LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-ABS-735945

Porting the Opacity Client Library to a CPU-GPU Cluster Using OpenMP 4.5

J. S. Kimko, M. M. Pozulp, R. Haque, L. Grinberg

July 31, 2017

Supercomputing 2017
Denver, CO, United States
November 12, 2017 through November 17, 2017

Porting the Opacity Client Library to a CPU-GPU Cluster Using OpenMP 4.5

Jason S. Kimko[1], Michael M. Pozulp[2], Riyaz Haque[2], Leopold Grinberg[3]
[1]College of William & Mary, [2]Lawrence Livermore National Laboratory, [3]IBM Research

***Abstract*—The poster accompanying this summary exhibits our experience porting the Opacity client library [1] to IBM's "Minsky" nodes [2] using OpenMP 4.5®. We constructed a GPU-friendly container class that mimics existing library functionality. We benchmarked our implementation on Lawrence Livermore National Laboratory's (LLNL) RZManta [3], a Minsky cluster. In our benchmarks on a single POWER8® CPU and Tesla® P100 GPU, we observed up to a 4x speedup including CPU-GPU data transfers and up to a 30x speedup excluding data transfers. Optimizing to reduce register pressure and increase occupancy may improve speedups. Our results demonstrate a successful and beneficial library port to the CPU-GPU architecture.**

I. Introduction

KULL [4] is one of LLNL's multi-physics simulation codes that models high energy density physics applications such as inertial confinement fusion. KULL uses the Opacity client library to obtain opacity data for radiation transport simulations and aims to run on CPU-GPU clusters in the future.

The Opacity client library is written in C++ and performs bilinear interpolations and various extrapolations to provide material opacities given density and electron temperature pairs. An opacity is a physical quantity that dictates the behavior of radiation as it interacts with a material, such as whether it is absorbed or scattered. In order to perform these interpolations and extrapolations, the library reads a data file, which contains density and electron temperature values along with their respective opacities, and distributes this data across many classes. These classes interact with one another during runtime to perform library functions.

OpenMP is used to achieve a portable, single source code solution, and since version 4.0, it has provided support for GPU offloading through its device constructs. OpenMP 4.5 provides additional features such as the `target enter/exit data` construct [5], which is used in our implementation.

The Opacity client library's main worker is the `Lookup()` function. The function itself is not computationally heavy, so the largest opportunity for user code speedups comes from increasing the throughput of simultaneous library calls (Figure 1). Therefore, the major contribution of this effort is to provide offloading support for the library rather than accelerating library functions.

```
#pragma omp target teams distribute parallel for map(…)
for (i=0; i<n; i++) {
    gpu->Lookup(density[i], temperature[i], &opacity[i])
}
```

Fig. 1. Expected kernel structure in user code to maximize throughput.

II. Implementation

Many of the classes that store interpolants use the C++ Standard Template Library and virtual functions, specifically the `std::vector` templated class. While these features provide convenience, they collectively make porting the code, i.e. managing CPU-GPU data transfers and generating PTX instructions for GPUs, more challenging.

To circumvent these issues, we created a GPU-compatible, C-styled container class that directly stores all of the necessary interpolant values and implements methods for identical library functionality. In total, the development process took around one month for an inexperienced OpenMP 4.5 developer.

Our implementation requires the following user code changes:

1) Setting a library flag that enables target offloading underneath existing function calls during the setup phase
2) Extracting the GPU-compatible object from the host object that normally calls `Lookup()`
3) Swapping the original host object for the extracted object to perform `Lookup()`'s

This implementation is not completely portable due to steps (2) and (3), so these changes should be hidden using conditional preprocessor macros for target offloading. In KULL, the calls to `Lookup()` are grouped together, so these changes are manageable. In the future, we wish to only require setting the library flag.

This port is not currently optimized and profiling with NVIDIA Visual Profiler [6] reveals high register pressure and low occupancy; how they affect performance and how they can be optimized is beyond the scope of this work. Using IBM's Clang 4.0 compiler, we observed 102 registers per thread and 24.9% achieved occupancy. Using IBM's XL 14.1 compiler, we observed 56 registers per thread and 50% achieved occupancy.

III. Alternative Solutions

Briefly, we will discuss alternative solutions for the two aforementioned problems: the C++ Standard Template Library and virtual functions. The underlying issue for both cases is that data is bitwise copied between memory spaces. If the data in question contains a pointer, that pointer may not be valid after copying. This issue manifests as the data pointer for `std::vector` and the virtual method table for virtual functions. In addition, there is no support for modifying these pointers directly.

```
#pragma omp target data map(…)      // data movement
{
double wtime = omp_get_wtime();
#pragma omp target teams distribute parallel for
for (i=0; i<n; i++)
    gpu->Lookup(density[i], temperature[i], &opacity[i])
wtime = omp_get_wtime() - wtime;
}
```

Fig. 2. Example structure of timing code that *excludes* time for data movement.

To resolve this issue for virtual functions, instances of classes with virtual functions must be constructed inside target regions to generate valid device function pointers. This can be accomplished by allocating device memory and then using placement new. Also, the virtual functions and their definitions must be placed inside a `declare target` region, or one can consider using a compiler flag to enable implicit target declaration if it is supported. For `std::vector`, a custom allocator can be defined that uses pinned or unified memory [7].

IV. Results

We ran an Opacity client library driver on an IBM Power System™ S822LC [2], which features two ten-core POWER8® prime CPUs, four NVIDIA Tesla® P100 GPUs, and NVLink 1.0 interconnects. Our driver mimics expected KULL usage and collects timings for the `Lookup()` loop. Wall times were collected on the host with 40 hardware threads using the environment variable `OMP_PLACES={0:40:2}` in order to limit the threads to one socket to avoid potential NUMA effects. On the target device, two timings were collected: one which includes the time required to map the query data and one without (Figure 2). Speedups are calculated as:

$$Speedup = \frac{time_{CPU}}{time_{GPU}}$$

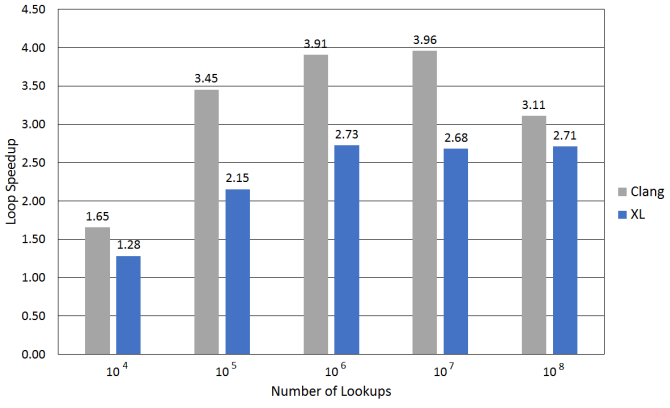

Fig. 3. Loop Speedup vs. Number of Lookups, including data mapping

Currently, KULL is required to map the query data containing density and electron temperature pairs before calling `Lookup()`. Accounting for this data movement, we see speedups from 1.65x to 3.96x using the Clang compiler and 1.28x to 2.73x using the XL compiler, depending on the number of lookups (Figure 3).

In the future, we expect KULL to initialize the query data in GPU memory. Thus, this data transfer will become unnecessary, resulting in speedups from 1.75x to 29.74x using Clang and 1.37x to 19.21x using XL (Figure 4).

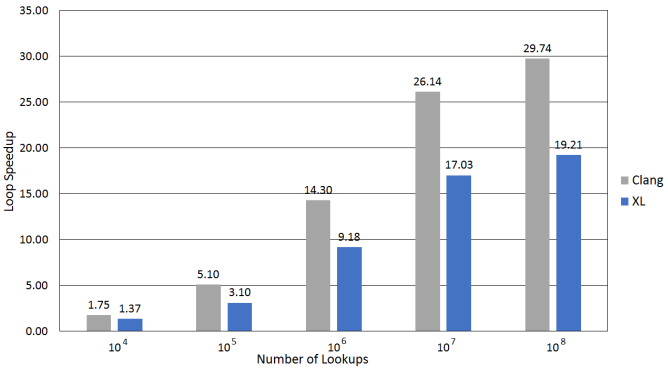

Fig. 4. Loop Speedup vs. Number of Lookups, excluding data mapping

V. Conclusion

With the ever increasing importance of the CPU-GPU architecture, many scientific applications will want to claim the advantages therein. With an unoptimized OpenMP 4.5 port, which features at its core a C-styled container class, performance gains were still observed and very attainable, peaking at 4x when including CPU-GPU data transfers and 30x when all of the data can be kept in GPU memory. We are integrating the ported library into KULL, and we hope to experiment with different runtime configurations and code structures to optimize the utilization of GPU resources.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-ABS-735945).

References

[1] LLNL Support Libraries. Web Page. Accessed Wed Jul 19, 2017. https://wci.llnl.gov/simulation/support-libraries

[2] A.B. Caldeira, V. Haug and S. Vetter, "IBM Power System S822LC for High Performance Computing Introduction and Technical Overview," IBM Redbooks, October 2016.

[3] LLNL RZManta. Web Page. Accessed Fri 21, 2017. https://hpc.llnl.gov/hardware/platforms/RZManta

[4] J.A. Rathkopf, D.S. Miller, J.M. Owen, L.M. Stuart, M.R.Zika, P.G. Eltgroth, N.K. Madsen, K.P. McCandless, P.F. Nowak, M.K. Nemanic, N.A. Gentile, N.D. Keen and T.S. Palmer, "KULL: LLNL's ASCI Inertial Confinement Fusion Simulation Code," January 2000.

[5] OpenMP Application Programming Interface, Version 4.5 November 2015. Web Page. Accessed Tue 25, 2017. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[6] NVIDIA Visual Profiler. Web Page. Accessed Tue 25, 2017. https://developer.nvidia.com/nvidia-visual-profiler

[7] L. Grinberg, C. Bertolli, H. Riyaz, "Hands on with OpenMP4.5 and Unified Memory: Developing Applications for IBM's Hybrid CPU+GPU Systems (Part II)," 2017.