Porting ATLAS Fast Calorimeter Simulation to GPUs with Performance Portable Programming Models

Mohammad Atif[2], *Zhihua* Dong[2], *Charles* Leggett[1], *Meifeng* Lin[2], and *Vakhtang* Tsulaia[1]

[1]Lawrence Berkeley National Laboratory, Berkeley CA 94720 USA
[2]Brookhaven National Laboratory, Upton NY USA

Abstract. FastCaloSim is a parameterized simulation of the particle energy response and of the energy distribution in the ATLAS calorimeter. It is a relatively small and self-contained package with massive inherent parallelism and captures the essence of GPU offloading via important operations like data transfer, memory initialization, floating point operations, and reduction. It was identified by the High Energy Physics Center for Computational Excellence project as a good testbed for evaluating the performance and ease of portability of programming models.
In this paper, we will discuss the results of our evaluation of the porting process to Kokkos, SYCL, Alpaka, OpenMP and std::par (nvc++), and compare performance on NVIDIA, AMD and Intel GPUs, as well as multicore CPUs.

1 Introduction

Simulation is an essential aspect of all High Energy Physics (HEP) experiments and the ATLAS experiment at the Large Hadron Collider is no exception, requiring massive numbers of simulated events for detector modeling. The traditional method of propagating particles through the detector to model interactions uses the Geant4 toolkit [1], which operates with small, step-wise increments, and is extremely compute-intensive, especially for the complex detector geometry of the ATLAS Liquid Argon Calorimeter. Using this technique for all Monte-Carlo simulation of the detector would either consume too large a fraction of the ATLAS computing budget or limit the physics precision. In order to address this, ATLAS developed the FastCaloSim package [2] [3] that uses a simplified detector geometry and parametrizations of shower development initiated by particles traversing the calorimeter volumes. Depending on the processes being modeled, this parametrized simulation can reduce CPU usage by a factor of 10 to 25 compared to a full Geant4 simulation.

We are currently witnessing a rapid expansion of available architectures for computational accelerators such as GPUs. Until very recently, NVIDIA was the dominant manufacturer of GPUs, and there was no reason not to use CUDA [4] to write code for them. In the past few years, both AMD and Intel have significantly increased their penetration into the computational GPU market, and we have seen multiple new facilities being built around them, such as Frontier (AMD) at Oak Ridge National Laboratory and Aurora (Intel) at Argonne National Laboratory in the US. Some HEP experiments have already committed to using non-NVIDIA GPUs, such as ALICE which is using AMD GPUs for its online event farm. However, most HEP experiments lack the resources to rewrite their algorithms for each individual architecture, so a portability solution that can execute on both CPUs and all current GPUs is essential.

There are a number of portability layers currently available that can target CPUs and GPUs from the same source code, such as Kokkos, alpaka, SYCL, OpenMP, and std::par. In order to understand their strengths and weaknesses, it is important to evaluate each one using real-world scenarios from HEP.

The original CPU-based FastCaloSim code has been extracted from the ATLAS repository into a standalone package, and ported to run on GPUs [5] using CUDA, where runtime speedups of as much as 60 times that of a single-threaded CPU process have been demonstrated. This code offers an excellent opportunity to test other GPU programming mechanisms, as not only is it self-contained with few external dependencies, but also it represents a real High Energy Physics workflow with valuable physics potential, making it far superior to an artificial testcase or synthetic benchmark.

The High Energy Physics Center for Computational Excellence (HEP-CCE) [16] has been evaluating portable parallelization strategies by selecting a number of representative GPU workflows from HEP and porting them to various GPU portability layers. FastCaloSim has been chosen as a testbed for this purpose.

2 FastCaloSim

FastCaloSim models the interaction of particles in the ATLAS Liquid Argon Calorimeter by using a simplified detector geometry and parameterizations of shower development initiated by particles traversing the calorimeter volumes. During a simulation, the optimal parameterization is selected based on the particle type, energy deposit and location in the detector in order to best model the corresponding electromagnetic or hadronic shower. The parameterizations are then used to create calorimeter *hits*, which contain information about the location, and amount of energy deposited in the detector.

The detector geometry is composed of 187,652 elements arranged in 24 layers. In the original CPU code, the detector description is encoded in a `std::vector` of pointers to elements, which is not ideal for usage on GPUs. These data structures have been flattened and coalesced into simple arrays for better performance on the GPU. Similarly, the parametrization tables, which are also held in an `std::vector`, were transformed into a simple array before being copied to the GPU. The size of the parametrization tables can vary significantly depending on the type of particles being modeled, from a megabyte for 65 GeV electrons, to over a gigabyte for high energy $t\bar{t}$ simulations.

The simulation requires a large number of random numbers. Each hit requires 3, and events can have up to 20,000 hits. The input data samples are between 500 and 10000 events, so for some runs we require on the order of 600 million random numbers. These numbers are generated in batches on the GPU to avoid repeated calls to the generator and regenerated when the batches are consumed. However, in order to simplify data validation, the random numbers can also be generated on the CPU and copied to the GPU.

There are three main kernels that perform the simulation on the GPU. The first resets a large workspace, an array of approximately 200k floats, which corresponds to the detector elements. The second performs the actual parameterized simulation, calculating into which detector cells the energy of the hit is deposited, and is mostly composed of floating point operations. Atomic operations are used to accumulate the energy deposits. A block reduction method was evaluated in order to avoid the use of atomics but was found to be less performant. It is parallelized over the number of hits in the event, with each hit being assigned to a different thread. If the hit count per event is below a certain minimum, then the kernel is not offloaded, but instead computed on the CPU, as the kernel launch penalty becomes larger than the CPU computation time. This minimum is on the order of 500 hits. The last kernel performs a stream compaction step, counting the hit cells and gathering the result for more efficient data transfer back to the host. Since the algorithmic processing is sequential in nature, the kernels are launched synchronously. After the third kernel executes, an array of structs containing the cell identifiers and their energy deposits are transferred back to the host.

For some particle types and energy combinations, the number of hits in the detector is small, leading to under-utilization of GPU resources. To increase the load on the GPU, we grouped hits between multiple events, providing sufficient extra information to the kernels to disambiguate between the grouped events. Though this method slightly increases the sizes of the data structures on the GPU, it allows for much longer running kernels, overshadowing the launch latency penalties and using all the computational cores on the GPU.

3 Portability Layers

3.1 Kokkos

Kokkos [6, 7] is a portable, performant, C++ based shared-memory programming model that is single source, i.e. it lets you write algorithms once and run on any supported backend architectures, such as a traditional CPU, NVIDIA, AMD, and Intel GPUs, and manycore CPUs, minimizing the amount of architecture-specific implementation details that users need to know. It provides a number of different parallel abstractions available, such as parallel_for, reductions, and scans, and also provides utilities such as random number generators, and support for atomic operations, chained kernels, and callbacks. The library, which is mostly header-based, is compiled for a selected set of backends - one serial, one host parallel, and one specific accelerator device can be chosen in a single binary. These backends must be selected at compile time. Though it provides constructs for allocating and managing data on the host and accelerator devices, these can be wrapped around pre-existing data objects. Execution kernels can also use bare pointers to operate on data allocated and transferred by other means.

3.2 SYCL

SYCL [8] is a cross-platform abstraction layer intended for heterogeneous computing, based on OpenCL, and originally released by the Khronos group in 2014. Since then, there have been a number of implementations by different groups. Like Kokkos, it is also intended to support single source portable C++ programming. It does not mandate explicit memory transfers, but rather builds a directed acyclic graph (DAG) of kernel data dependencies, and transfers the data between host and offload device as needed. SYCL runs on a broad range of architectures, and in theory, permits the selection of the execution devices at runtime. In practice, different accelerator backends require different compilers, such as openSYCL, to target AMD GPUs, and different builds of llvm/dpc++ to target NVIDIA or Intel GPUs.

3.3 OpenMP

OpenMP [9], originally a shared-memory programming model through compiler directives, has extended support for parallel execution on both host and device architectures via its `target offload` model. Several compilers currently support OpenMP's `target offload`, among which LLVM Clang and GCC are community-developed, while NVIDIA's nvc++, AMD's amdclang, AOMP, AFAR, and Intel's icpx are vendor-developed. As the compilers are undergoing rapid development, appropriate flags that vary with compilers and architecture are currently required for device execution. Among the compilers, LLVM Clang stands out due to the support from the developers and because most vendor based compilers are based on it. It offers features such as optimization remarks and debug environments. The optimization remarks, invoked with flags `-Rpass=openmp-opt, -Rpass-analysis=openmp-opt`, and `-Rpass-missed=openmp-opt` provide details about the location or movement of data and insights into performance improvement or degradation.

3.4 alpaka

Alpaka[12–14] is another single source portability layer implemented as a header-only C++ library. Alpaka is platform-independent and it allows for concurrent use of multiple devices, such as the CPU host and the attached accelerators. User kernels are represented by function objects with a special interface. There is no need to write specialized code in various GPU programming languages, only one implementation of the kernel works across different devices. The alpaka library provides backends for several GPU programming technologies (e.g., CUDA, HIP, SYCL, OpenMP) and for several CPU threading libraries (e.g., C++ Threads, Intel TBB). Its C++ template interface allows for implementing user-defined extensions of the list of supported accelerators and libraries. The abstraction used by alpaka is similar to the strategy used by CUDA, which builds a parallel execution hierarchy from elements, threads, blocks, and the grid. In the alpaka library, the memory allocation function is uniform for all devices, including the host. It returns reference counted memory buffer objects that take care of proper memory handling for specific backends. Finally, for porting the existing CUDA code alpaka offers its extension called cupla [15], where only the includes and syntax of the kernel calls need to be changed, while the kernel body remains intact.

3.5 std::par

`std::execution::parallel` (`std::par`), is an existing C++ standard introduced in C++17 to enable parallel processing of algorithms on CPUs by defining execution policies. It offers execution policies such as serial, parallel execution using threads, and parallel execution using threads and vectorization. NVIDIA introduced a new compiler (`nvc++`) in 2020 which enabled the execution of these policies on NVIDIA GPUs. In order to transparently move data between host and device, `nvc++` uses unified shared memory, with data being migrated on demand via page faults. It is not intended to be a replacement for a low-level language such as CUDA, as it lacks many GPU-specific features and optimizations, but rather as a stepping stone or bridge between CPU-based serial code, and that explicitly written for GPUs, dramatically lowering the entry bar for execution on accelerators. Intel also has a compiler (`oneAPI:dpl`) that mostly supports this standard.

4 Porting Experience

A CUDA version of FastCaloSim had been produced by a previous project and was used as a starting point for the ports to the various portability technologies. Since it had already been heavily tested, we used the CUDA version as both a performance baseline, and a source of validated "truth".

In order to maximize code reuse, the FastCaloSim git repository was structured so that the same branch would be used for all the portability technologies. We kept the same CMake infrastructure as the original code, and added configuration time switches, such as `-DUSE_KOKKOS=ON` or `-DUSE_ALPAKA=ON`, to select the desired technology at build time. Technology specific files were identified by name, such as `File.cxx` would be used by all technologies, `File_cu.cu` would be used for CUDA, `File_kk.cxx` for Kokkos, `File_al.cxx` for alpaka, etc.

One low-level GPU feature that is used by FastCaloSim is incrementing integer and float counters using `atomics`. We tried to use the portability layer specific implementation for each portability layer, instead of using a device-specific implementation, in order to maximize portability.

4.1 Kokkos

Kokkos permits the intermingling of native and Kokkos kernels in the same application or even file, for example if the CUDA backend of Kokkos is used, then code written in CUDA can be used in the same file as a Kokkos kernel. This permits the piecemeal migration of kernel, greatly simplifying the process.

FastCaloSim uses shared libraries, so Kokkos needs to be compiled with this feature enabled. However, shared libraries are not compatible with device symbol relocation, meaning that all symbols within one kernel have to be visible by a single compilation unit. Due to the file and class structure of FastCaloSim, which we attempted to preserve, this required using a wrapper file that used `#include` to load individual files into one. In order to simplify this process, and to be able to reuse as much common code in the CUDA and Kokkos implementations, a considerable amount of code refactoring was necessary. One issue is that the main Kokkos include headers cannot be exposed to `nvcc`, the CUDA compiler, so careful file separation and judicious use of macros to select compilation paths are necessary to maintain the CUDA/Kokkos separation while maximizing the amount of shared code. However, the effort put into this refactoring paid off later by considerably simplifying the ports to other technologies.

While `Kokkos::View<>` can wrap data objects that have been previously allocated on host or device, in order to fully explore the Kokkos API and maximize portability, we rewrote all data allocations to use `Kokkos::Views` natively. We did, however, simplify the porting process by first wrapping pre-existing structures before fully converting them to native Views. FastCaloSim uses a number of different data structures that need to be transferred to the device, from simple structures to more complicated arrays of classes and jagged multi-dimensional arrays. While the former are easy to convert to `Kokkos::View`, the latter are not well supported by Kokkos, which strongly recommends against using `Views` of `Views`. We chose to convert these jagged arrays to either regular 2-dimensional arrays by padding them, or flattened them into 1-dimensional arrays. This required a significant amount of extra boilerplate code as compared with the CUDA implementation.

The Kokkos runtime needs to be explicitly initialized and finalized. This requires inserting appropriate code into the `main()` routine, and since this routine is shared between portability technologies, it is necessary to protect it with macros so that it only gets used during Kokkos builds. Kokkos uses lambda syntax to define its kernels, but while the syntax is different than the triple chevrons used by CUDA, the functionality is largely the same, and the conversion between the two is trivial. `Kokkos::atomic_fetch_add` was used to increment the atomic counters.

Kokkos code could be somewhat challenging to debug and profile, as at runtime, the kernel names are heavily templated and very long. Profilers such as nsys will often truncate these long names, rendering the kernels indistinguishable.

4.2 SYCL

Intel provides a CUDA → SYCL translation program, originally called the DPC++ Compatibility Tool, now an open source project called SYCLomatic, which we tried using to generate the first pass of the SYCL port of FastCaloSim. While this proved instructive in generating useful boilerplates for kernel dispatching, queue generation, and data buffer creation, it did not in fact generate runnable code. So, unlike the other ports, instead of trying to merely translate the CUDA code to the SYCL syntax, we instead re-wrote the code following SYCL best coding practices.

We kept the structure of jagged arrays, and used a mixture of explicit data transfers and Unified Shared Memory for automatic data migration. The explicit transfers were used for the more static data, such as the detector geometry and parametrization tables, while the USM was used for the event varying data that the kernels would work on.

At the time of code development, there was no uniform interface from SYCL to GPU based random number generators. We developed separate implementations for Intel, AMD and NVIDIA devices, and then ultimately made an addition to the Intel OneMKL random number generator that offers both buffer and USM based data access and uses cuRAND and rocRAND for native hardware support on NVIDIA and AMD GPUs.

Intel offers very powerful tools for profiling and debugging SYCL on Intel hardware. We were very fortunate to have access to development prototypes for the Aurora supercomputer during our studies which enabled us to run these profiling tools. These tools are now available to run on commodity Intel GPUs.

The SYCL standard and language implementations were seeing tremendous development while we were working on FastCaloSim. This was both a blessing, as many bugs that we discovered were resolved, and new features, such as support for atomics were added, as well as being a curse, since features or syntax that worked in one release would not function in another, and careful selection of specific builds of the compilers, of which they were many, were required. We experimented with a number of different compilers, including Intel's oneAPI/dpc++ (for Intel GPUs), hipSYCL/openSYCL (for AMD GPUs), Codeplay (for NVIDIA GPUs), and llvm (for Intel, NVIDIA and AMD GPUs). Some of these no longer exist, or have been subsumed by others. SYCL compilers have since stabilized, resulting in a much smoother developer experience.

4.3 OpenMP

The FastCaloSim port of OpenMP's `target offload` was relatively easy to implement. OpenMP allows simultaneous device and host parallelization and is interoperable with other programming models. This makes an incremental porting of a serial C++ code or an existing CUDA code easy. The major modifications when porting from an existing CUDA implementation were inserting appropriate memory allocation APIs and replacing CUDA kernel calls with compiler directives. The data structures containing the geometry and random numbers were allocated on the GPU using `omp_target_alloc` which requires the clause `is_device_ptr` to indicate to the compiler that a pointer has been allocated memory on the target device. The arrays were moved between host and device using `omp_target_memcpy` whenever needed.

The `declare mapper` directive along with `targer enter/exit data map` was preferred over individually mapping members of structure with `data map` as the latter is slower. LLVM Clang has been introducing compiler flags such as `-fopenmp-cuda-mode`, `-fopenmp-assume-no-thread-state`, `-fopenmp-assume-no-nested-parallelism` for performance optimization, and `-foffload-lto` for link-time optimization. The compiler flag `-fopenmp-cuda-mode` drastically improved performance on NVIDIA GPUs for sequential simulations, but did not have a significant impact on AMD GPUs or group simulations. The rest of the compiler flags had marginal impact on the performance. Further performance optimization required fine-tuning the number of threads per team and carefully managing static variables with `map` and `target` region clauses [10, 11].

Performance profiling on NVIDIA hardware can be achieved with Nsight Systems' `profile` tool with command-line argument `--stats=true --trace=openmp,cuda`. This generates an `nvprof`-style profiling data about data movement and offloaded kernels. Similarly, `rocprof` from the ROCm stack with option `--stats` generates profiling data of the offloaded kernels on AMD hardware. The environment variable `LIBOMPTARGET_INFO=-1` also provides detailed runtime information about the performance of the various offloaded kernels. Debugging is at times challenging as tools like `compute_sanitizer` have limited support from the vendors for OpenMP offloading. The demangling tools do not seem to work for symbols beginning with `__omp_offloading_`. However, LLVM Clang offers debugging options via the flags `-fopenmp-target-debug=<N>` or the environment variable `LIBOMPTARGET_DEVICE_RTL_DEBUG=<N>` for compile time and runtime debugging respectively.

4.4 alpaka

The process of porting FastCaloSim code to alpaka was rather straightforward. The learning curve was rather steep due to the challenging nature of the alpaka coding style and principles. Also, the first piece of code in FastCaloSim which we decided to port from CUDA to alpaka was the code generating a set of random numbers on the accelerator. At the time of the code development, the documentation/guidelines for developing random number generation code in alpaka were missing, and the number of code snippets showing the usage of random number generators within alpaka examples was rather limited (the situation in this area has since been greatly improved). As a result, it took us longer than anticipated to complete the first porting task. Nonetheless, becoming familiar with developing random number generation code in alpaka was a valuable experience in the overall learning process.

Similarly to Kokkos, it is possible to mix alpaka code with native GPU application code (e.g., written in CUDA when targeting NVIDIA hardware) in the same application. This feature turned out to be very helpful in the process of step-by-step migration of the existing CUDA implementation of FastCaloSim to alpaka. This made it also possible to check the correctness of the migrated code by using the original CUDA implementation as a reference and validating alpaka implementations of kernels one at a time. In general, we found the process of porting CUDA kernels to alpaka relatively simple, as in our application the kernel bodies remained practically unchanged.

In general, given that alpaka offers a heavily templated, low-level API, the application code written in alpaka tends to be rather verbose. Hence, for applications with a large code base, it may be beneficial to implement a shallow layer of template functions on top of the alpaka API in order to hide this complexity from the application code.

alpaka offers two mechanisms for performing memory operations (e.g., allocation, and copying). One can either explicitly allocate memory using reference-counted alpaka buffers or use already pre-allocated chunks of memory via alpaka memory views. In the alpaka implementation of FastCaloSim we used only the former mechanism, which is also an approach recommended by the alpaka developers as the usage of buffers removes the need for manual freeing and the possibility of memory leaks. As mentioned above, FastCaloSim uses different data structures, including some jagged arrays. There is no explicit support for jagged arrays in alpaka, so we took the approach of flattening them into 1-dimensional arrays. This required the addition of some boilerplate code, although the amount of such extra code was not dramatic.

4.5 std::par

We encountered a major hurdle in porting FastCaloSim to `std::par` - after compiling all of FastCaloSim with `nvc++`, we suffered from runtime errors when using a ROOT library compiled with `g++`. We were unable to perform a purely native build of ROOT using `nvc++`, and were thus forced to compile part of the project which used ROOT with `g++` and the parts that were offloaded to the GPU with `nvc++`. Since CMake only permits one compiler per file extension, this required creating a compiler wrapper that would select the appropriate compiler and compile flags based on the directory in which the file was located.

`nvc++` migrates data between host and device automatically, triggered by page faults on memory accesses. At compile time, it instruments object allocations to aid this process. However, if data is allocated using `g++`, this instrumentation is absent, and so a runtime error will occur if the data is accessed on the device. It needs to be copied to another structure allocated by `nvc++` before it can be utilized on the device.

`std::par` was designed to be used on STL algorithms and containers, with access via a forward iterator. It lacks the ability to access a specific index. When multiple containers or data structures need to be accessed within a single kernel, having an index is essential. When utilizing the C++20 specification, this functionality can be accomplished using an `mdspan`, but C++20 was not compatible with ROOT at the time of our studies. For C++17, which is the latest version of the standard supported by FastCaloSim, we needed to use a helper class called the `CountingIterator` to replicate index based access to arrays.

FastCaloSim makes use of atomic operations on floats and integers, however `std::atomic<float>` is only supported by C++20 and later. To work around this limitation, we tried two techniques - converting the float to an appropriately scaled integer and using a `std::atomic<int>`, and by using the CUDA implementation `atomicAdd(float)`. While the latter is not a portable solution, `nvc++` currently only works for NVIDIA GPUs, and this was used as a comparison and placeholder until we could implement a C++20 compatible solution.

5 Performance Results

When benchmarking the performance of the Kokkos port of FastCaloSim, the overheads from the Kokkos layers over the data structures and the kernel launches becomes immediately evident. As seen in Figure 1, where the timing of the individual kernels, data transfers and overall event loop is shown relative to a native CUDA, HIP or SYCL implementation, Kokkos imposes a non-negligible penalty on the creation of objects in Views, the dispatching of kernels, and the data transfers from device to host. This is exacerbated by the fact that the kernel runtimes are between 10 and 100 μs on an NVIDIA A100, and thus heavily impacted by the unavoidable kernel launch latency, which is on the order of 5 to 10 μs. The apparently good performance of the HIP backend to Kokkos for the simulation kernel is in fact due to the very poor performance of FastCaloSim on AMD hardware, with runtimes being between 40× and 4000× slower than on NVIDIA GPUS, depending on the energy of the simulated particles, which overshadows the overheads from Kokkos. The SYCL backend on AMD hardware has significantly worse performance for the main simulation kernel than the other technologies, however the code is still very immature and lacks the years of optimization that the Kokkos developers have had for the NVIDIA and AMD backends.

We also see performance penalties when using the host parallel backends of Kokkos to run on multiple CPU cores using OpenMP or pThreads. As shown in Figure 2, the single core serial backend of Kokkos runs at about 75% of the speed of the original CPU implementation

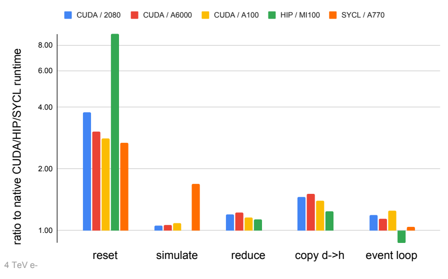

Figure 1. Performance of FastCaloSim using Kokkos relative to native the CUDA, HIP or SYCL on various GPUs, for events simulated one by one.

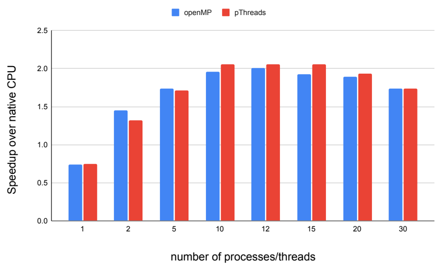

Figure 2. Performance of the event loop host parallel execution using the OpenMP and pThreads Kokkos backend.

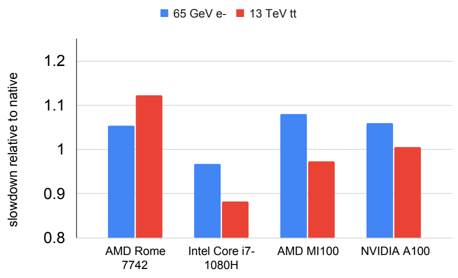

Figure 3. Performance of the main simulation kernel with SYCL relative to native implementation.

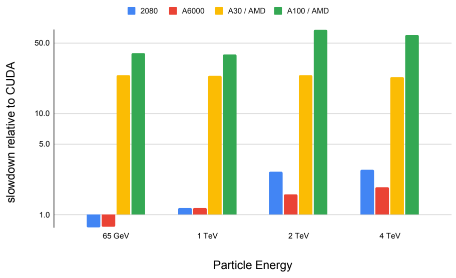

Figure 4. Memory transfers from device to host using stdpar.

when measuring the full event loop runtime. As the number of threads or processes is increased, we see performance topping out at around 12 threads/processes, though the runtime is only about 2× that of the original CPU implementation.

The performance with SYCL is much more uniform. As shown in Figure 3, we see close to native performance on a variety of hardware, including both GPU and CPUs. While we did observe some overheads from the use of Unified Shared Memory for the automatic migration of data between host and device, the amount of data transferred per event is relatively small, and thus these overheads remain small. We witnessed the same poor performance of the main simulation kernel on AMD hardware as compared with NVIDUA GPUs when using SYCL for several different SYCL compilers, including hipSYCL (now openSYCL) and llvm, though the SYCL performance was in fact slightly better than the native HIP implementation.

The performance of OpenMP is somewhat consistent with Kokkos for GPU backends. The NVIDIA GPUs are about 10× faster than AMD GPUs for 65 GeV energy runs (see Figure 5) with a larger disparity being observed in higher energy runs. We also observe that LLVM Clang's optimization flags `-fopenmp-cuda-mode` improves performance on NVIDIA GPUs by 10× but has no impact on AMD GPUs. For the host backends, OpenMP's performance improves with the number of threads (see Figure 6) even benefiting from hyperthreading up to 48 threads. Performance comparison with native programming models reveals an interesting pattern: on NVIDIA V100 for all energies, the data copy from device to host using OpenMP shows similar performance to CUDA, however, the kernels are about 1.5× to 2× slower (see Figure 7). On AMD MI100 GPUs, on the other hand, the data copy from device to host using OpenMP is 1.5× to 20× slower than HIP for different energies (see Figure 8), however, the offloaded kernels either perform better than or similar to HIP.

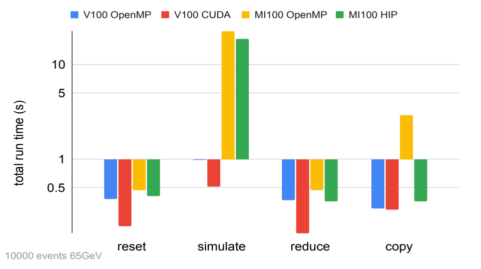

Figure 5. Performance of FastCaloSim kernels using LLVM Clang-15's OpenMP implementation for 65GeV on NVIDIA V100 and AMD MI100 GPUs.

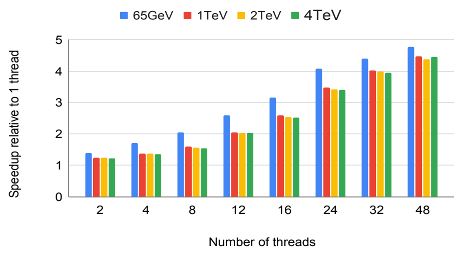

Figure 6. Performance of the kernels (reset + simulate + reduce) and device to host data copy using LLVM Clang-15's OpenMP implementation on host (AMD Ryzen Threadripper 3960X 24-Core Processor) with hyperthreading.

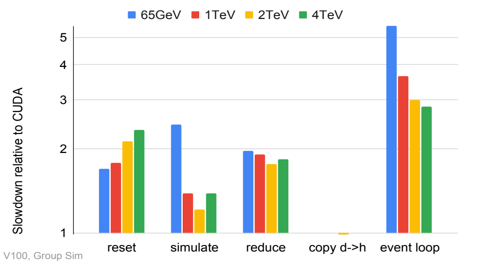

Figure 7. Performance of FastCaloSim using LLVM Clang-15's OpenMP implementation relative to native the CUDA on NVIDIA V100 for group simulations.

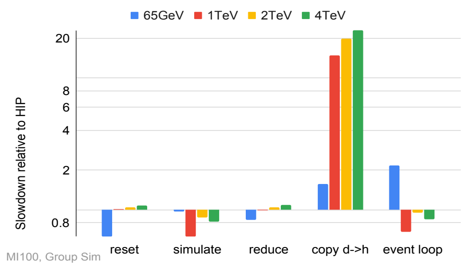

Figure 8. Performance of FastCaloSim using LLVM Clang-15's OpenMP implementation relative to native the HIP on AMD MI100 for group simulations.

The `std::par` implementation produced some very interesting results. While for single event simulation, where the kernel times are less than 100 μs, we do see a significant overhead in the main simulation kernel runtimes due to the translation to Thrust that nvc++ performs in order to run on GPUs, as the workload on the GPU increases with increased particle energies, this overhead goes down (see Figure 9). And for event batched data, where the kernel runtimes are on the order of milliseconds, `std::par` performed better than the original CUDA implementation for high energy particles (see Figure 10). We also saw a significant performance hit, sometimes as much at 70%, when using `atomicAdd(float)` from CUDA to perform atomic operations instead of a scaled `std::atomic<int>` operation. We also observed another unusual effect, where data transfers between device and host were up to 50× slower when the host had an AMD CPU instead of one from Intel (see Figure 4). This phenomena was reproduced on several systems with different combinations of GPUs and CPUs. Another interesting result was that the CPU serial backend to `std::par` ran 10-20% faster than the original CPU version, showing the advantage of refactoring the code to expose the parallelism.

Figure 11 shows the results of performance comparisons between alpaka implementations of the FastCaloSim using event-batched data with CUDA and HIP backends and corresponding native implementations. The general observation is that with alpaka we are getting quite similar performances in all tests, except for memory copying operations on M100 GPU where alpaka adds a visible overhead compared to native HIP. For the execution of simulation kernels on NVIDIA GPUs we even observe a slight speedup when using alpaka kernels compared to native CUDA implementations.

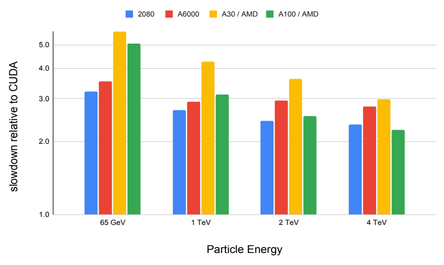

Figure 9. Performance of the main simulation kernel with std::par.

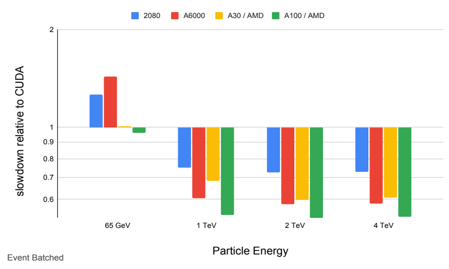

Figure 10. Performance of the main simulation kernel with std::par using event batched data.

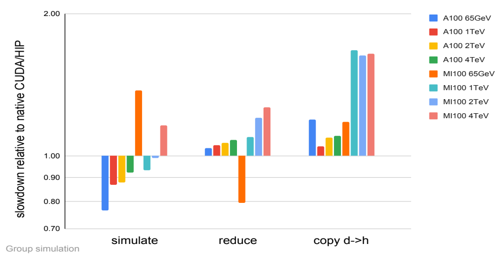

Figure 11. Performance of the kernels with alpaka using event batched data.

6 Conclusions

There are a number of competitive alternatives to CUDA currently available that enable cross platform compatibility on GPUs, including Kokkos, SYCL, OpenMP, alpaka and std::par. Each portability layer has its own characteristics, and associated advantages and disadvantages. There is no turn-key solution, and all layers require a certain amount of work to port to, though since we had a CUDA implementation to start with, the most challenging aspect of porting code to GPUs, i.e. exposing the parallelism and refactoring the exported data objects, had already been accomplished. None of the developers were familiar with the portability layers under study when this project started, so each port had an associated learning curve. std::par was the easiest to learn, as it is plain C++, and alpaka the most abstruse. Kokkos, SYCL and OpenMP fall between the two extremes. We found that for short kernels, the overheads from Kokkos structures can be much more significant than for the other portability layers. We found that the alpaka coding style to be rather verbose, with significant amounts of boilerplate code needed. Intel offers a powerful SYCL ecosystem for code development, debugging and profiling, as well as tools to aid conversion from CUDA, though it is not a turn-key process. OpenMP required a significant amount of experimenting with different compiler flags to achieve optimal performance, though was able to exercise host-parallel targets better than all others. std::par had significant hurdles in setting up the build system, which we expect to be corrected as the compiler matures, but performed remarkably well after that. We were ultimately able to extract reasonable performance from all the different portability layers.

The choice of a portability layer for a project is a complicated decision. Though important, merely evaluating performance is insufficient. There are many aspects of a project, such as external dependencies, build systems, library structure, and available hardware which must be considered. Despite being small and self-contained, while FastCaloSim has allowed us to explore many aspects of this decision phase space, it is by no means an exhaustive evaluation.

7 Acknowledgments

This work was supported by the DOE HEP Center for Computational Excellence at Brookhaven National Laboratory, and Lawrence Berkeley National Laboratory under B&R KA2401045. This work was also done as part of the offline software research and development programme of the ATLAS and CMS Collaborations, and we thank the collaborations for its support and cooperation. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231. We gratefully acknowledge the use of the computing resources provided by Brookhaven National Laboratory and by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

References

[1] Agostinelli et al, GEANT4–a simulation toolkit, Nucl. Instrum. Meth. A **506**, 250-303 (2003)
[2] The ATLAS Collaboration, The ATLAS Simulation Infrastructure, The European Physical Journal, **70**, 823-874 (2010)
[3] The ATLAS Collaboration, The new Fast Calorimeter Simulation in ATLAS, Tech. Rep. **ATL-SOFT-PUB-2018-002**, (2018)
[4] John Nickolls, Ian Buck, Michael Garland, Kevin Skadron, Scalable Parallel Programming with CUDA, ACM Queue **vol. 6 no. 2**, 40-53 (2008)
[5] Zhihua Dong, Heather Gray, Charles Leggett, Meifeng Lin, Vincent Pascuzzi, Kwangmin Yu, Porting HEP Parameterized Calorimeter Simulation Code to GPUs, Frontiers in Physics: Big Data and AI in High Energy Physics **4** (2021)
[6] Trott, Christian R. et al., Kokkos 3: Programming Model Extensions for the Exascale Era, IEEE Transactions on Parallel and Distributed Systems **vol. 33 no. 4** 805-817 (2022)
[7] H. Carter Edwards et al., Kokkos: Enabling manycore performance portability through polymorphic memory access patterns , Journal of Parallel and Distributed Computing **vol. 74 no. 12** 3202-3216 (2014)
[8] https://registry.khronos.org/SYCL/
[9] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, Computational Science & Engineering, IEEE, **vol 5 no. 1**, 46-55 (1998)
[10] M. Atif et al., Evaluating Portable Parallelization Strategies for Heterogeneous Architectures in High Energy Physics, arXiv:2306.15869 (2023).
[11] M. Lin et al., Portable Programming Model Exploration for LArTPC Simulation in a Heterogeneous Computing Environment: OpenMP vs. SYCL, arXiv:2304.01841 (2023).
[12] A. Matthes et al., Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the Alpaka library, arXiv:1706.10086 (2017)
[13] E. Zenker et al., Alpaka - An Abstraction Library for Parallel Kernel Acceleration, arXiv:1602.08477 (2016)
[14] B. Worpitz et al., Investigating performance portability of a highly scalable particle-in-cell simulation code on various multi-core architectures, doi:10.5281/zenodo.49768 (2015)
[15] https://github.com/alpaka-group/cupla
[16] https://www.anl.gov/hep-cce