

OpenMP 4.5 IBM November 2015 Hackathon: Current Status and Lessons Learned

Erik W. Draeger, Ian Karlin, Tom Scogland,
David Richards, Jim Glosli, Holger Jones,
David Poliakoff, Adam Kunen

Lawrence Livermore National Laboratory

January 11, 2016

LLNL-TR-680824

Lawrence Livermore National
Laboratory is operated by Lawrence
Livermore National Security, LLC,
for the U.S. Department of Energy,
National Nuclear Security
Administration under Contract
DE-AC52-07NA27344.



Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Executive Summary

On November 3-5, 2015, IBM hosted a three-day hackathon at their Yorktown Heights facility, to pair code developers from multiple DOE labs with IBM applications and compiler experts.

The OpenMP 4.5 hackathon was an extremely successful event. It provided an immersive environment for LLNL and other laboratory software developers to gain knowledge of how to begin reasoning about porting their code to OpenMP 4.5. The agenda provided a good mix of overview material and coding time but also allowed enough time and flexibility for side discussions. The level of commitment from IBM was high, with at least one IBM employee per two laboratory visitors available for pair programming, question answering and real time compiler bug fixing. IBM was extremely responsive to bugs, and fixes were usually available in less than an hour, thus keeping productivity high. The hackathon was a high value engagement well worth traveling to for all participants and we compliment the IBM team on their preparation. The quality of this event should be considered a gold standard for organization, preparation and involvement by the hosting organization.

The authors of this report studied three different codes: LULESH, Kripke, and Cardioid. All teams gained valuable experience with the new standard and were able to make progress on porting their codes to OpenMP 4.5. However we also uncovered several issues that are likely to create significant barriers to porting large and complex codes, especially C++ codes, to Sierra using OpenMP 4.5. These issues include:

1. OpenMP 4.5 is highly biased toward a loop-offload accelerator execution model. It is difficult to move code to the GPU at granularities larger than a loop or small function.
2. OpenMP 4.5 does not guarantee support for common C++ features including the STL, exceptions, and virtual functions in target code. The specification implies that they should be available, but many meaningful backends including GPUs do not. It is not trivial to remove these features from all code we wish to run on the GPU.
3. It is not clear how to write OpenMP 4.5 code with target directives such that it will perform well on both GPU-based and Intel Phi-based architectures.

It seems likely that we will need to devise new programming patterns and refactor code to adopt these patterns to take full advantage of OpenMP 4.5.

Bugs are expected in early access compilers, and our codes quickly exposed problems. We believe that most or all of the correctness issues that we found can be quickly and easily addressed. Poor performance of generated code is a more

complex issue and future work with IBM and NVIDIA will likely be necessary to identify appropriate guidance for application developers and to implement necessary compiler features to create code that performs well.

Finally, we have identified a number of limitations or deficiencies of the OpenMP standard. These include:

1. Mapping complex data structures to the GPU is tedious and produces very ugly code. Serialization and deep copy operations are badly needed.
2. The `declare target` directive, required to create device versions of functions and methods, is under-specified for use in and around C++ classes.
3. The new `defaultmap` clause in OpenMP 4.5 is well needed, but lacks options to change the behavior for aggregates or scalars; specifically the ability to make aggregates default `firstprivate` is an issue.
4. Mapping of class members and especially the `this` pointer is under-specified, and will be important for C++ applications.
5. Mapping of virtual classes, even when the class in question has a concrete type as in a non-virtual method of same, is not allowed.
6. There is currently no way to request a pointer to a device function on a given target device, some application codes will require this support in both C and C++.

Hopefully future versions of the OpenMP standard will be able to address these limitations.

OpenMP 4.5 is a significant improvement compared to OpenMP 4.0. Despite these improvements, implementing large complex codes using OpenMP 4.5 remains a significant challenge. However, we believe OpenMP will continue to evolve in ways that will make code implementation and device usage easier in the future. Also, despite the limitations in OpenMP 4.5, as of today it provides the best opportunity for us to generate performance portable codes for our diverse C, C++, and Fortran code base. While it will be more challenging than any of us may like to use OpenMP 4.5 effectively, we believe that the close collaboration with IBM and our early start have us heading in the right direction.

1 Review of OpenMP 4.x Features

OpenMP 4.0 was released in July 2013. One of the most important feature additions was the support for accelerators such as GPUs.

OpenMP 4.5 was released in November 2015 and contains significant feature improvements based on lessons learned from the 4.0 release. These features include:

- **Significantly improved support for devices.** OpenMP now provides mechanisms for unstructured data mapping, asynchronous execution and also runtime routines for device memory management. These routines allow for explicit allocation, copying and freeing of memory between devices.
- **Support for doacross loops.** A natural mechanism to parallelize loops with well-structured dependences is provided.
- **New taskloop construct.** Support to divide loops into tasks, avoiding the requirement that all threads execute the loop.
- **Reductions for C/C++ arrays.** This often requested feature is now available by building on support for array sections.
- **Task and lock hint mechanisms.** Hint mechanisms can provide guidance on the relative priority of tasks and on preferred synchronization implementations.
- **Thread affinity support.** It is now possible to use runtime functions to determine the effect of thread affinity clauses.
- **Improved support for Fortran 2003.** Users can now parallelize many Fortran 2003 programs.
- **SIMD extensions.** These extensions include the ability to specify exact SIMD width and additional data-sharing attributes.

It is clear from our experiences at the hackathon that OpenMP 4.5 features are going to be critical to application performance and portability of code. However, given that the 4.5 standard is new, and we have already found some significant shortcomings that we discuss below, we believe OpenMP 4.5 should be thought of a significant waypoint in an evolving and improving ecosystem.

2 Key Findings for Application Developers

Porting large applications to Sierra using OpenMP 4.5 will be a major challenge. Our experience during the hackathon uncovered some of the most significant challenges application developers will need to confront.

While new features of OpenMP 4.5 and beyond make it easier to access GPU accelerators and manage data movement than earlier versions of the standard, the OpenMP programming model still has limitations and may require significant code restructuring to ensure portability. The two main lessons we took away from this hackathon were (1) to maximize memory coalescence and avoid divergence on the GPU as much as possible, and (2) to write “static” code as much as possible to use on the device, e.g., C-style kernels or encapsulated C++ static templates and objects that eschew polymorphism, external libraries, etc. Simply decorating an existing code without considering how to limit the scope of pragma markup and the subsequent requirements on libraries and functions used within is not recommended.

2.1 OpenMP 4.5 is biased toward a loop/kernel offload execution model

OpenMP 4.5 is a major improvement compared to OpenMP 4.0. In particular the unstructured data mapping pragmas (`#pragma omp target data enter` and `#pragma omp target data exit`) provide a needed mechanism to map data to and from the GPU without being limited to a single program scope. However, OpenMP 4.5 is highly biased toward an execution model where loops or kernels are offloaded to the accelerator one at a time with all flow control handled by the host processor. Applications will likely discover that the most practical approach to using the accelerator is to identify highly self-contained loops or kernels and decorate these code segments to execute on the device. Sending large blocks of code to the accelerator all at once, or attempting to run code natively on the accelerator with the CPU serving as a “serial accelerator” is either impossible or impractical.

Specific examples of problems developers are likely to encounter are given in the appendix of this document. See for example:

- Appendix A.1: All functions and methods called from target code must be device code and in the case of Nvidia GPUs either written in CUDA or decorated with `#pragma omp declare target`. This can quickly cascade and require pragma decorations across large parts of the code base.
- Appendix A.2: Code within `#pragma omp declare target` regions can not call library functions that do not have a GPU-compatible version available.

2.2 Code intended for the GPU cannot use important C++ features such as polymorphism or the STL

Although OpenMP is agnostic about most language features in Fortran and C/C++, GPUs are not capable of supporting some programming constructs. Many commonly used C++ features, such as templates, that are inlined at compile time are OK to use in GPU code. However, dynamic code that requires accessing a virtual function table are either not allowed or not enabled by the model.

For a developer this means that virtual member functions cannot be defined within a `#pragma omp declare target` region, limiting the use of polymorphism. Although this behavior is not expressly forbidden by the OpenMP standard, it is also not required, making it unclear if such polymorphism will be consistently supported. In addition, GPUs do not currently have the hardware support needed for polymorphism. Finally, the standard does ensure that any object with virtual member functions can be mapped to the device limiting how a developer can define certain classes. Note this last restriction is in place regardless of if only concrete objects are called from the device.

Using the standard template library (STL) on a GPU is currently not possible. Many STL functions (e.g., `push_back`) are inherently serial, as are most iterator types. While it is possible in principle to parallelize algorithms with random access iterators, any attempt will quickly encounter the issue of calling library code in terms of the decoration requirements described in the previous subsection.

2.3 Performance portability is a major concern

Although OpenMP 4.5 offers a welcome portable approach to GPU programming, there is still reason to be concerned with performance portability. How will device code be interpreted on non-GPU architectures such as Xeon Phi? Will we be required to maintain separate device and non-device versions of large amounts of code? Even if there is a sensible interpretation of device code on Xeon Phi, the inherent architecture differences will dictate different loop schedules to maximize the efficient use of resources. For example GPUs need a (static, 1) loop schedule to coalesce memory access, but this schedule is terrible for a SIMD unit on Xeon Phi. We have only started to consider the possible challenges for cross platform programming in OpenMP 4.5.

Furthermore, discussions with the IBM compiler developers revealed the degree to which runtime behavior is implementation development. It is possible for vendors to interpret the OpenMP standard sufficiently differently so as to make it difficult or impossible for code to be performance portable. Detailed examples of where the standard is vague enough to warrant concern are outlined in:

- Appendix A.3 Implementation Dependent Runtime Behavior

We note the IBM compiler team is aware of these issues and does have a good working relationship with the Intel compiler team so there is hope for good solutions. Continuing this early engagement with Intel is recommended to ensure that we avoid creating “solutions” that turn out to be future problems.

2.4 Mapping data can be tedious. No deep copy.

Facilities to map data to the device are still relatively limited. Typically each member of a complex data structure must be mapped individually. There is no awareness of C++ constructors or destructors and currently no support for serialize/deserialize methods that could make data mapping less tedious and/or error prone. Deep copy, i.e., the ability to follow pointers when mapping data from host to device, is not facilitated by OpenMP 4.5. As such, using data structures of pointers to pointers is a significant programming burden. It is hoped that this functionality will exist in OpenMP 5.0.

2.5 New programming patterns are necessary

The above programming challenges and concerns will require application developers to redesign many portions of their application. The issues, workarounds and challenges mentioned earlier in this section should provide a start towards best practices for moving codes to OpenMP 4.5 and/or GPU accelerated architectures. However, the issues identified above are an OpenMP and GPU centric view. Developers investing significant effort into refactoring their code to OpenMP 4.5 with an eye towards performance portability will need to understand how these challenges intersect with the constraints imposed by the current machines their codes run on and other platforms that are likely to be deployed in the foreseeable future.

3 Key Findings for the CORAL Compiler Team

Overall the hackathon participants were impressed by the quality of the compiler team and progress achieved to date. Members of the team were helpful, knowledgeable and able to resolve many issues as they arose during the hackathon. In this section, we present a high-level overview of the unresolved compiler issues we encountered and in the appendix we document some of these issues in more detail. It is important to note that fixing many of these issues is already plan of record for IBM.

3.1 Thread scheduling is not performance portable

The current compiler requires marking loops with `schedule(static, 1)` in order to get coalesced memory access on the GPU. However, this schedule results in

poor performance on the CPU. To avoid the need for separate pragmas for the GPU and CPU cases (or code that is only performant on one of the two architectures) GPU pragmas should be treated as `schedule(static, 1)` by default for code writing ease, performance portability and clarity.

3.2 Splitting pragmas across lines hurts performance but line splits are required since the if clause can't target specific keywords

Although the OpenMP standard allows one to break up directives into multiple pragma lines, doing so with the current compiler implementation can significantly impact performance. In addition, since the if clause in the current implementation can not target single clauses in a pragma (this is an OpenMP 4.5 feature) to generate CPU and GPU code from the current compiler requires splitting pragmas resulting in sub-optimal performance. We note that while we did not encounter any code at the hackathon that needed if statements to be split across lines, performance portability might require this in some cases.

3.3 Issues with mapping data

Private class member variables cannot be mapped directly to a device. As a workaround, one must create a local reference to them or a private copy. Currently, the compiler only supports the OpenMP 4.0 mapping behavior where all variables used in a target construct are implicitly mapped to/from. A switch to the 4.5 standard will reduce both data traffic and should correct some of the bugs in the current compiler related to mapping (See Appendices B.1 and B.2)

3.4 Register usage higher than comparable CUDA code

In tests using LULESH, register usage was 2-3x greater than comparable CUDA implementations, causing register spills and decreased occupancy. Performance of one tested kernel showed it was approximately 2x slower than CUDA. Some of these differences are due to how mapped data is handled in the current implementation; however, most of the register usage differences are not explained at this time. (See Appendix B.3)

3.5 Lambda issues

Currently there is no compiler support for nested lambdas needed for some of our applications, such as Kripke. In addition, lambda capture is capturing items from the outer scope, resulting in the device getting host pointers.

4 Key Findings regarding the OpenMP 4.5 Standard

These are issues we identified where the OpenMP standard doesn't specify a solution or indicate whether one is even possible. Note that these are simply issues that were encountered during the course of the hackathon, and do not represent a comprehensive listing for the spec as a whole.

4.1 Support for complex data-structures is lacking

OpenMP currently offers no facilities for handling nested data structures, those that contain pointers to sub-objects for example. It is tedious, ugly, and difficult to map such complex data-structures, and they are ubiquitous in meaningful application codes. Solutions like deep-copy and user-definable mapping or serialization/deserialization mechanisms could go a long way to alleviate this, but none are currently available.

4.2 The `defaultmap` clause is incomplete

While the concept of `defaultmap`, being able to change the default behavior for a given variable type on a scope, is quite useful we found that its current definition is insufficient for several important use-cases. As it is now, it can only be used to change the behavior for scalars to be `map(tofrom:)`. This is useful in that it returns the OpenMP 4.0 behavior, but it lacks the ability to change the behavior of aggregates and pointers and even the ability to express the actual default. OpenMP should offer more configurability with this construct.

4.3 OpenMP interoperability with other threading models is undefined

Some LLNL codes (e.g. Hydra) combine specialized pthreads, C11 threads or other such models, sometimes hidden in libraries, with OpenMP. The OpenMP standard doesn't specify how these should interact, nor does it specify how two implementations of OpenMP may interact when linked into the same program. This is usually a performance issue, but in some cases can cause complete failure as well.

4.4 Pointers to device functions

The OpenMP standard doesn't offer a mechanism to retrieve a pointer to a target function, or to ensure that such a pointer exists inside a target region. Since any pointer to such a construct must come from OpenMP, this makes any code using function pointers inside target regions non-portable for the time being. This is likely to be important across all OpenMP supported languages, including C, C++ and Fortran versions since Fortran 90.

4.5 C++ support

C++ is currently supported as C is supported, which is to say that it can be used but not all extensions above C are guaranteed or facilitated. Specifically, three issues stand out:

- Interactions between the `declare target` directive and top-level scoped constructs such as classes and namespaces have not been defined. While the committee seems to agree this is an issue to address soon, there is currently no specification for how these interact.
- Mapping of member variables inside methods is not well defined, they may either be references to a mapped this pointer or handled by their individual default attributes. This ambiguity could be quite harmful to portability if unresolved.
- Any class containing a virtual method cannot be mapped, have its data brought into the device data environment, even when the class is concrete and that method is never invoked on the device. While full support for virtual functions is a significant issue on target platforms, the inability to even use the data storage of the class is problematic for some codes.

5 Final Impressions and Future Thoughts

The OpenMP 4.5 hackathon was an extremely valuable use of time for the LLNL application developers that attended. Lessons learned about how to write OpenMP 4.5 code are already propagating back to the wider application development community at LLNL. In addition, we learned that while OpenMP 4.5 and beyond make it easier to access GPU accelerators and manage data movement there are still areas we need to work with the standards committee to enable needed productivity and performance features. While the compiler had to be expected problems, we were impressed with how the team took our constructive criticism and is already addressing issues uncovered. We would like to thank the IBM team one final time for organizing the event. We realize that significant effort was required beyond the three days we worked together side by side to make it the success it was. We hope to have future engagements like the hackathon as appropriate.

Appendix A OpenMP Issues

The teams discovered multiple issues with the OpenMP 4.5 standard that may significantly limit the design, functionality and performance portability of large object-oriented codes.

A.1 All functions and methods called from target code must be enclosed within #pragma omp declare target regions

If the loop you want to execute uses C++ objects or functions, all code used or included by these objects or functions must be within #pragma omp declare target regions.

```
ReactionFHN rfhn;  
#pragma omp target map(to:rfhn)  
{  
    rfhn.calc();  
}
```

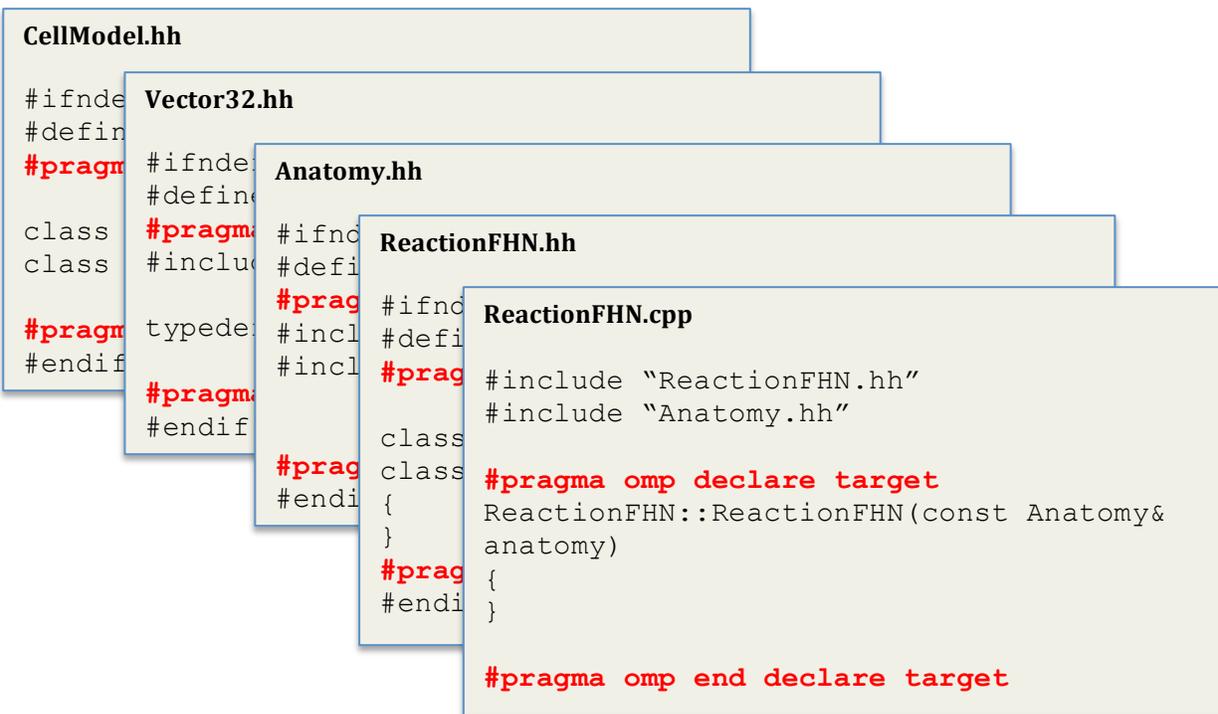


Figure 1: Pseudo-code demonstrating how the OpenMP requirement that #pragma omp declare target regions (red) follow all object dependencies can lead to markup throughout the code base.

A.3 Implementation dependent runtime behavior

In order to allow flexibility and support many devices the OpenMP committee left the behavior of many key attributes loosely defined. While necessary to allow OpenMP to support a plethora of devices for HPC use cases this is sometimes sufficiently vague to cause concern. How various vendors interpret these vague sections can result in significant performance, performance portability and programming challenges. At the hackathon we identified the following potential issues:

1. Interpretation of the teams construct: Implementations are allowed to create as few as one team for the teams construct, which may leave more contention in the application than necessary. Others require multiple teams to perform well, which may create unfortunate conflicts of interest.
2. The `map` clause only specifies that a device will acquire a view of the mapped data. This allows the runtime to decide when and if to move data between the multiple memory spaces on a node. Therefore, a program written only using `map` to target devices will need to rely on the runtime to map data in a performant manner. In order to guarantee what a programmer wants to happen occurs on all machines, new mechanisms are required to ensure sensible placement of data, especially giving users some control over such placement.

Appendix B Implementation Issues

In this section we identify issues we faced with the current implementation of OpenMP 4.5 within the version of the Clang LLVM compiler used at the hackathon.

B.1 Unable to map private class member variables to device

When mapping data to the device from within a member function, one must create a local reference to or copy of private member variables one wants to map to the device. (At the start of the hackathon, references to private member variables could not be mapped. This was identified as a compiler bug and fixed.)

```
// This won't work
void SimulationLoop::moveCells()
{
    #pragma omp target teams distribute parallel for map(to:dt_) \
        map(to:nCells_)
    for (int ii=0; ii<nCells_; ii++)
    {
        integrator.updateCell(ii,dt_);
    }
    return;
}
```

```
// This should work
void SimulationLoop::moveCells()
{
    const double& dtR = dt_;
    const int& nCellsR = nCells_;
    #pragma omp target teams distribute parallel for map(to:dtR) \
        map(to:nCellsR)
    for (int ii=0; ii<nCellsR; ii++)
    {
        integrator.updateCell(ii,dtR);
    }
    return;
}
```

B.2 Implicit mapping behavior

In OpenMP 4.0 anything used in a target construct is implicitly mapped to from. In OpenMP 4.5, scalars are first private by default, structures are mapped to from by default, and pointers and arrays are handled as zero-length array sections. The current compiler uses the 4.0 behavior, which caused significant issues with pointer passing resulting in multiple mappings and various failures.

B.3 Register usage higher than comparable CUDA code

In tests using LULESH, register usage was 2-3x greater than comparable CUDA implementations, causing register spills and decreased occupancy which can have a significant impact on performance. At the hackathon, performance numbers were obtained for one of the LULESH kernels and found to be approximately 2x slower than CUDA. Part of the register usage differences is how pointers and device mapped data is handled and IBM is working on this (see 2.5). However, it is unclear where the other register usage differences come from. They could be from the IBM compiler or the Nvidia PTX compiler used by IBM as a backend, and more extensive performance testing is needed across sizes to draw definitive conclusions.

Kernel/Registers	64	128	255
1	28381	22694	27143
4	31898	27493	32349
6	40806	42521	39330
7	67852	52589	52910
14	16207	9107	5435
15	3384	2548	2676
16	5282	5689	8854
Sum	193810	162641	168697

Table 1: LULESH kernel performance in microseconds for a 120^3 size. Results were similar for other sizes (60^3 and 90^3)

We note that when maximum kernel register usage is below 64 registers, OpenMP kernel performance on the GPU is within ~10% of CUDA of equivalent CUDA code. Testing showed that relaxing the use of compiler defaults on the number of registers to use could be advantageous. A simple heuristic of determine how many registers a kernel needs and round to the nearest number available can produce better performance on a kernel by kernel basis at a reasonable compile time cost.