Achieving Near-Native Runtime Performance and Cross-Platform Performance Portability for Random Number Generation Through SYCL Interoperability

Vincent R. Pascuzzi[1(✉)] and Mehdi Goli[2]

[1] Brookhaven National Laboratory, Upton, NY 11973, USA
pascuzzi@bnl.gov
[2] Codeplay Software Ltd., Edinburgh EH3 9DR, UK
mehdi.goli@codeplay.com

Abstract. High-performance computing (HPC) is a major driver accelerating scientific research and discovery, from quantum simulations to medical therapeutics. While the increasing availability of HPC resources is in many cases pivotal to successful science, even the largest collaborations lack the computational expertise required for maximal exploitation of current hardware capabilities. The need to maintain multiple platform-specific codebases further complicates matters, potentially adding constraints on machines that can be utilized. Fortunately, numerous programming models are under development that aim to facilitate portable codes for heterogeneous computing. One in particular is SYCL, an open standard, C++-based single-source programming paradigm. Among the new features available in the most recent specification, SYCL 2020, is interoperability, a mechanism through which applications and third-party libraries coordinate sharing data and execute collaboratively. In this paper, we leverage the SYCL programming model to demonstrate cross-platform performance portability across heterogeneous resources. We detail our NVIDIA and AMD random number generator extensions to the oneMKL open-source interfaces library. Performance portability is measured relative to platform-specific baseline applications executed on four major hardware platforms using two different compilers supporting SYCL. The utility of our extensions are exemplified in a real-world setting via a high-energy physics simulation application. We show the performance of implementations that capitalize on SYCL interoperability are at par with native implementations, attesting to the cross-platform performance portability of a SYCL-based approach to scientific codes.

Keywords: performance portability · HPC · SYCL · random number generators · high energy physics · simulation

S. Bhalachandra et al. (Eds.): WACCPD 2021, LNCS 13194, pp. 22–45, 2022.
https://doi.org/10.1007/978-3-030-97759-7_2

1 Introduction

The proliferation of heterogeneous platforms in high performance computing (HPC) is providing scientists and researchers opportunities to solve some of the world's most important and complex problems. Coalescing central processing units (CPU), co-processors, graphics processing units (GPU) and other hardware accelerators with high-throughput inter-node networking capabilities has driven science and artificial intelligence through insurmountable computational power. Industry continues to innovate in the design and development of increasingly performant architectures and platforms, with each vendor typically commercializing a myriad of proprietary libraries optimized for their specific hardware. What this means for physicists and other domain scientists is that their codes need to be translated, or ported, to multiple languages, or adapted to some specific programming model for best performance. While this could be a useful and instructive exercise for some, many are often burdened by their limited numbers of developers that can develop such codes. Fortunately, as a result of the numerous architectures and platforms, collaborative groups within academia, national laboratories and even industry are developing portability layers atop common languages that aim to target a variety of vendor hardware. Such examples include Kokkos [21] (Sandia National Laboratory, USA), RAJA [25] (Lawrence Livermore National Laboratory, USA) and SYCL [12] (Khronos Group).

Mathematical libraries are crucial to the development of scientific codes. For instance, the use of random numbers in scientific applications, in particular high energy physics (HEP) software, is almost ubiquitous [26]. For example, HEP experiments typically have a number of steps that are required as part of their Monte Carlo (MC) production: event generation, simulation, digitization and reconstruction. In the first step, an MC event generator [17] produces the outgoing particles and their four-vectors given some physical process. Here, random numbers are used, *e.g.*, to sample initial state kinematics and evaluate cross sections. Simulation software, *e.g.*, Geant4 [15] and FastCaloSim [20,31] from the ATLAS Experiment [14], require large quantities of random numbers for sampling particle energies and secondary production kinematics, and digitization requires detector readout emulation, among others. With the rise of machine learning, random number production is required even at the analysis level [22].

1.1 Contribution

The focus of this paper is to evaluate the cross-platform performance portability of SYCL's interoperability functionality using various closed-source vendor random number generation APIs within a single library, and analyze the performance of our implementation in both artificial and real-world applications.

To achieve this, we have:

- integrated AMD and NVIDIA random number generators (RNG) within the oneMKL open-source interfaces library by leveraging existing hipRAND and cuRAND libraries, to target these HPC hardware from these vendors from a single API via SYCL interoperability;

- evaluated the performance portability of the API on Intel and AMD CPUs, and Intel, AMD and NVIDIA GPUs to investigate the performance overhead of the abstraction layer introduced by the SYCL API;
- integrated our RNG implementations into FastCaloSim to further investigate the applicability of the proposed solution on an existing real-world application for high-energy physics calorimeter simulations, which currently relies on separate implementations based on vendor-dependent libraries; and
- analyzed the cross-platform performance portability by comparing the SYCL-based implementation of FastCaloSim to the original C++-based and CUDA codes, which use native vendor-dependent RNGs, to investigate possible performance overheads associated with SYCL interoperability.

Our work utilizes Data Parallel C++ (DPC++) [6] and hipSYCL [16], two different existing LLVM-based SYCL compilers, capable of providing plug-in interfaces for CUDA and HIP support as part of SYCL 2020 features that enable developers to target NVIDIA and AMD GPUs, respectively.

The rest of this paper is organized as follows. Section 2 discusses existing parallel programming models and libraries providing functionalities used in scientific applications, along with our proposed solution to target the cross-platform portability issue. Section 3 briefly introduces the SYCL programming model used in this work. In Sect. 4, we discuss more technically the aspects and differences between the cuRAND and hipRAND APIs, and also detail the implementation of our work. Benchmark applications are described in Sect. 5 and performance portability in Sect. 6. The results of our studies are presented in Sect. 7. Lastly, Sect. 8 summarizes our work and suggests potential extensions and improvements for future developments.

2 Related Work

2.1 Parallel Programming Frameworks

Parallelism across a variety of hardware can be provided through a number different parallel frameworks, each having a different approach and programming style. Typically written in C or C++, each framework provides different variations on the language, allowing programmers to specify the task parallel patterns.

Introduced by Intel, Thread Building Blocks (TBB) [30] provides a C++-based template library supporting parallel programming on multi-core processors. TBB only support parallelism on CPUs, hence, parallel applications dependent on TBB cannot be directly ported to GPUs or any other accelerator-based platform.

NVIDIA's CUDA [9] API is a C/C++-based low-level parallel programming framework exclusively for NVIDIA GPUs. Its support of C++-based template meta programming features enables CUDA to provide performance portability across various NVIDIA devices and architectures, however, its lack of portability

across other vendor hardware can be a barrier for research groups with access to non-NVIDIA resources.

OpenCL [33], from the Khronos Group, is an open-standard cross-platform framework supported by various vendors and hardware platforms However, its low-level C-based interface and lack of support by some vendors could hinder the development of performance portability on various hardware. Also from the Khronos Group is SYCL [12], an open-standard C++-based programming model that facilitates the parallel programming on heterogeneous platforms. SYCL provides a single-source abstraction layer enabling developers to write both host-side and kernel code in the same file. Employing C++-based template programming, developers can leverage higher level programming features when writing accelerator-enabled applications, having the ability to integrate the native acceleration API, when needed, by using the different interoperability interfaces provided.

The Kokkos [21] and RAJA [25] abstraction layers expose a set of C++-based parallel patterns to facilitate operations such as parallel loop execution, reorder, aggregation, tiling, loop partitioning and kernel transformation. They provide C++-based portable APIs for users to alleviate the difficulty of writing specialized code for each system. The APIs can be mapped onto a specific backend—including OpenMP, CUDA, and more recently SYCL—at runtime to provide portability across various architectures.

2.2 Linear Algebra Libraries

There are several vendor-specific libraries which provide highly optimized linear algebra routines for specific hardware platforms. The ARM Compute Library [13] provides a set of optimized functions for linear algebra and machine learning optimized for ARM devices. Intel provides MKL [5] for its linear algebra subroutines for accelerating BLAS, LAPACK and RNG routines targeting Intel chips, and NVIDIA provides a wide ecosystem of closed source libraries for linear algebra operations, including cuBLAS [8] for BLAS routines, cuRAND [10] for RNG and cuSPARSE [11] for sparse linear algebra. AMD offers a set of hipBLAS [1] and hipRAND [2] libraries atop the ROCm platform, which provide linear algebra routines for AMD GPUs. Each of these libraries is optimized specifically for particular hardware architectures, and therefore do not provide portability across vendor hardware.

oneMKL [7] is an community-driven open-source interface library developed using the SYCL programming model, providing linear algebra and RNG functionalities used in various domains such as high-performance computing, artificial intelligence and other scientific domains. The front-end SYCL-based interface could be mapped to the vendor-optimized backend implementations either via direct SYCL kernel implementations or SYCL interoperability using built-in vendor libraries to target various hardware backends. Currently, oneMKL supports BLAS interfaces with vendor-optimized backend implementations for Intel GPU and CPU, CUDA GPUs and RNG interfaces which wrap the optimized Intel routines targeting x86 architectures and Intel GPUs.

2.3 The Proposed Approach

There are numerous highly-optimized libraries implemented for different device-specific parallel frameworks targeting different hardware architectures and platforms. Several parallel frameworks provide parallel models which hide the memory hierarchies and execution policies on different hardware. This can be due to a lack of a common language to abstract away the memory and execution models from various heterogeneous devices, hence, leaving cross-platform performance portability of high-level applications a challenging issue and an active area of research. Recent work in adopting SYCL [18,19,32] as the unifying programming model has shown to be a viable approach for developing cross-platform performance portable solutions targeting various hardware architectures while sharing the same interface. More specifically, SYCL interoperability with built-in kernels enables vendors to use a common unifying interface, to "glue-in" their optimized hardware-specific libraries.

In this paper, we leverage the SYCL programming model and interoperability to enable cross-platform performance portable random number generator targeting major HPC hardware, including NVIDIA and AMD GPUs. The proposed solution has been integrated into the oneMKL open-source interfaces library as additional backends targeting these vendors, extending the library's portability and offering nearly native performance. The applicability of the proposed approach was further studied in a high-energy physics calorimeter simulation software to evaluate the performance of the proposed abstraction method on a real-world scientific application.

3 SYCL Overview

SYCL is an open-standard C++-based programming model that facilitates parallel programming on heterogeneous platforms. It provides a single source programming model, enabling developers to write both host-side and kernel code in the same file. Employing C++-based template programming, developers can leverage higher-level programming features when developing accelerator-enabled applications. Developers also have the ability to integrate the native acceleration API, when needed, by using the different interoperability interfaces provided by SYCL.

A SYCL application is structured in three code scopes that control the flow, as well as the construction and lifetimes of the various objects used within it.

- *Application scope*: all code outside of a command group scope
- *Command group scope*: specifies a unit of work that is comprised of a kernel function and data accessors
- *Kernel scope*: specifies a single kernel function to interface with native objects and is executed on the device

To execute a SYCL kernel on an accelerator device, *command groups* containing the kernel must be submitted to a SYCL queue. When a command group is

submitted to a queue, the SYCL runtime system tracks data dependencies and creates (expands) a new (existing) dependency graph—a directed acyclic graph (DAG)—to orchestrate kernel executions. Once the dependency graph is created, the correct ordering of kernel execution on any available device is guaranteed by the SYCL runtime system via a set of rules defined for dependency checking[1].

Interoperability is enabled via the aforementioned low-level APIs by facilitating the SYCL runtime system's interaction with native objects for the supported backends [12,23].

SYCL interoperating with existing native objects is supported by either `host_task` or `interop_task` interfaces inside the command group scope. When using the `interop_task` interface, the SYCL runtime system injects a task into the runtime DAG that will execute from the host, but ensures dependencies are satisfied on the device. This allows code within a kernel scope to be written as though it were running directly at the low-level API on the host, but produces side-effects on the device, *e.g.*, external API or library function calls.

There are several implementations of SYCL API available including ComputeCpp [3] that currently supports the SYCL 1.2.1 specification, DPC++ and hipSYCL which incorporate SYCL 2020 features, such as unified shared memory (USM), and triSYCL [24] which provides SYCL supports for FPGAs.

4 SYCL-Based RNG Implementations of NVIDIA and AMD GPUs in oneMKL

4.1 Technical Aspects

The integration of third-party RNG backends within oneMKL depends primarily on compiler support for (a) SYCL 2020 interoperability and (b) generating the specific intermediate representation for a given architecture's source code. Hence, to enable RNG on NVIDIA and AMD GPUs, one requires SYCL compilers supporting parallel thread (PTX) and Radeon Open Compute (ROCm) execution instruction set architectures which are used in the CUDA and AMD programming environment, respectively. At present, PTX support is available in Intel's open-source LLVM project, and the ROCm backend is supported by the hipSYCL LLVM project.

The oneMKL interface library provides both buffer and USM API implementations for memory management. Buffers are encapsulating objects which hide the details of pointer-based memory management. They provide a simple yet powerful way for the SYCL runtime system to handle data dependencies between kernels, both on the host and device, when building the data-flow DAG. The USM API gives a more traditional pointer-based approach, *e.g.*, memory allocations performed with `malloc` and `malloc_device`, familiar to those accustomed to C++ and CUDA. However, unlike buffers, the SYCL runtime system cannot generate the data dependency graph from USM alone, and so it is the user's responsibility to ensure dependencies are met. The ability for SYCL to

[1] This is not the case when using unified shared memory, as explained later.

internally satisfy buffer-based data dependencies is beneficial in cases when quick prototyping is, to first order, more important than optimizing. Figure 1 represents the architectural view of the cuRAND and hipRAND integration for each scope in the SYCL programming model for both buffer-based approach and USM-based approaches.

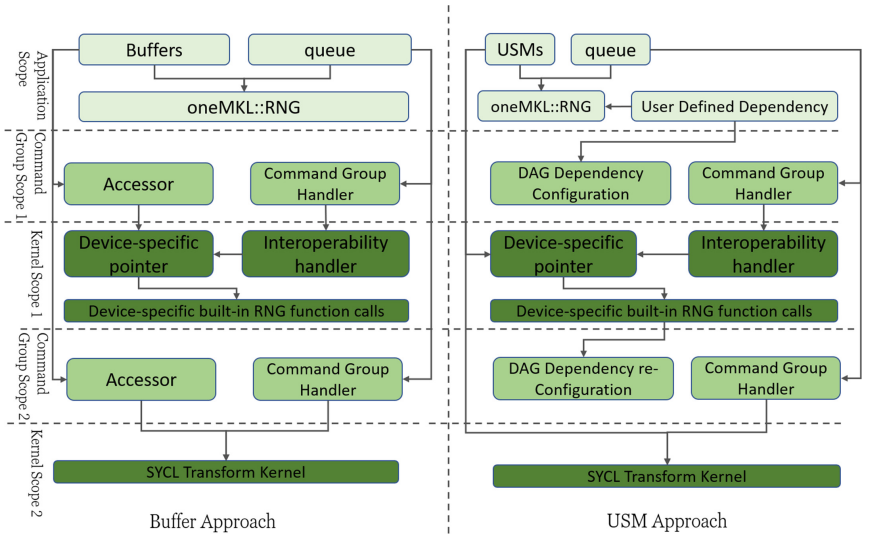


Fig. 1. Architectural view of device-specific RNG kernels integration in oneMKL for both cuRAND and hipRAND on different scopes in SYCL programming model using both buffer and USM approach.

The oneMKL library currently contains implementations for Philox- and MRG-based generators for ×86 and Intel GPUs. In oneMKL, each engine class comprises 36 high-level `generate` function templates—18 per buffer and USM API—with template parameters to specify a distribution and output types. In addition to having the ability to specify distribution properties, *e.g.*, mean, standard deviation for Gaussian distributions, custom ranges on the generated numbers can also be specified. This is in sharp contrast to the lower level interfaces provided by cuRAND or hipRAND; generation of random numbers is performed using functions with fixed types, and there is no concept of a "range", and it is therefore left to the user to post-process the generated numbers. For example, `curandGenerateNormal` will output a sequence of normally-distributed pseudo-random numbers in $[0, 1)$ and there is no API functionality to transform the range. As such, native cuRAND and hipRAND support generation of strictly positive-valued numbers.

Lastly, whereas oneMKL provides copy-constructors and constructors for setting seed initializer lists for multiple sequences, cuRAND and hipRAND do not. The oneMKL library also supports inverse cumulative distribution function (ICDF) methods for pseudorandom number generation, while such methods are available only for quasirandom number generators in the cuRAND and hipRAND API.

4.2 Native cuRAND and hipRAND flow

Generation of random numbers with cuRAND and hipRAND host APIs in native applications typically has the following workflow:

1. the creation of a generator of a desired type;
2. setting generator options, *e.g.*, seed, offset, *etc.*;
3. allocation of memory on the device using {`cuda, hip`}`Malloc`;
4. generation of the random numbers using a generation function, *e.g.*, {`cu, hip`}`randGenerate`; and
5. clean up by calling the generator destructor {`cu, hip`}`randDestroyGenerator` and {`cuda, hip`}`Free`.

In addition, a user may wish to use the generated numbers on the host, in which case host memory must also be allocated and data transferred between devices.

4.3 Implementation of cuRAND and hipRAND in oneMKL

Our implementation of cuRAND and hipRAND libraries within oneMKL follows closely the procedure outlined in Sect. 4.2. We also include additional *range transformation* kernels for specifying the output sequence of random numbers, a feature not available in the cuRAND and hipRAND APIs.

Each generator class comprises a native `xrandGenerator_t` object, where `xrand` could be either of `curand` or `hiprand`. Class constructors create the generator via a native `xrandCreateGenerator` API call and sets the seed for generation of the output sequence with `xrandSetPseudoRandomGeneratorSeed`; due to limitations of the cuRAND and hipRAND host API, our implementation does not support copy-construction or seed initializer lists. Of the total 36 `generate` functions available in oneMKL, 20 are supported by our cuRAND and hipRAND backends as the remaining 16 use ICDF methods (see Sect. 4.1). Each `generate` function in the cuRAND and hipRAND backends have the same signature as the corresponding ×86 and Intel GPU function to facilitate "pointer-to-implementation".

The buffer and USM API `generate` function implementations are nearly identical; access to the buffer pointer via a SYCL `accessor` is needed before retrieving the native CUDA memory.

```
virtual inline void generate(
  const oneapi::mkl::rng::uniform<float, uniform_method::standard>& distr,
  std::int64_t n, cl::sycl::buffer<float, 1>& r) override {
    queue_.submit([&](cl::sycl::handler& cgh) {
      auto acc = r.get_access<cl::sycl::access::mode::read_write>(cgh);
      cgh.codeplay_host_task([=](cl::sycl::interop_handle ih) {
      auto r_ptr = reinterpret_cast<float*>(
        ih.get_native_mem<cl::sycl::backend::cuda>(acc));
      curandStatus_t status;
      CURAND_CALL(curandGenerateUniform, status, engine_, r_ptr, n);
      cudaError_t err;
      CUDA_CALL(cudaDeviceSynchronize, err);
    });
  });
  range_transform_fp<float>(queue_, distr.a(), distr.b(), n, r);
}
```

Listing 1.1. Example code calling functions from the cuRAND library within a SYCL kernel using the buffer API.

```
template <typename T>
static inline void range_transform_fp(cl::sycl::queue& queue, T a, T b,
                                      std::int64_t n,
                                      cl::sycl::buffer<T, 1>& r) {
  queue.submit([&](cl::sycl::handler& cgh) {
    auto acc =
      r.template get_access<cl::sycl::access::mode::read_write>(cgh);
    cgh.parallel_for(cl::sycl::range<1>(n), [=](cl::sycl::id<1> id) {
      acc[id] = acc[id] * (b - a) + a;
    });
  });
}
```

Listing 1.2. Example code of transform function for cuRAND using the buffer API. The function can be used to transform the range of the generated numbers. Its dependencies are detected via the auto-generated runtime DAG graph from SYCL accessors.

As shown in Fig. 1, cuRAND and hipRAND backend integration into the oneMKL open-source interfaces library requires two kernels. The first kernel makes the corresponding `xrandGenerate` third-party library function call, as per the distribution function template parameter type; Listing 1.1 shows an example kernel for the cuRAND backend using the buffer API. A second kernel is required to adjust the range of the generated numbers, altering the output sequence as required. As this is not a native functionality in the cuRAND and hipRAND APIs, we implemented this it as a SYCL kernel. Listing 1.2 gives an example of one such transformation kernel for floating-point data types using the buffer API. It is hardware agnostic: the same code can be compiled for, and executed on, all platforms for which there exists a SYCL compiler. In the command group scope, an `accessor` is required for the buffer API to track the kernel dependency and memory access within the kernel scope. In this case, the graph dependencies between the two kernels are automatically detected by the SYCL runtime system scheduling thread, tracking the data-flow based on the data access type, *e.g.*, `read`, `write`, `read_write`. The `accessor` has a `read_write` access type and is passed as an input with `read_write` for *in-situ* updates to be made. This forces the transformation kernel to depend on the SYCL interoperability kernel and hence the kernels will be scheduled for execution in this order.

The USM API does not require `accessor`s in the command group scope, but does take an additional argument for specifying dependent kernels for subsequent calculations on the data outputted. The dependency is preserved by a direct injection of the `event` object returned by the command group handler to the existing dependency list.

Inside the kernel scope for both buffer and USM APIs, calls to the cuRAND or hipRAND API are made from the host and, if using buffers, the `accessor` is then reinterpreted as native memory—*i.e.*, a raw pointer to be used for cuRAND and hipRAND API calls. The random numbers are then generated by calling the appropriate `xrandGenerate` as per the distribution function template parameter type.

The application scope remains the same as the one proposed in the oneMKL SYCL RNG interface for both buffer and USM API, enabling users to seamlessly execute codes on AMD or NVIDIA GPUs with no code modification whatever.

5 Benchmark Applications

Two benchmark applications were used for performance portability studies, and are detailed below. The SYCL codes were compiled using the `sycl-nightly-20210330` tag of the Intel LLVM open-source DPC++ compiler for targeting CUDA devices and hipSYCL v0.9.0 for AMD GPUs. The applications' native counterparts were compiled with `nvcc` 10.2 and `hipcc` 4.0, respectively, for NVIDIA and AMD targets. Calls to the high-resolution `std::chronos` clock were bootstrapped at different points of program execution to measure the execution time of different routines in the codes.

5.1 Random Number Generation Burner

The first application was designed as an artificial benchmark to stress the hardware used in the experiments by generating a sequence of pseudorandom numbers of a given batch size using a specified API—*i.e.*, CUDA, HIP or SYCL—and platform. We use this simple test as the primary measure of our oneMKL RNG implementations. Having a single application to benchmark all available platforms has a number of advantages, namely, ensuring ease of consistency among the separate target platform APIs, *e.g.*, all memory allocations, and data transfers between host and devices are performed analogously for each API.

The workflow of this benchmark application can be outlined as follows:

1. target platform, API and generator type are chosen at compile-time, specified by `ifdef` directives;
2. target distribution from which to sample, number of iterations and cardinality of the output pseudorandom sequence are specified at runtime; for SYCL targets, buffer or USM API is also specified;

3. host and device memory are allocated, and the generator is constructed and initialized; for SYCL targets, a distribution object is also created as per Step 2 above;
4. pseudorandom output sequence is generated and its range is transformed; and
5. the output sequence is copied from device memory to host memory.

5.2 FastCaloSim

Our second benchmark is a real-world application that aims to solve a real-world problem: rapid production of sufficiently accurate high-energy collider physics simulations[2]. The parameterized calorimeter simulation software, FastCaloSim [31], was developed by the ATLAS Experiment [14] for this reason. The primary ATLAS detector comprises three sub-detectors; from inner radii outward, a silicon-based inner tracking detector; two types of calorimeter technologies consisting of liquid argon or scintillating tiles for measurements of traversing particles' energies; and at the periphery a muon spectrometer. Among these three sub-detectors, the simulation of the calorimeters are the most CPU-intensive due to the complex showering—*i.e.* production of additional particles in particle-material interactions—and stopping of highly energetic particles, predominantly in the liquid argon calorimeters.

The original FastCaloSim codes, written in standard C++, were ported to CUDA and Kokkos [20], and subsequently to SYCL; the three ports were written to be as similar as possible in their kernels and program flow so as to permit comparisons between their execution and runtimes. The SYCL port, largely inspired in its design by the CUDA version, permits execution on AMD, Intel and NVIDIA hardware, whereas the CUDA port permits execution on NVIDIA GPUs exclusively.

We briefly describe the core functionality of FastCaloSim here; for more details on the C++ codes and CUDA port, the reader is referred respectively to [31] and [20]. The detector geometry includes nearly 190,000 detecting elements, $\mathcal{O}(10)$ MB, each of which can record a fraction of a traversing particle's energy. Various parameterization inputs, $\mathcal{O}(1)$ GB, are used for different particles' energy and shower shapes, derived from Geant4 simulations. The detector geometry, about 20 MB of data, is loaded onto the GPU; due to the large file size of the parameterization inputs, only those data required—based on the particle type and kinematics—are transferred during runtime.

The number of calorimeter *hits*—*i.e.* energy deposited by interacting particles in the sensitive elements—depends largely on the physics process being simulated. For a given physics event, the number of secondary particles produced can range from one to $\mathcal{O}(10^4)$, depending on the incident parent particle type, energy and location in the calorimeter. Three uniformly-distributed pseudorandom numbers are required for each hit to sample from the relevant energy distribution, with the minimum set to 200,000 (approximately one per calorimeter cell).

[2] Use of proprietary data that cannot be made publicly available.

We consider two different simulation scenarios in our performance measurements. The first is an input sample of 10^3 single-electron events, where each electron carries a kinetic energy of 65 GeV and traverses a small angular region of the calorimeters. An average number of hits from this sample is typically 4000–6500, leading to 12000–19500 random numbers per event. Because only a single particle type is used within a limited region of the detector, this scenario requires only several energy and shower shape parameterizations to be loaded onto the GPU during runtime. The second, more realistic, scenario uses an input of 500 top quark pair ($t\bar{t}$) events. In this simulation, the number of calorimeter hits is roughly 600–800 times greater than the single-electron case, requiring $\mathcal{O}(10^7)$ random numbers in total be generated during simulation. Also, a range of secondary particles are produced with various energies that traverse a range of angular regions of the detector. As such, $t\bar{t}$ simulations require data from 20–30 separate parameterizations that need to be loaded to the GPU during runtime, and thus result in a significant increases in time-to-solution on both CPUs and GPUs.

6 Performance Evaluation

6.1 Performance Portability Metrics

There are numerous definitions of performance portability, *e.g.*, [21,27,28,34]. In this paper, we adopt the definition from [29]: the performance portability $\mathcal{P}$ of an application a that solves a problem p correctly on all platforms in a given set H is given by,

$$\mathcal{P}(a,p;H) = \begin{cases} \frac{|H|}{\sum_{i\in H}\frac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall_i \in H \\ 0 & \text{otherwise} \end{cases}, \tag{1}$$

where $e_i(a,p)$ is the *performance efficiency* of a solving p on $i \in H$.
We introduce an *application efficiency* metric, being the ratio between the time-to-solution (TTS) measured using our portable, vendor-agnostic (VA) solution to the native, vendor-specific (VS) performance,

$$\text{VAVS} \equiv \frac{TTS_{\text{portable}}}{TTS_{\text{native}}}. \tag{2}$$

The VAVS metric is useful to identify if runtime overheads are introduced in portability layers which otherwise do not exist in a native API optimized for a specific platform.

6.2 Hardware Specifications

We evaluate performance portability using a variety of AMD, Intel and NVIDIA platforms, ranging from consumer-grade to high-end hardware. This large set

of platforms can be subdivided into CPUs and GPUs, as well as the union of the two, and also helps determine the regime in which the use of GPUs is more efficient for solving a given problem, if one exists.

The Intel x86-based platform tested was a Core i7-10875, consisting of 8 physical CPU cores and 16 threads, a base (maximum) clock frequency of 2.30 (5.10) GHz. To benchmark native oneMKL GPU performance, we use the Intel(R) UHD Graphics 630, an integrated GPU (iGPU) that shares the same silicon die as the host CPU described previously. This iGPU has 24 compute units (CU) and base (maximum) frequency of 350 (1200) MHz. Through Intel's unified memory architecture (UMA), the iGPU has a theoretical maximum memory of 24.98 GB, *i.e.*, the total available RAM on the host. The main advantage of UMA is that it enables zero-copy buffer transfers; no buffer copy between the host and iGPU is required since physical memory is shared between them.

We evaluated SYCL interoperability for AMD and NVIDIA GPUs using an MSI Radeon RX Vega 56 and NVIDIA A100. The Radeon is hosted by an Intel Xeon Gold 5220 36-core processor with a base (maximum) clock of 2.2 (3.9) GHz. An AMD CPU and NVIDIA GPU were evaluated using a DGX A100 node, comprising an AMD Rome 7742 64-core processor with a base (maximum) clock frequency of 2.25 (3.4) GHz. The A100 is NVIDIA's latest high-end GPU, with 6912 CUDA cores and peak FP32 (FP64) of 19.5 (9.7) TF. Note that 16 CPU cores and a single A100 of the DGX were used for these studies.

6.3 Software Specifications

The software used for these studies can be found in Table 1. As our work is relevant only for Linux operating systems (OS), all test machines run some flavor of Linux that supports the underlying hardware and software required for our studies. In this table, DPC++ refers to the Intel LLVM compiler nightly tag from March 3, 2021; separate builds of the compiler were used for targeting $\times 86$ platforms and NVIDIA GPUs. The HIP compiler and hipSYCL are based on Clang 12.0.0, and were installed from pre-compiled binaries available from [4].

Our implementations of SYCL-based cuRAND and hipRAND RNGs within oneMKL were compiled into separate libraries for each platform using the respective compiler for the targeted vendor.

7 Results

The RNG burner application was run 100 iterations for each batch size for statistically meaningful measurements. Each test shown in the following was performed with the Philo $\times 4 \times 32 \times 10$ generator to produce uniformly-distributed FP32 pseudorandom numbers in batches between 1–10^8, as per the requirements of our FastCaloSim benchmark application. Unless otherwise specified, all measurements are of the total execution time, which includes generator construction, memory allocation, host-to-device data transfers, generation and post-processing (*i.e.*, range transformations), synchronisation and finally device-to-

Table 1. Driver and software versions for each platform considered in these studies.

Platform	Driver version	OS and Kernel	Compiler	RNG library
AMD Rome 7742	-	OpenSUSE 15.0 4.12	GNU 8.2.0 DPC++	CLHEP 2.3.4.6 oneMKL
Intel Core i7-1080H	-	Ubuntu 20.04 5.8.18	GNU 8.4.0 DPC++	CLHEP 2.3.4.6 oneMKL
Intel UHD Graphics	21.11.19310	Ubuntu 20.04 5.8.18	DPC++	oneMKL
Radeon RX Vega 56	20.50	CentOS 7 3.10.0	HIP 4.0.0 hipSYCL 0.9.0	hipRAND 4.0.0 oneMKL
NVIDIA A100	450.102.04	OpenSUSE 15.0 4.12	CUDA 10.2.89 DPC++	cuRAND 10.2.89 oneMKL

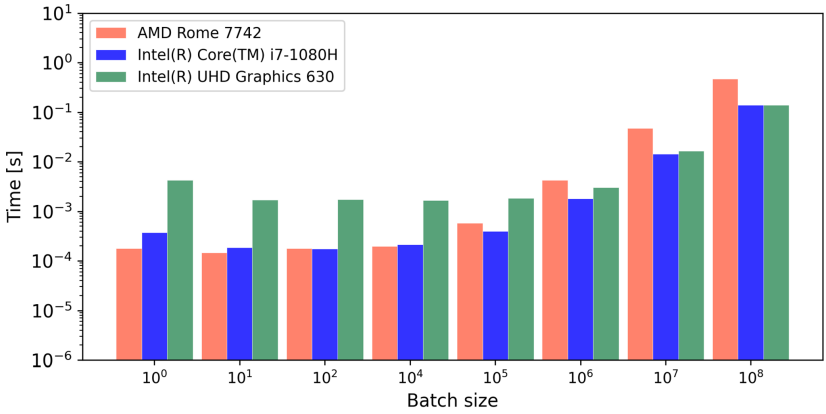
host data transfer times, as determined by the high-resolution `std::chronos` clock.

Shown in Fig. 2 are plots of the total FP32 generation time for the two x86-based CPUs, as well the integrated GPU, using Philox-based generator for both buffer and USM APIs. In general, little overhead is introduced when using the USM API versus buffers. This is a promising result and, to the authors' knowledge, the first benchmark of the different APIs; it is often more productive for developers to port existing codes to SYCL using USM as this approach is often more familiar to C++ programmers who use dynamic memory allocations in their applications.

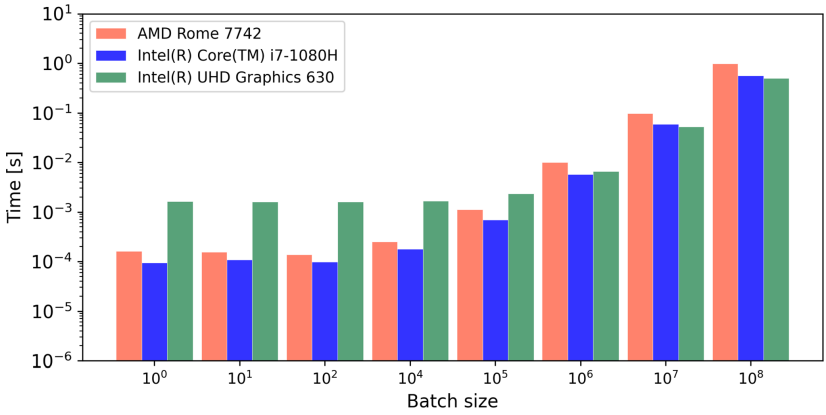
Figure 3 shows separately the RNG burner application results between the buffer and USM APIs, and their native counterparts. Again, we observe statistically equivalent performance using either buffers or USM, with a slight overhead at large batch sizes DPC++ USM and the A100 GPU. More importantly, however, is the level of performance achieved by our cross-platform RNG implementation; TTS for both the cuRAND and hipRAND SYCL backend implementations are on par with their native application.

One immediate point of discussion are the differences in TTS between the Radeon oneMKL-based generator application and native application: the oneMKL version shows slightly better performance for small batch sizes. This is understood as being a result of the optimizations within the hipRAND runtime system for its ROCm back-end. Due to the data dependencies among the three kernels—seeding, generation and post-processing—in the test application, call-backs are issued to signal task completion. These call-backs introduce latencies into the application execution that are significant with respect to small-scale kernels. The nearly callback-free hipRAND runtime system therefore offers higher task throughput. As the batch sizes increase to 10^8, the difference in TTS becomes negligible.

To further investigate this discrepancy, we separate each kernel's duration for both the oneMKL and native cuRAND applications; due to technical and soft-



(a)



(b)

Fig. 2. Results from the RNG burner test application using the buffer API (a) and USM API (b) for Philo $\times 4 \times 32 \times 10$ generation of uniformly-distributed FP32 pseudorandom numbers.

ware limitations, we were unable to profile the Radeon GPU in the following way. Three kernels in total are profiled: generator seeding, generation and our transformation kernel that post-processes the output sequence to the defined range. Figure 4 shows both the time of each kernel executed and relative occupancy in the RNG burner application using data collected from NVIDIA Nsight Compute 2020.2.1. Comparison between each kernel duration is statistically compatible over a series of ten runs. It can therefore conclude that the discrepancies in Fig. 3 between the Radeon oneMKL and native applications can be attributed to differences between the applications themselves, and not fundamentally to the native library kernel executions. Shown also in Fig. 4(b) are the relative occupancy of

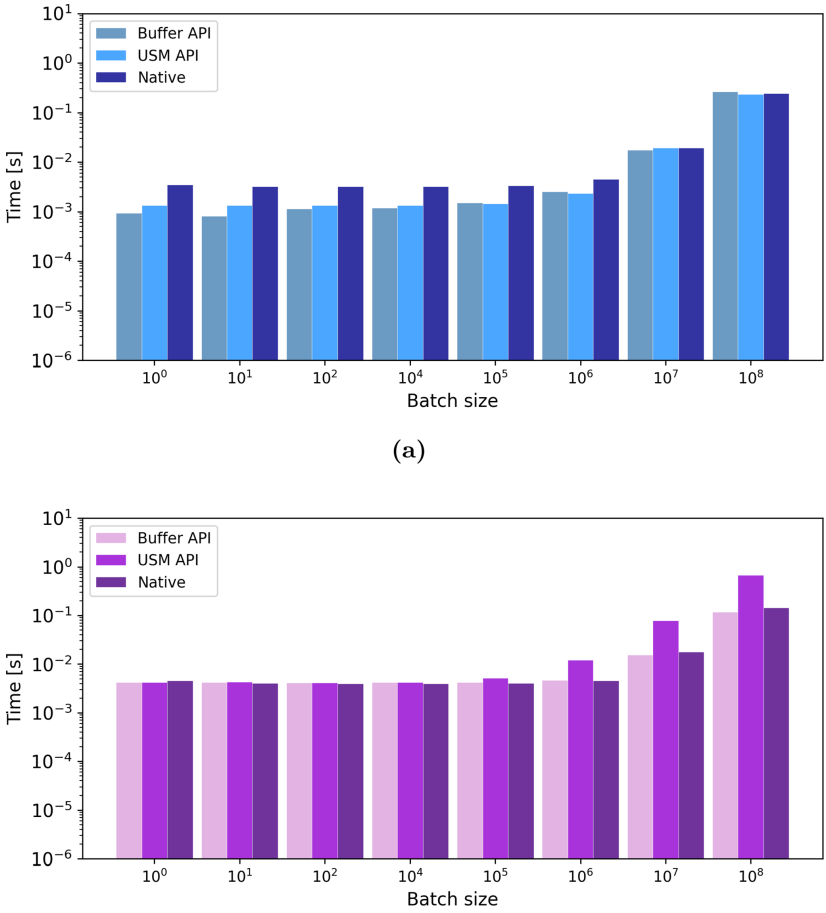
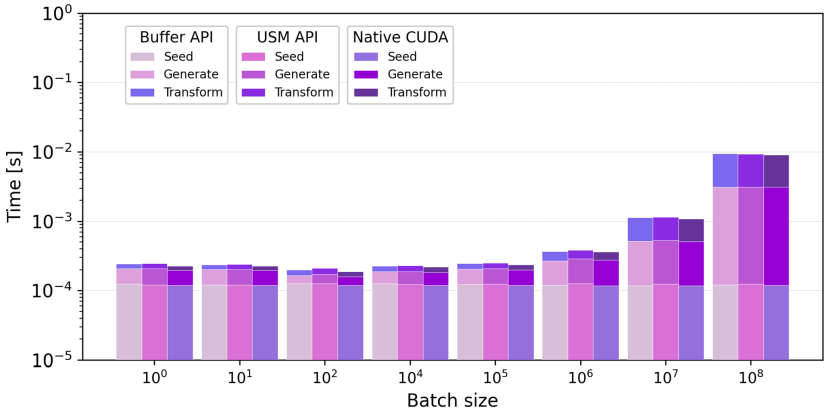
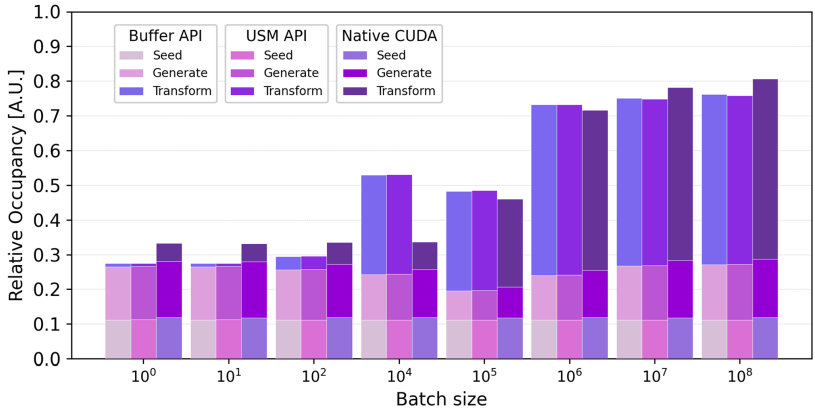


Fig. 3. Comparisons of the RNG burner test application execution time between SYCL buffer and USM APIs, and their native counterparts running on the MSI Radeon RX Vega 56 (a) and NVIDIA A100 (b). The Philo $\times 4 \times 32 \times 10$ generator was used to produce uniformly-distributed FP32 pseudorandom numbers of different batch sizes.

each kernel for the batch sizes generated. Both cuRAND kernels—seeding and generation—are in all cases statistically equivalent between oneMKL and the native application. It can be seen that, despite the nearly identical kernel duration, the buffer and USM API occupancies have a large increase between 10^2 and 10^4 in batch size compared to the native occupancy. This is because when not explicitly specified, the SYCL runtime system optimizes the number of required block size and threads-per-block, whereas in CUDA these values must be determined by the developer as per the hardware specifications. While in the native version the thread-per-block size is fixed to 256, the SYCL kernel runtime chose



(a)



(b)

Fig. 4. Per-kernel total execution time (a) and relative occupancy (b) executed on the NVIDIA A100 with the Philo $\times 4 \times 32 \times 10$ generator producing uniformly-distributed pseudorandom sequences of various batch sizes.

1024 for the NVIDIA A100 GPU. This resulted in the observed differences in kernel occupancy in the native application, as opposed to the SYCL codes for the transform kernel which handle such intricacies at the device level.

Table 2 reports the calculated performance portability of our oneMKL RNG backends using the VAVS metric introduced in Sect. 6. Note that VAVS values closer to unity are representative of greater performance, while smaller values are indicative of poor performance. The data used in calculating the various values of $\mathcal{P}$ are taken from Fig. 4.

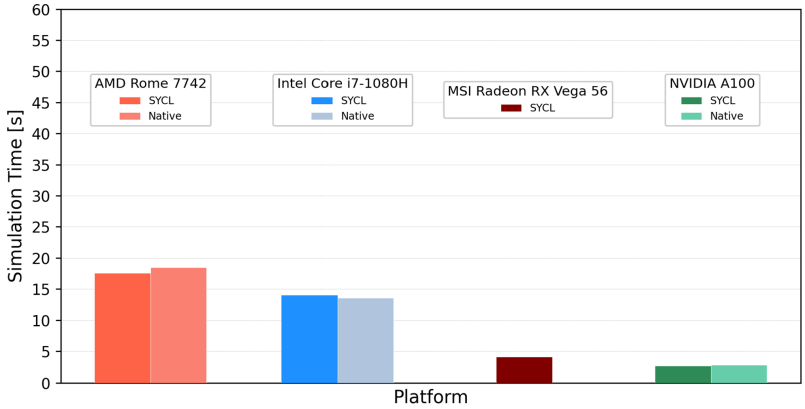
Table 2. Calculated performance portability using the VAVS metric.

H	$\mathcal{P}$ buffer	$\mathcal{P}$ USM	$\mathcal{P}$ Mean (buffer+USM)
{Vega 56, A100}	1.070	0.393	0.575
{Vega 56}	0.974	1.076	1.022
{A100}	1.186	0.240	0.400

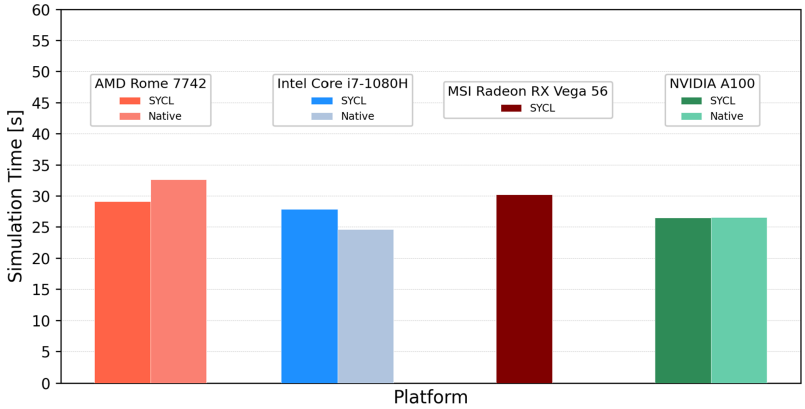
As reported in Table 2, the performance portability measure in a number of cases is greater than unity. This result is consistent with the performance improvement over the native version observed in Fig. 3 for the buffer API on both AMD and NVIDIA GPUs. Although the interoperability kernel time is the same in both native and SYCL versions (see Fig. 4(a)), the buffer API leverages the SYCL runtime system DAG mechanism and hipSYCL optimizations, improving throughput relative to the native application, particularly for small batch sizes. On the other hand, the DPC++ runtime system scheduler does not perform the same with USM as it does when using buffers. Therefore, the performance drop observed in the USM version in Fig. 3 leads to a reduction in the performance portability metric by ∼%40. This behaviour is not observed with hipSYCL.

As a demonstration of cross-platform performance portability in a real-world application, we show in Fig. 5 the average runtime of the FastCaloSim code implementing the proposed SYCL RNG solution across four platforms. Both SYCL and native implementations are shown for each platform, with the exception of the Radeon GPU as no native HIP-based port exists. Ten single-electron and $t\bar{t}$ simulations were run on each platform for reliability of measurements. Where applicable, all measurements made in this study are consistent with those in [20]. The left plot in the figure pertains to the 10,000 single-electron events and the right to the 500 $t\bar{t}$ events (see Sect. 5.2).

In the simpler scenario of single electrons, an approximately 80% reduction in processing time is required on the Vega or A100 GPUs compared to the CPUs considered. However, the overall insufficient use of the full compute capability of the GPUs in this application is made apparent in the more complex topology of $t\bar{t}$ events. This inefficiency is due primarily to the initial strategy in porting FastCaloSim to GPUs; while maximum intra-event parallelism—*i.e.* parallel processing of individual hits within a given event—is met, inter-event parallelism is not implemented in this version of the codes. Future work on the FastCaloSim ports includes event batching to better utilize GPU compute but is beyond the scope of this paper. While the contribution of RNG to the overall runtime of FastCaloSim is small, to investigate SYCL as a portability solution for these codes nevertheless required a SYCL RNG to do so. With cuRAND and hipRAND support added to oneMKL, we can run this prototype application on all major vendors' platforms with no code modifications whatever, and with comparable performance to native codes.



(a)



(b)

Fig. 5. Total runtimes of FastCaloSim across a range of platforms simulating single-electron events (a) and $t\bar{t}$ events (b).

8 Conclusions and Future Work

In this paper, we detailed our implementations of cuRAND and hipRAND backends into oneMKL, and studied their cross-platform performance portability in two SYCL-based applications using major high performance computing hardware, including x86-based CPUs from AMD and Intel, and AMD, NVIDIA and Intel GPUs. We have shown that utilizing SYCL interoperability enables performance portability of highly-optimized platform-dependent libraries across different hardware architectures. The performance evaluation of our RNG codes carried out in this paper demonstrates little overhead when exploiting vendor-optimized native libraries through interoperability methods.

The applicability of the proposed solution has been evaluated in a parameterized calorimeter simulation software, FastCaloSim, a real-world application consisting of thousands of lines of code and containing custom kernels in different languages and vendor-dependent libraries. The interfaces provided by oneMKL enabled the seamless integration of SYCL RNGs into FastCaloSim with no code modification across the evaluated platforms. The SYCL 2020 interoperability functionality enabled custom kernels and vendor-dependent library integration to be abstracted out from the application, improving the maintainability of the application and reducing the source lines of code. The application yields comparable performance with the native approach on different architectures. Whereas the ISO C++ version of FastCaloSim had two separate codebases for x86 architectures and NVIDIA GPUs, the work presented here has enabled event processing on a variety of major vendor hardware from a single SYCL entry point. Hence, the SYCL RNG based integration facilitates the code maintainability by reducing the FastCaloSim code size without introducing any significant performance overhead.

While we have demonstrated that SYCL interoperability leads to reusability of existing optimized vendor-dependent libraries and enables cross-platform portability, devices without vendor libraries cannot be supported. For example, no RNG kernels exist yet for ARM Mali devices. One possible solution would be to provide pure SYCL kernel implementations for common RNG engines. The kernel could then be compiled for any device for which a SYCL-supported compiler exists. Moreover, in scientific applications and workflows where reproducibility is essential, kernels written entirely in the SYCL programming model can offer improved reliability across architectures and platforms. Although the portability of such an RNG kernel would be guaranteed, performance remains challenging and likely would necessitate mechanisms such as tuning of kernels for different architectures.

Finally, extending performance portability to include also productivity and reproducibility in an objective way would general scientific applications and workflows aiming for architecture and platform independence.

Acknowledgement. This work was completed while V.R.P. was at Lawrence Berkeley National Laboratory, and was funded in part by the DOE HEP Center for Computational Excellence at Lawrence Berkeley National Laboratory under B&R KA2401045.

Data Availability Statement

Summary of the Experiments Reported

We ran two benchmark applications on a variety of hardware:

1. Intel Core i7-1080H, Intel UHD Graphics 630 (Razer Blade Studio Edition 2020) 2. AMD Rome 7742, NVIDIA A100 (DGX node from NERSC) 3. MSI Radeon RX Vega 56 (Private Intel Xeon Gold 5220 node).

Both applications are freely available (Github link below) but inputs to FastCaloSim are proprietary data of the ATLAS Experiment that we unfortunately cannot shared publicly (special access may be granted upon request).

We used Intel LLVM sycl-nightly/20210330, nvcc 10.2 and hipSYCL 0.9.0 for the various targets. oneMKL is used for all RNG but the hipRAND backend is not publicly available due to DOE restrictions on software developed by employees. We are happy to make arrangements for this to be made available.

Artifact Availability

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: All author-created hardware artifacts are maintained in a public repository under an OSI-approved license.

Data Artifact Availability: Some author-created data artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Proprietary Artifacts: There are associated proprietary artifacts that are not created by the authors. Some author-created artifacts are proprietary.

List of URLs and/or DOIs where artifacts are available:

`https://github.com/oneapi-src/oneMKL`
`https://github.com/vrpascuzzi/FastCaloSim-GPU/tree/benchmarking`
`https://github.com/vrpascuzzi/benchprof/tree/sc21`

Baseline Experimental Setup, and Modifications Made for the Paper

Relevant hardware details: DGX A100, Intel Core i7-1080H, Intel UHD Graphics 630, MSI Radeon RX Vega 56, NVIDIA A100, Intel Xeon Gold 5220

Operating systems and versions: Ubuntu 20.04 with kernel 5.8.18, OpenSUSE 15.0 with kernel 4.12, CentOS7 with kernel 3.10

Compilers and versions: GNU 8.2, nvcc 10.2, hipSCYL 0.9.0, Clang 12.0.0

Libraries and versions: oneMKL v0.1.0, CUDA 10.2.89, hip 4.0

Key algorithms: Philo $\times 4 \times 32 \times 10$, MRG32k3a

Input datasets and versions: ATLAS FastCaloSim single-electron and top-antitop quark n-tuple inputs

Paper Modifications: We added to the oneMKL open-source interfaces library random number generator (RNG) support for AMD (hipRAND) and NVIDIA (cuRAND) GPUS through SYCL interoperability. This provides a single entry point for executing on a wide range of available HPC systems scientific and other codes which utilize RNGs.

Output from scripts that gathers execution environment information

```
A number of systems were used for these studies. As these studies
↪ were performed several months ago, and due to access
↪ privileges and updates, the hardware and software
↪ specifications are no longer valid. For example, the DGX node
↪ was a NERSC Perlmutter early access machine offered to
↪ Pascuzzi, and is no longer online.

For the most accurate details, please see "Baseline experimental
↪ setup, and modifications made for the paper" section above.
```

Artifact Evaluation

Verification and validation studies: Each experiment was run hundreds of times over the course of several weeks to validate day-to-day and operational fluctuations of the systems used for benchmarking.

Accuracy and precision of timings: Each experiment was run hundreds of times over the course of several weeks to validate day-to-day and operational fluctuations of the systems used for benchmarking.

Used manufactured solutions or spectral properties: N/A

Quantified the sensitivity of results to initial conditions and/or parameters of the computational environment: Each experiment was run hundreds of times over the course of several weeks to validate day-to-day and operational fluctuations of the systems used for benchmarking.

References

1. AMD hipBLAS: Dense Linear Algebra on AMD GPUs. https://github.com/ROCmSoftwarePlatform/hipBLAS. Accessed 05 Apr 2021
2. AMD hipRAND: Random Number Generation on AMD GPUs. https://github.com/ROCmSoftwarePlatform/rocRAND. Accessed 05 Apr 2021
3. ComputeCpp: Codeplay's implementation of the SYCL open standard. https://developer.codeplay.com/products/computecpp/ce/home. Accessed 28 Feb 2021
4. hipSYCL RPMs. http://repo.urz.uni-heidelberg.de/sycl/test-plugin/rpm/centos7/. Accessed 13 Mar 2021
5. Intel Math Kernel Library. https://intel.ly/32eX1eu. Accessed 31 Aug 2020
6. Intel oneAPI DPC++/C++ Compiler. https://github.com/intel/llvm/tree/sycl. Accessed 28 Feb 2021
7. Intel oneAPI Math Kernel Library (oneMKL). https://docs.oneapi.com/versions/latest/onemkl/index.html. Accessed 28 Feb 2021
8. NVIDIA cuBLAS: Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas. Accessed 31 Aug 2020

9. NVIDIA CUDA programming model. http://www.nvidia.com/CUDA. Accessed 05 Apr 2021
10. NVIDIA cuRAND: Random Number Generation on NVIDIA GPUs. https://developer.nvidia.com/curand. Accessed 28 Feb 2021
11. NVIDIA cuSPARSE: the CUDA sparse matrix library. https://docs.nvidia.com/cuda/cusparse/index.html. Accessed 05 Apr 2021
12. SYCL: C++ Single-source Heterogeneous Programming for OpenCL. https://www.khronos.org/registry/SYCL/specs/sycl-2020-provisional.pdf. Accessed 23 July 2020
13. The ARM Computer Vision and Machine Learning library. https://github.com/ARM-software/ComputeLibrary/. Accessed 31 Aug 2020
14. Aad, G., et al.: The ATLAS Experiment at the CERN Large Hadron Collider, vol. 3, p. S08003, 437 (2008). https://doi.org/10.1088/1748-0221/3/08/S08003, https://cds.cern.ch/record/1129811, also published by CERN Geneva in 2010
15. Agostinelli, S., et al.: GEANT4-a simulation toolkit, vol. 506, pp. 250–303 (2003). https://doi.org/10.1016/S0168-9002(03)01368-8
16. Alpay, A., Heuveline, V.: SYCL beyond OpenCL: the architecture, current state and future direction of hipSYCL. In: Proceedings of the International Workshop on OpenCL, p. 1 (2020)
17. Buckley, A., et al.: General-purpose event generators for LHC physics. Phys. Rep. **504**(5), 145–233 (2011)
18. Costanzo, M., Rucci, E., Sanchez, C.G., Naiouf, M.: Early Experiences Migrating CUDA codes to oneAPI (2021)
19. Deakin, T., McIntosh-Smith, S.: Evaluating the performance of HPC-Style SYCL applications. In: Proceedings of the International Workshop on OpenCL, IWOCL 2020. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3388333.3388643
20. Dong, Z., Gray, H., Leggett, C., Lin, M., Pascuzzi, V.R., Yu, K.: Porting HEP parameterized calorimeter simulation code to GPUs. Front. Big Data **4**, 32 (2021)
21. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**, 3202–3216 (2014)
22. Feickert, M., Nachman, B.: A Living Review of Machine Learning for Particle Physics (2021)
23. Goli, M., et al.: Towards cross-platform performance portability of DNN models using SYCL. In: 2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 25–35. IEEE (2020)
24. Gozillon, A., Keryell, R., Yu, L.Y., Harnisch, G., Keir, P.: triSYCL for Xilinx FPGA. In: The 2020 International Conference on High Performance Computing and Simulation. IEEE (2020)
25. Hornung, R.D., Keasler, J.A.: The RAJA portability layer: overview and status. Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States) (2014)
26. James, F., Moneta, L.: Review of high-quality random number generators. Comput. Softw. Big Comput. **4**, 1–12 (2020). https://doi.org/10.1007/s41781-019-0034-3
27. Larkin, J.: Performance portability through descriptive parallelism. In: Presentation at DOE Centers of Excellence Performance Portability Meeting (2016)
28. McIntosh-Smith, S., Boulton, M., Curran, D., Price, J.: On the performance portability of structured grid codes on many-core computer architectures. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2014. LNCS, vol. 8488, pp. 53–75. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07518-1_4

29. Pennycook, S.J., Sewall, J.D., Lee, V.W.: Implications of a metric for performance portability. Future Gener. Comput. Syst. **92**, 947–958 (2019)
30. Pheatt, C.: Intel threading building blocks. J. Comput. Sci. Coll. **23**(4), 298 (2008)
31. Schaarschmidt, J.: The new ATLAS fast calorimeter simulation. J. Phys. Conf. Ser. **898**, 042006 (2017). https://doi.org/10.1088/1742-6596/898/4/042006
32. Stauber, T., Sommerlad, P.: ReSYCLator: transforming CUDA C++ source code into SYCL. In: Proceedings of the International Workshop on OpenCL, IWOCL 2019. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3318170.3318190
33. Stone, J.E., Gohara, D., Shi, G.: OpenCL: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(3), 66 (2010)
34. Zhu, W., Niu, Y., Gao, G.R.: Performance portability on EARTH: a case study across several parallel architectures. Cluster Comput. **10**(2), 115–126 (2007)