This is the direct result of OpenACC compiler picking a better schedule for the loops.

5.2 SP Benchmark

In Listings 5.2.1 and 5.2.2 we compare OpenMP and OpenACC versions of the SP benchmark. We see that the outer loop is parallelized using OpenMP `target teams distribute parallel for` combined directive and using kernels, respectively. The OpenACC version parallelizes the "k" and "i" loop with gang vector schedules.

The loop schedule selected by OpenACC was $5 \times 40 \times 1$ grid size and $32 \times 4 \times 1$ thread block. OpenMP selected a $2 \times 1 \times 1$ grid size and $128 \times 1 \times 1$ thread block. The GPU occupancy for OpenACC was 50% and for OpenMP 31.2%. The 135x faster performance using OpenACC can be contributed to (1) better occupancy and (2) optimum registers per thread. In spite of OpenMP benchmark having shared memory between CPU and GPU and more registers per thread, the default block size was not the optimum size. This is an important aspect and leads to degraded performance due to inadequate resources per thread (Figs. 10 and 11).

Listing 5.2.1. SP Kernel for y_solve using OpenMP

```
...
#pragma omp target teams
     distribute parallel for
     private(i,j,k,m,fac1,j1,j2)
for (k = 1; k <= gp2-2; k++) {
    for (j = 0; j <= gp1-3;
         j++) {
        j1 = j + 1;
        j2 = j + 2;
        for (i = 1; i <= gp0-2;
             i++) {
            ...
            for (m = 0; m < 3;
                 m++) {...}
            ...
            for (m = 0; m < 3;
                 m++) {...}
            ...
            for (m = 0; m < 3;
                 m++) {...}
        }
    }
}
...
```

Listing 5.2.2. SP Kernel for y_solve using OpenACC

```
...
#pragma acc kernels loop
for (k = 1; k <= gp2-2; k++) {
    for (j = 0; j <= gp1-3;
         j++) {
        j1 = j + 1;
        j2 = j + 2;
        for (i = 1; i <= gp0-2;
             i++) {
            ...
            for (m = 0; m < 3;
                 m++) {...}
            ...
            for (m = 0; m < 3;
                 m++) {...}
            ...
            for (m = 0; m < 3;
                 m++) {...}
        }
    }
}
...
```

Listing 5.2.3. PGI's Parallelization Stratergy for y_solve

```
645, Loop is parallelizable
646, Loop carried dependence of lhsY prevents parallelization
         Loop carried backward dependence of lhsY prevents vectorization
         Loop carried dependence of rhs prevents parallelization
         Loop carried backward dependence of rhs prevents vectorization
649, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
     645, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
     646, #pragma acc loop seq
     649, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
     653, #pragma acc loop seq
     658, #pragma acc loop seq
     663, #pragma acc loop seq
```

__xl_y_solve_l765_OL_24

Queued	n/a
Submitted	n/a
Start	2.183 s (2,183,209,865 ns)
End	2.334 s (2,333,694,638 ns)
Duration	150.485 ms (150,484,773 ns)
Stream	Stream 35
Grid Size	[2,1,1]
Block Size	[128,1,1]
Registers/Thread	86
Shared Memory/Block	916 B
Launch Type	Normal
▼Occupancy	
Theoretical	31.2%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Fig. 10. SP benchmark compute_rhs OpenMP calls profiled.

y_solve_649_gpu

Queued	n/a
Submitted	n/a
Start	2.441 s (2,441,456,011 ns)
End	2.443 s (2,442,565,604 ns)
Duration	1.11 ms (1,109,593 ns)
Stream	Default
Grid Size	[5,40,1]
Block Size	[32,4,1]
Registers/Thread	64
Shared Memory/Block	0 B
Launch Type	Normal
▼Occupancy	
Theoretical	50%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Fig. 11. SP benchmark compute_rhs OpenACC calls profiled

5.3 LBM Benchmark

The OpenACC and OpenMP version of LBM are almost identical. Since the entire subroutine is called, we do not include the code listing. The OpenMP version uses the `target` combined directive and the OpenACC version uses parallel loop. In this case both versions use the same schedule 10157×1 for grid block and 128×1 for `threadblocks`. However, we observe that the OpenMP version is 2X faster than the OpenACC version. Contributing factors include (1) GPU shared memory, and (2) the number of registers per thread (3x as those in the OpenACC versions) (Figs. 12 and 13).

__xl_LBM_performStreamCollide_l159_OL_1

Queued	n/a
Submitted	n/a
Start	997.516 ms (997,516,434 ns)
End	1.001 s (1,000,590,913 ns)
Duration	3.074 ms (3,074,479 ns)
Stream	Stream 20
Grid Size	[10157,1,1]
Block Size	[128,1,1]
Registers/Thread	122
Shared Memory/Block	896 B
Launch Type	Normal
▼Occupancy	
Theoretical	25%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Fig. 12. LBM benchmark OpenMP kernel details.

LBM_performStreamCollide_195_gpu

Queued	n/a
Submitted	n/a
Start	755.546 ms (755,546,152 ns)
End	761.619 ms (761,619,310 ns)
Duration	6.073 ms (6,073,158 ns)
Stream	Stream 19
Grid Size	[10157,1,1]
Block Size	[128,1,1]
Registers/Thread	56
Shared Memory/Block	0 B
Launch Type	Normal
▼Occupancy	
Theoretical	56.2%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Fig. 13. LBM benchmark OpenACC kernel details.

5.4 LBDC Benchmark

Table 2 shows that relax_collstream subroutine is invoked 5000 times by both OpenMP and OpenACC versions of the LBDC benchmarks. The OpenMP benchmark uses the combined construct `target teams distribute parallel do simd` to offload the computation loop to the GPU. This allows for a `team` of threads to, in parallel, execute `simd` instructions when possible.

The corresponding code for the OpenACC version depicted in Listing 5.4.2 uses a simple OpenACC parallel loop. Since the OpenMP code has been better optimized to use vectorization through SIMD construct we see up to 2.5X performance improvement on Summit. The sub-routine details highlighted in Figs. 14 and 15 show that though most other parameters are identical OpenMP uses 900 B of GPU shared memory. This leads to better data access patterns leading to better execution times for the OpenMP version.

Listing 5.4.1. LBDC OpenMP Offloading of relax_collstream

```
!$omp target ! present(f_now,f_nxt,send)
!$omp teams distribute parallel do simd
     private(f_tmp_NE,f_tmp_N,...,feq_common)                &
!$omp    shared(omega_h,asym_omega_h,f_now,f_nxt,n_cells,omega,send)
    do i_ct = 1, n_cells
       f_tmp_NE = f_now( F_IDX(i_ct,Q19_NE) )
       ...
       f_tmp_S  = f_now( F_IDX(i_ct,Q19_S ) )
 ...
!$omp end target
```

Listing 5.4.2. LBDC OpenACC Offloading of relax_collstream

```
!$acc parallel loop present(f_now,f_nxt,send)
    do i_ct = 1, n_cells
       ...
   ...
    end do
```

__xl__mod_relax_NMOD_relax_collstream_l47_OL_1

Queued	n/a
Submitted	n/a
Start	1.457 s (1,456,545,867 ns)
End	1.458 s (1,458,423,488 ns)
Duration	1.878 ms (1,877,621 ns)
Stream	Stream 20
Grid Size	[25669,1,1]
Block Size	[128,1,1]
Registers/Thread	64
Shared Memory/Block	900 B
Launch Type	Normal
▼Occupancy	
Theoretical	50%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Fig. 14. LBDC benchmark OpenMP kernel details.

relax_collstream_48_gpu

Queued	n/a
Submitted	n/a
Start	747.049 ms (747,048,905 ns)
End	749.05 ms (749,049,783 ns)
Duration	2.001 ms (2,000,878 ns)
Stream	Stream 19
Grid Size	[25669,1,1]
Block Size	[128,1,1]
Registers/Thread	64
Shared Memory/Block	0 B
Launch Type	Normal
▼Occupancy	
Theoretical	50%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Fig. 15. LBDC benchmark OpenACC kernel details.

6 Conclusion

In this paper we highlight the differences in the much used HPC accelerator programming models - OpenMP and OpenACC through the in depth analysis of the SPEC ACCEL 1.2 benchmarks suite. Both OpenACC and OpenMP versions of each benchmark followed similar parallelization strategies at the directive level, save some vectorization hints through OpenMP's SIMD directives. However, OpenACC gives more freedom to the compiler to accelerate their `loopnests`. OpenMP leaves all the choices to the user because of its more prescriptive

nature. As a result, in many cases, OpenACC picks better schedules than what a programmer or OpenMP implementation allows because OpenACC relies on compiler optimization technology to generate their directives. This shows that OpenACC needs good compiler implementations as most of the choices are left to the implementation.

Another factor is the number of active blocks on the GPU device. This contributes to the occupancy of the device. We have seen that low occupancy results in poor instruction issue efficiency (BT and SP). In such cases there are not enough eligible warps to hide latency between dependent instructions. When occupancy is at a sufficient level to hide latency, increasing it further may degrade performance due to the reduction in resources per thread (as seen for LBM). For better performance as well as optimal use of resources an early step of kernel performance analysis must check occupancy and observe the effects on kernel execution time when running at different occupancy levels.

OpenMP can mimic OpenACC behavior by tuning to the parameters selected by the OpenACC compilers. However, the OpenMP implementations are becoming more sophisticated and sometimes support optimizations that are not supported by OpenACC compilers, such as GPU shared memory. We saw this case where the loop schedules were identical for OpenMP and OpenACC implementations of LBDC but OpenMP version took advantaged of GPU shared memory and thus performed better.

Acknowledgement. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC05-00OR22725. This research 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. We would like to thank Dr. Oscar Hernandez from ORNL for his guidance and support during the writing of this manuscript.

References

1. Percival quickstart guide. https://www.olcf.ornl.gov/percival-quickstart-guide/
2. Summit: Scale new heights. Discover new solutions. https://www.olcf.ornl.gov/summit/
3. Boehm, S., Pophale, S., Vergara Larrea, V.G., Hernandez, O.: Evaluating performance portability of accelerator programming models using SPEC ACCEL 1.2 benchmarks. In: Yokota, R., Weiland, M., Shalf, J., Alam, S. (eds.) ISC High Performance 2018. LNCS, vol. 11203, pp. 711–723. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02465-9_51
4. Juckeland, G., Grund, A., Nagel, W.E.: Performance portable applications for hardware accelerators: lessons learned from SPEC ACCEL. In: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, pp. 689–698, May 2015. https://doi.org/10.1109/IPDPSW.2015.26
5. Juckeland, G., et al.: From describing to prescribing parallelism: translating the SPEC ACCEL OpenACC suite to OpenMP target directives. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds.) ISC High Performance 2016. LNCS, vol. 9945, pp. 470–488. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46079-6_33

6. NVIDIA: NVIDIA Visual Profiler. https://developer.nvidia.com/nvidia-visual-profiler
7. Oak Ridge National Lab: Titan supercomputer. https://www.olcf.ornl.gov/titan/
8. Top 500: Top 500: June 2018. https://www.top500.org/lists/2018/06/
9. Wienke, S., Terboven, C., Beyer, J.C., Müller, M.S.: A pattern-based comparison of OpenACC and OpenMP for accelerator computing. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014. LNCS, vol. 8632, pp. 812–823. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09873-9_68