Evaluating the Performance of the hipSYCL Toolchain for HPC Kernels on NVIDIA V100 GPUs

Brian Homerding
bhomerding@anl.gov
Argonne National Laboratory
Lemont, IL

John Tramm
jtramm@anl.gov
Argonne National Laboratory
Lemont, IL

ABSTRACT

Future HPC leadership computing systems for the United States Department of Energy will utilize GPUs for acceleration of scientific codes. These systems will utilize GPUs from various vendors which places a large focus on the performance portability of the programming models used by scientific application developers. In the HPC domain, SYCL is an open C++ standard for heterogeneous computing that is gaining support. This is fueling a growing interest in understanding the performance of SYCL toolchains for the various GPU vendors.

In this paper, we produce SYCL benchmarks and mini-apps whose performance on the NVIDIA Volta GPU is analyzed. We utilize the RAJA Performance Suite to evaluate the performance of the hipSYCL toolchain, followed by an more detailed investigation of the performance of two HPC mini-apps. We find that the kernel performance from the SYCL kernels compiled directly to CUDA preform at a competitive level with their CUDA counterparts when comparing the straightforward implementations.

CCS CONCEPTS

• **Computing methodologies → Parallel programming languages**; • **Software and its engineering → Compilers**; • **Applied computing → Physical sciences and engineering**.

KEYWORDS

high performance computing, heterogeneity, GPU, SYCL, CUDA

ACM Reference Format:
Brian Homerding and John Tramm. 2020. Evaluating the Performance of the hipSYCL Toolchain for HPC Kernels on NVIDIA V100 GPUs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The end of Moore's law and the ever increasing demand for computational power in high performance computing has lead the United States Department of Energy leadership computing facilities to move towards heterogeneous systems utilizing GPU accelerators. While NVIDIA has been a leader in this space, the future Exascale systems Aurora, at Argonne National Laboratory, and Frontier, at Oak Ridge National Laboratory, will utilize GPU accelerators from Intel and AMD respectively.

Historically vendors have provided their own specific programming models for their hardware. However with the increased diversity of GPUs in HPC there has been increased interest in portable and open programming models. While various open programming standards are being explored by the HPC community, the performance portability of any programming model depends greatly of the strength of the toolchains available for any give platform. For an emerging standard to achieve widespread adaption, the level of support for the existing systems will greatly impact developers programming model decisions For this work we investigate the performance of the hipSYCL toolchain for running HPC SYCL code on the NVIDIA V100 GPU.

This paper makes the following contributions

(1) We collect performance data on a standard benchmark suite for the programming models and toolchains of interest
(2) We investigate performance significant performance differences found in the benchmark suite.
(3) We analyze and investigate the performance obtained for two HPC mini-apps of interest.

2 RELATED WORK

The growing interest in utilizing an open heterogeneous computing programming model that has strong support for standard C++ has fueled exploration into programming models and performance. There have been studies focused on the programming models themselves. Hammond, Kinsner and Brodman[7]compared and contrasted the Kokkos[4] programming models with a focus on how parallel execution is expressed and controlled. While Brown, Reyes and Wong[2] discussed these models with a focus on moving towards a unifed model of execution for the C++ standard.

Beyond this, there has been recent performance studies of HPC kernels across a variety of programming models and platforms. Reguly [9] collected performance data for several multi-material algorithms in order to understand the performance portability of each programming model. They found that the level of compiler support may lead to significant differences in performance. Joó et al.[5] studied the performance portability for a Wilson Dslash stencil operator mini-app. There investigation focuses on a comparison of the Kokkos programming model to SYCL, for collecting the performance of SYCL on the V100 NVIDIA GPU they utilized the POCL toolchain. Silva, Pisani and Borin [3] preformed a comparative study of SYCL, OpenCL and OpenMP. Their study found the SYCL based programs to be slower the OpenCL and OpenMP, though the gap in performance was shown to be closing from an earlier study[15].

3 TECHNOLOGIES

Parallel programming models for GPUs can impact performance through the control they offer for hiding data transfer costs and how well the parallelism can be mapped to the underlying hardware. For HPC, the quality of the generated code for execution on the device also impacts the overall performance of the code. The quality of the generated code can be impacted from the programming model from the information that is, or is not, available to make optimizations or simply from the maturity of the toolchain. We will compare the performance of HPC kernels of a relatively new programming model (SYCL) and an established one (CUDA) with a focus on a specific toolchain (hipSYCL). Our decision to focus on this toolchain is due to it being able to compile directly into CUDA code. We expect that the performance should be comparable due to this. This decision has the added benefit that it allows usto utilize the NVIDIA performance tools for our study.

3.1 SYCL

The SYCL standard builds on the underlying concepts of OpenCL while including the strengths of single-source C++ [12]. This includes a hierarchical parallelism syntax and separation of data access from data storage. This memory model manifests itself with implicit data transfer between the host and device. Users create buffer objects to manage the underlying data and then create accessor objects or the host or device. These accessor objects declare their type of access (read, write, read_write).

Knowing how memory is accessed allows the SYCL runtime to create a dependency graph for the data and utilize this graph to schedule kernels. Kernels are executed either with an `nd_range`, which organizes work items into work groups, or a simplified `range`, which executes over an iteration space of unspecified work group size.

3.2 CUDA

CUDA is a programming model created by NVIDIA which is designed to work with C, C++ and Fortran[8]. This makes the programming model accessible to HPC developers. It is designed to provide a scalable programming model by utilizing abstractions for the hierarchy of thread groups, shared memories and barrier synchronization.

CUDA kernels can utilize one, two or three-dimensional block of threads. This provides a clear mappings to the underlying hardware. Memory can be managed explicitly through the use of device memory allocation and deallocation along with data transfer capabilities.

3.3 hipSYCL

hipSYCL provides a SYCL 1.2.1 implementation that is built on top of NVIDIA CUDA/AMD HIP. It includes two components, a SYCL runtime that operates on top of the CUDA/HIP runtime and a compiler plugin to compile SYCL using the CUDA frontend of clang as shown in figure 1 . We chose to investigate the kernel performance using the hipSYCL toolchain specifically because it compiles directly to CUDA, which should enable SYCL code to be performance competitive with standard CUDA implementations. While we expect the performance to be competitive, the functionality supported is an important factor of analyzing a toolchain. hipSYCL[1] provides a reference to several of the important unimplemented SYCL features. These features are not implemented due to hipSYCL still being under development, to our understanding this is not because of any fundamental issue involved with compiling SYCL directly to CUDA/HIP.

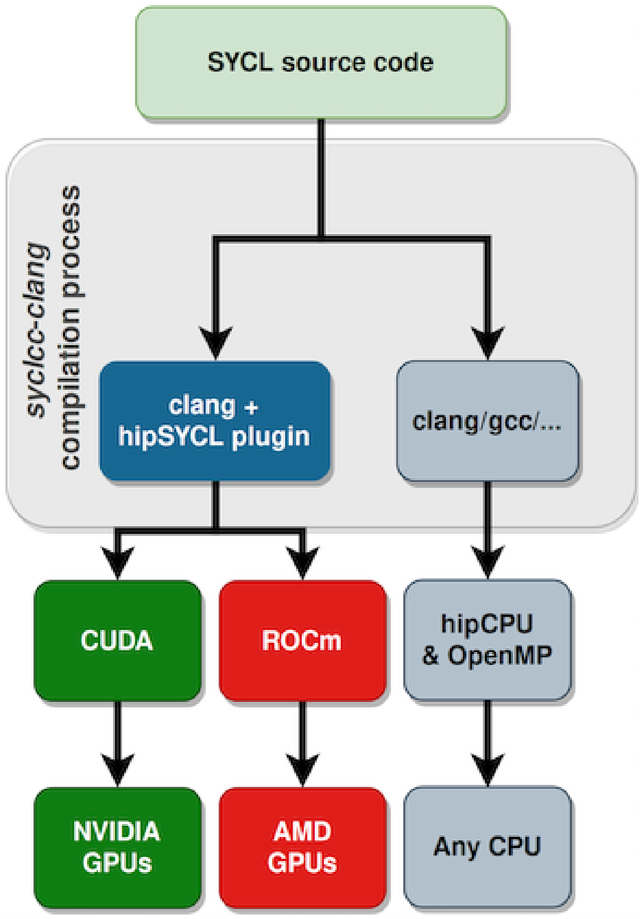

Figure 1: hipSYCL compilation model

4 TOOLCHAIN EVALUATION

While the discussion relating to the programming models becomes subjective with respect to programmer productivity, the performance metrics for programming evaluation often is dependent of the existing toolchains. The programming model will influence the rate at which the toolchains are developed along with several other factors, such as community and company investment and developer experience. With this in mind, we begin our investigate into the performance of the hipSYCL toolchain on NVIDIA GPUs by evaluating the performance using a standard compiler performance suite.

4.1 RAJA Performance Suite

To investigate the toolchain with respect to performance, we utilized the RAJA performance suite. The RAJA performance suite is developed to investigate the performance of various kernels which are of interest in high performance computing. These kernels are developed in a variety of programming models, referred to as variants. These variants include both traditional and RAJA versions of sequential, OpenMP threading, OpenMP offloading and CUDA kernels. The various kernels are broken into groups based on their origin. The simpler of these kernels are from the "Basic" group, which includes kernels such as DAXPY and initialization, and the "Stream" group, based on the traditional stream memory bandwidth benchmark. The complexity of the kernels grows from there to include the PolyBench collection of benchmarks and the Livermore Compiler Analysis Loop Suite (LCALS). Finally, the Apps group includes kernels pulled directly from scientific applications.

The RAJA performance suite produces reports which include timing information along with a checksum to verify the result of the various variants. Additionally, the kernels are run in an outer loop to collect many performance samples. The iteration count for this outer loop is dependent on the kernel. The inner loop which defines the amount of work for the kernel to perform is also set with a default specific to the kernel. For our investigation we focused on the base CUDA variant (CUDA without RAJA) along with a port to a base SYCL variant. The SYCL implementations were based on the CUDA implementations to keep the kernels useful for making a performance comparison between the two models. The RAJA performance suite base CUDA kernels all follow the same basic pattern which allowed the porting to SYCL to follow a similar effort for all kernels.

4.2 Programming Model Mapping

The basic abbreviated outline of the CUDA kernels in the RAJA performance suite is shown in Listing 1. This is a basic CUDA kernel with the memory management moved into a `#define`. The CUDA memory management APIs are abstracted into a common file in the performance suite to combine the repeated function calls.

Shown in Listing 2 is the corresponding SYCL example. While the porting is straight forward the implicit memory management model of SYCL imposed limitations on using the builtin timer functions from the performance suite. Without having a simple way to have the runtime trigger the data movement isolating the kernel execution time from the memory transfer cost. This is a concern for HPC as it becomes difficult to prefetch the data to the device as soon as it is prepared.

4.3 Benchmark Performance Analysis

For our analysis we are able to collect data for the kernel execution independent of memory transfer because of our decision to utilize the hipSYCL toolchain. As hipSYCL compiles directly to CUDA and the runtime sits on top of the CUDA runtime, we are able to use NVIDIA's performance analysis toolset. We used CUDA version 10.0.130, and the 12/10/2019 version of hipSYCL for our study.

Of the forty-one kernels in the performance suite, we present data on thirty-six kernels. This exposes some current functionality limitation in the hipSYCL compiler, namely the implementation of SYCL atomics and issues we have with creating multiple accessors from the same buffer with different offsets. In order to reduce noise and ensure sufficient work for the Volta GPU we scaled the work set of the kernels by a factor of five. Beyond the question of functionally being able to compile the SYCL functionality into CUDA code, when compiling directly to CUDA we should be able to obtain competitive performance with directly writing CUDA code. The performance data we collected is presented in figure 2. The figure shows the kernel speedup of the SYCL variant relative to the CUDA variant.

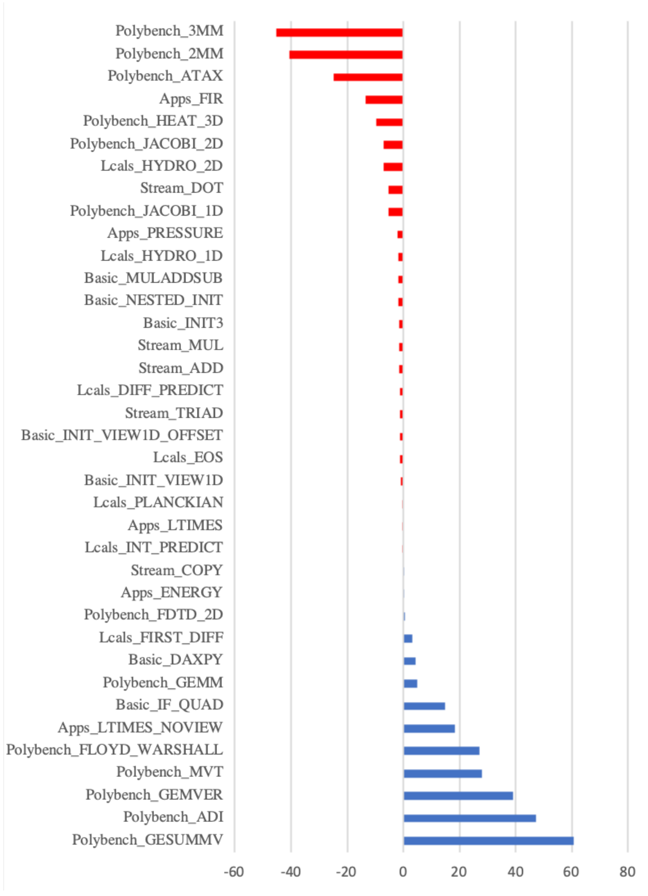

Figure 2: Percent speedup of SYCL kernels relative to CUDA kernels

The benchmarks show that the performance that several kernels have a performance difference above 20%, while most fall within a narrow range. In an attempt to understand the reason for these performance differences we collected additional performance data for six of the benchmarks. The six benchmarks whose preformed showed the greatest difference between the hipSYCL and CUDA toolchains were all from the PolyBench group of benchmarks.

Listing 1: CUDA Example

```
const size_t block_size = 256;

#define DATA_SETUP_CUDA \\
  Double a; \\
  cudaMalloc(a, iend); \\
  cudaMemcpy(a, m_a, iend);

#define DATA_TEARDOWN_CUDA \\
  cudaMemcpy(m_a, a, iend); \\
  cudaFree(a);

__global__ void example(double a) {
  size_t i = blockId.x * blockDim.x + threadIdx.x;
  if (i < iend) {
    EXAMPLE_BODY
  }
}

void EXAMPLE::runCudaVariant(VariantID vid) {
  const size_t iend = getRunSize();
  DATA_SETUP_CUDA;
  startTimer();

  for (size_t irep = 0; irep , num_reps; ++irep) {
    const size_t grid_size = DIVIDE_CEILING(iend,
         block_size);
    example<<<grid_size, block_size>>> (a, iend);
  }

  stopTimer();
  DATA_TEARDOWN_CUDA;
}
```

Listing 2: SYCL Example

```
const size_t block_size = 256;

#define DATA_SETUP_SYCL \\
  sycl::buffer<double> d_a {m_a, iend};

void EXAMPLE::runSyclVariant(VariantID vid) {
 { // Buffer Scope
  const size_t iend = getRunSize();
  DATA_SETUP_SYCL;
  startTimer();

  for (size_t irep = 0; irep , num_reps; ++irep) {
    const size_t grid_size = block_size *
                      DIVIDE_CEILING(iend, block_size);
    q.submit([&] (sycl::handler& h) {
      auto a =
           d_a.get_access<sycl::access::mode::read_write>(h);

      h.parallel_for<class EXAMPLE> (sycl::nd_range<1>
                                 {grid_size, block_size},
                           [=] (sycl::nd_item<1> item) {

        size_t i = item.get_group(0) *
             item.get_local_range().get(0) +
             item.get_local_id(0);
        if (i < iend) {
          EXAMPLE_BODY
        }
      });
    });
  }
 } // Buffer Destruction
  stopTimer();
}
```

With the exception of the ADI stencil kernel, the five kernels showing a significant performance delta are all linear algebra kernels. These performance differences all manifest as significantly different amounts of achieved memory throughput. We examined the ptx assembly produced from CUDA version and the SYCL version to attempt to understand the cause varying memory performance between the two programming models. One noticeable difference is the CUDA toolchains use of non-coherent memory loads in contrast to hipSYCL using coherent memory loads.

Overall our analysis utilizing the RAJA Performance Suite showed that, other than unsupported features, the hipSYCL toolchain produces code which performs at a competitive level with the comparable CUDA version. We also learned that for memory bound kernels there is an increased chance to observe a performance delta. These results gave us confidence to undertake a more in depth performance analysis utilizing more complex codes.

5 EXPERIMENTAL EVALUATION

In the previous section, we analyzed the performance of a wide range of generic computational kernels that are used by a variety of simulation applications. However, many of these kernels are extremely simple (sometimes as short as a few lines of code), which can make it easier for a toolchain to convert into performant code. However, to fully evaluate the relative performance characteristics of a toolchain, it is also important to test it on more complex HPC simulation kernels that may be more challenging for a compiler to handle.

5.1 N-Body

In addition to our investigate of the performance of a performance benchmark suite, we investigated the hipSYCL toolchain performance using a simple N-body simulation code. The N-body simulation is of interest for astrophysics for the simulation of a dynamical system of particles. For each particle in the simulation the velocity, acceleration and mass are stored in a C-like structure. For each time step of the simulation the code the velocity for each particles is updated using finite difference methods by offloading to the device. This allows for the control of the amount of work through the scaling of the number of particles. We evaluated this kernel's performance by running the simulation with 400,000 particles and

500 integration steps. The simple implementation of this N-body simulation code is limited in how representative it is of production codes, specifically production codes of interest will preform a more complex calculation resulting in increased floating point work for the kernel. This limitation is kept in consideration for our performance analysis.

5.1.1 Results. The results, shown in Figure 3, show the performance that was obtained using the hipSYCL toolchain for SYCL and the CUDA toolchain. This figure shows the average execution time of the offloaded kernel for our problem of interest. We find that the SYCL port of the code is running 16% slower than the CUDA version when executed with a similar pattern. While a performance gap exists, we consider the performance to be competitive with the CUDA version.

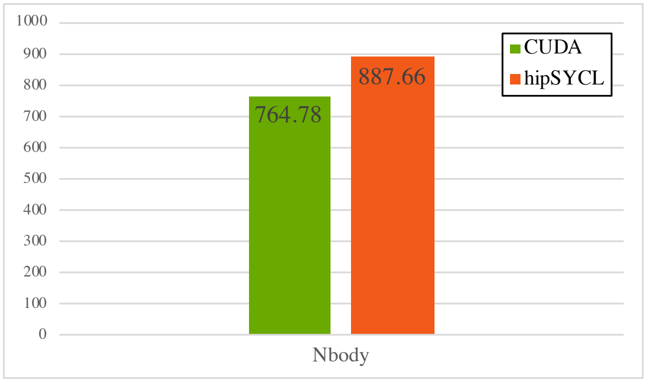

Figure 3: Performance of N-Body kernel on V100 GPU.

We proceeded to investigate the performance difference by using the NVIDIA profiling tool to collect performance metrics. Both versions show a similar performance profile with respect to occupancy, stall reasons, floating point operations. However the SYCl version compiled using the hipSYCL toolchain executes a significant amount of additional miscellaneous instructions. This manifests in the performance data with the SYCL version of the code executing 437,270 instructions per warp on average, while the CUDA version executes 334,560 instructions per warp on average. These additional instructions increase the amount of work,and the overall execution time.

5.2 XSBench

Monte Carlo (MC) algorithms are a great candidate for testing programming models as their stochastic nature often challenges toolchains in ways that more traditional linear algebra kernels may not. In particular, MC particle transport applications like OpenMC [10, 11] and Shift [6] are slated to be used extensively on Exascale supercomputers, making them an excellent target for programming model testing. However, the programmer-hour cost of porting full scale applications into a variety of GPU programming models for comparison is prohibitively high. Thankfully, there exists a much smaller mini-app XSBench [14] that has been shown to serve as an excellent stand-in for full scale MC apps in the context of performance analysis on HPC node architectures.

The XSBench mini-app represents the dominant kernel for a specific use case of Monte Carlo transport – the transport of neutrons through a partially depleted nuclear reactor featuring hundreds of fuel nuclides, making it a very challenging problem computationally. The kernel XSBench represents, known as the macroscopic cross section kernel, can account for up to 85% of the runtime of the MC reactor criticality simulations [14]. By abstracting this kernel into a mini-app, the essential computational conditions and tasks of fully featured MC transport codes are largely retained in the kernel, without the additional complexity of the full application. This provides a more transparent platform for isolating where both hardware and software bottlenecks degrade the performance of the algorithm. XSBench features a default problem size that represents the single node load of a full core nuclear reactor simulation.

Due to the greatly simplified nature of XSBench, ports already exist for it using OpenMP threading, OpenMP target offloading, CUDA, SYCL, and OpenCL [13]. The various ports are all written by the same author and were written with a similar level of optimization. In particular, the code was written in all programming models with general, portable optimizations in mind, but do not use programming model specific or vendor specific intrinsics, so as to make for an even playing field.

It is important to node that XSBench features three different methods for executing the same kernel, all of which are commonly options for use in full applications like OpenMC. While they all produce the same results, they build and utilize different types of acceleration structures to speed up macroscopic cross section lookups. These different acceleration structures attempt to improve performance at the cost of increases in the overall memory footprint. Their relative effectiveness in improving performance often varies significantly between computational architectures such that the most optimal method for a given architecture may be different than a different one. Thus, in this study, we will test all three of the methods available in XSBench. The "Nuclide Grid" method of lookup uses the lest amount of memory (about 184 MB). The "Hash Grid" method of lookups adds a very small acceleration structure, raising overall memory usage slightly to about 198 MB. The "Unionized Grid" method adds a very large acceleration structure, increasing memory usage to about 5,650 MB. All of these methods fit well within the 32 GB available on our V100 GPU.

5.2.1 Results. In this study, we will compare the performance of the SYCL and CUDA ports of XSBench against each other when run on a system featuring four NVIDIA V100 GPU and two Intel Xeon Gold 6152 CPUs (44 cores total). All tests were performed using a single CPU host thread and a single V100 GPU. In both cases, the `-O3` optimization level was used. The performance figure of merit (FOM) in XSBench is given as the number of macroscopic cross section lookups executed per second. The FOM was calculated in all cases using timing data provided by the NVIDIA `nvprof` profiling tool, with final values given as the average of 25 independent runs.

The results, shown in Figure 4, show all three different lookup methods in XSBench for both the hipSYCL toolchain using the SYCL port and the CUDA toolchain using the CUDA port. One interesting result that we found immediately was that the baseline CUDA port ran about 25% slower than the hipSYCL port when using the unionized lookup method. This result was highly surprising, given that CUDA is a very mature vendor-specific toolchain whereas the hipSYCL toolchain is much newer and is still in active development.

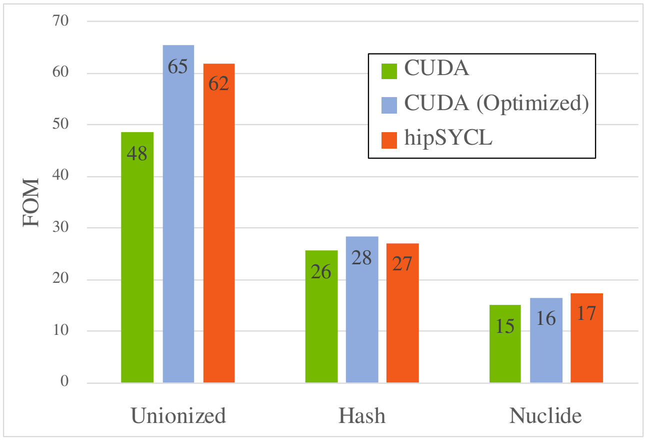

Figure 4: Performance of various cross section lookup methods and programming models in XSBench on a V100 GPU.

In an effort to determine the cause of the performance discrepancy between the CUDA and hipSYCL toolchains, we examined the PTX assembly that was generated by each toolchain. Specifically, we examined some of the most costly lines (as indicated in the NVIDIA visual profiler) and their assembly, and found that CUDA had generated a very different ordering of instructions. In particular, we found that the innermost portion of the kernel required 12 memory loads and a similar number of floating point operations. These 12 loads loads in global memory are very likely to be cache misses (due to the stochastic nature of the MC algorithm). We found that hipSYCL arranged instructions so as to perform all memory loads needed by the kernel first, before issuing any floating point instructions. Conversely, CUDA broke the 12 memory loads up into 6 disjoint blocks, interspersed with floating point instructions – probably in an effort to reduce the number of registers used by the kernel. This essentially caused the CUDA generated version to pay a higher latency cost, thus resulting in lower bandwidth usage and a longer time to solution.

To verify that this difference was indeed what was causing the performance discrepancy, we created a second "optimized" CUDA kernel that utilized the CUDA "`__ldg()`" intrinsic to force load instructions to be generated for all required data before floating point work is performed in the inner kernel. Figure 4 shows that the optimized CUDA version is about as fast as the hipSYCL version. Therefore, for this specific kernel, we found that hipSYCL was able to provide a higher performance solution than the CUDA toolchain when used naively. However, if more expert level vendor-specific intrinsics were used in CUDA, performance could be improved to reach parity with hipSYCL. We believe it is a very promising result for the hipSYCL toolchain to achieve such high performance using only a straightforward baseline SYCL implementation that does not require any vendor specific intrinsics or expert level optimizations.

6 CONCLUSION & FUTURE WORK

To investigate the performance that can be obtained today using the hipSYCL toolchain for SYCL kernels on NVIDIA GPUs we evaluated the RAJA Performance Suite and preformed an in depth analysis of two kernels of interest to the HPC community. While we found that there is missing functionality, overall the performance of running SYCL by compiling directly in CUDA is competitive with using CUDA directly. Many of the performance differences we found were due to the ordering and choices of how to load memory. We find that this compilation is a promising path for achieving performance on current NVIDIA devices.

We intend to extend this work to include additional SYCL compilation toolchains for GPUs from different vendors. This future work will provide us with a more complete understanding of the level of support for SYCL across GPUs. We will also, when available, explore the performance impact of utilizing the Intel's SYCL extensions.

7 ACKNOWLEDGEMENTS

This work was supported by the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We also gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Thanks also to Paulius Velesko at Intel for assistance with the nbody SYCL implementation.

REFERENCES

[1] Aksel Alpay. 2019. hipSYCL. https://github.com/illuhad/hipSYCL
[2] Gordon Brown, Ruyman Reyes, and Michael Wong. 2019. Towards Heterogeneous and Distributed Computing in C++. In *Proceedings of the International Workshop on OpenCL*. ACM, New York, NY, USA.
[3] Hércules Cardoso da Silva, Flávia Pisani, and Edson Borin. 2016. A Comparative Study of SYCL, OpenCL, and OpenMP. In *Int. Symp. on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. IEEE.
[4] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202 – 3216. https://doi.org/10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.
[5] Joó et al. 2019. Performance portability of a Wilson Dslash Stencil Operator Mini-App using Kokkos and SYCL. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC*. IEEE, Denver, CO, USA.
[6] Steven P. Hamilton and Thomas M. Evans. 2019. Continuous-energy Monte Carlo neutron transport on GPUs in the Shift code. *Annals of Nuclear Energy* 128 (2019), 236–247. https://doi.org/10.1016/j.anucene.2019.01.012
[7] Jeff Hammmond, Michael Kinsner, and James Brodman. 2019. A comparative analysis of Kokkos and SYCL as heterogeneous, parallel programming models for C++ applications. In *Proceedings of the International Workshop on OpenCL*. ACM, New York, NY, USA.
[8] NVIDIA Corporation. 2020. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[9] István Z. Reguly. 2019. Performance Portability of Multi-Material Kernels. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC*. IEEE, Denver, CO, USA.
[10] Paul K. Romano and Benoit Forget. 2013. The OpenMC Monte Carlo particle transport code. *Annals of Nuclear Energy* 51 (2013), 274 – 281. https://doi.org/10.1016/j.anucene.2012.06.040
[11] Paul K. Romano, Nicholas E. Horelik, Bryan R. Herman, Adam G. Nelson, Benoit Forget, and Kord Smith. 2015. OpenMC: A state-of-the-art Monte Carlo code for research and development. *Annals of Nuclear Energy* 82 (2015), 90 – 97. https://doi.org/10.1016/j.anucene.2014.07.048 Joint International Conference on

Supercomputing in Nuclear Applications and Monte Carlo 2013, SNA + MC 2013. Pluri- and Trans-disciplinarity, Towards New Modeling and Numerical Simulation Paradigms.
[12] Khronos OpenCL Working Group SYCL subgroup. 2018. SYCL Specification.
[13] John R. Tramm. 2020. XSBench: The Monte Carlo macroscopic cross section lookup benchmark. https://github.com/ANL-CESAR/XSBench
[14] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench-the development and verification of a performance abstraction for Monte Carlo reactor analysis. In *PHYSOR 2014: The Role of Reactor Physics toward a Sustainable Future.* https://www.mcs.anl.gov/papers/P5064-0114.pdf
[15] Angelos Trigkas. 2014. *Investigation of the OpenCL SYCL Programming Model*. Master's thesis. The University of Edinburgh, Edinburgh, UK.