EPJ Web of Conferences **251**, 03030 (2021) https://doi.org/10.1051/epjconf/202125103030
CHEP 2021

Novel features and GPU performance analysis for EM particle transport in the Celeritas code*

Seth R. Johnson[1,**], *Stefano* C. Tognini[1], *Philippe* Canal[2], *Thomas* Evans[1], *Soon Yung* Jun[2], *Guilherme* Lima[2], *Amanda* Lund[3], and *Vincent R.* Pascuzzi[4]

[1]Oak Ridge National Laboratory
[2]Fermi National Accelerator Laboratory
[3]Argonne National Laboratory
[4]Lawrence Berkeley National Laboratory

Abstract. Celeritas is a new computational transport code designed for high-performance simulation of high-energy physics detectors. This work describes some of its current capabilities and the design choices that enable the rapid development of efficient on-device physics. The abstractions that underpin the code design facilitate low-level performance tweaks that require no changes to the higher-level physics code. We evaluate a set of independent changes that together yield an almost 40% speedup over the original GPU code for a net performance increase of 220× for a single GPU over a single CPU running 8.4M tracks on a small demonstration physics app.

1 Introduction

The new High Luminosity Large Hadron Collider (HL-LHC) Era of the LHC, along with the upgrades in the detectors of its main experiments (CMS, ATLAS, ALICE, and LHCb), will result in a steep rise in computing resource usage, far beyond the expected availability within current funding scenarios [1]. Monte Carlo (MC) simulations are a large component of that expected increase, which can be reduced by utilizing the GPUs that provide the bulk of computational horsepower in modern high performance computing (HPC) due to their comparatively low power consumption.

The new *Celeritas* particle transport code aims to close the gap between the impending advanced architectures and the vast computational requirements of the upcoming HEP detector campaigns. The main focus of Celeritas is to implement full-fidelity high energy physics simulation of LHC detectors on the advanced architectures that will form the backbone of HPC over the next decade.

A short-term goal for Celeritas is a proof-of-concept app and library for simulating electromagnetic (EM) physics for photons and charged leptons on geometry models currently

*This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. DOE will provide access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

**e-mail: johnsonsr@ornl.gov

used by the high energy physics (HEP) community, starting with the CMS detector. This choice provides us the opportunity to verify the performance gain of simulating EM showers, which are the most computationally intensive part of a CMS event, on GPUs. Celeritas relies on the CUDA-compatible VecGeom [2] library to load and navigate existing Geant4 [3]-compatible geometry definitions.

This paper describes in §2 some key software architecture developments in Celeritas for enabling rapid implementation of high performance, GPU-enabled physics. Using a demonstration physics app that incorporates many of the these novel developments, we explore in §3 how changes to the computational kernel and to lower-level Celeritas components effect run time performance on a contemporary GPU.

2 Code Architecture

In the short term, Celeritas is designed as a standalone application that transport particles exclusively on device. To support robust and rapid unit testing, its components are designed to run natively in C++ on traditional CPUs regardless of whether CUDA is available for on-device execution. This is accomplished both with simple macros that hide CUDA function tags when not compiling CUDA code, and a new data model for constructing and using complex hierarchical data on CPU and copying it to GPU.

Like other GPU-enabled MC transport codes such as Shift [4, 5], the low-level component code used by transport kernels is designed so that each particle track corresponds to a single thread, since particle tracks once created are independent of each other. There is therefore essentially no cooperation between individual threads, facilitating the dual host/device annotation of most of Celeritas. The allocation of secondary particles and the initialization of new tracks from these secondaries both require CUDA-specific programming, but those components are encapsulated so that physics code can correctly allocate secondaries both in production code (on device) and in unit tests (on the host).

To support parallelizing our initial development over several team members, and to facilitate refactoring and performance testing of code, Celeritas uses a highly modular programming approach based on composition rather than inheritance. As much as possible, each major code component is built of numerous smaller components and interfaces with as few other components as possible.

2.1 Physics Interactor classes

As an example of the granularity of classes, consider the sampling of secondaries from a model. In contrast to the virtual `G4VModel::sample_secondaries` member function in Geant4 [3], each model in Celeritas defines an independent function-like object for sampling secondaries. Each `Interactor` class is analogous to a C++ standard library "distribution": the distribution parameters (including particle properties, incident particle energy, incident direction, and interacting element properties) are class construction arguments, and the function-like `Interactor::operator()` takes as its sole input a random number generator and returns an `Interaction` object, which encodes the change in state to the particle and the secondaries produced.

Table 1 summarizes the models that are implemented in Celeritas on device, as well as key challenges that were first encountered while implementing each model. Future work will elaborate on the models and the novel aspects of implementing them on GPU in Celeritas. The mini-app and performance analysis in §3 will focus on the first interactor implemented, Klein–Nishina.

Table 1. Fully implemented model interactors in Celeritas.

Model	Challenges
Klein–Nishina	Allocating secondaries
Bethe–Heitler	Accessing element properties
$e^+ \to (\gamma, \gamma)$	—
Moller/Bhabha scattering	Multiple energy distributions
Livermore photoelectric	Accessing shell data
	Calculating cross sections on the fly
...with atomic relaxation	Cascading electron vacancies
	Dynamic number of secondaries

2.2 Data model

Software for heterogeneous architectures must manage independent *memory spaces*. *Host* allocations use `malloc` or standard C++ library memory management, and the allocated data is accessible only on the CPU. *Device* memory is allocated with `cudaMalloc` and is generally available only on the GPU. The CUDA Unified Virtual Memory feature allows CUDA-allocated memory to be automatically paged between host and device with a concomitant loss in performance. Another solution to memory space management is a portability layer such as Kokkos [6], which manages the allocation of memory and transfer of data between host and device using a class `Kokkos::View<class, MemorySpace>` which can act like a `std::shared_pointer` (it is reference counted), a `std::vector` (it allocates and manages memory), and a `std::span` (it can also provide a non-owning view to stored data). A similar class has been developed for Celeritas but with the design goal of supporting complicated heterogeneous data structures needed for tabulated physics data, in contrast to Kokkos' focus on dense homogeneous linear algebraic data.

The `celeritas::Collection<class, Ownership, MemSpace>` class manages data allocation and transfer between CPU and GPU with the primary design goal of constructing deeply hierarchical data on host at setup time and seamlessly copying to device. The templated `class` must be trivially copyable—either a fundamental data type or a struct of such types. An individual item in a collection can be accessed with `ItemId<class>`, which is a trivially copyable but type-safe index; and a range of items (returned as a `Span<class>`) can be accessed with a trivially copyable `ItemRange<class>`, which is a container-like slice object containing start and stop indices. Since `ItemRange` is trivially copyable and `Collection`s have the same data layout on host and device, a set of Collections that reference data in each other provide an effective, efficient, and type-safe means of managing complex hierarchical data on host and device.

As an example, consider material definitions (omitting isotopics for simplicity), which contain three levels of indirection: an array of materials has an array of pointers to elements and their fractions in that material. In a traditional C++ code this could be represented as `vector<vector<pair<Element*, double>>>` with a separately allocated `vector<Element>`. However, even with helper libraries such as Thrust [7] there is no direct CUDA analog to this structure, because device memory management is limited to host code. We choose not to use in-kernel `cudaMalloc` calls for performance and portability considerations. Instead, Celeritas represents the nested hierarchy as a set of `Collection` objects bound together as a `MaterialParamsData` struct.

```
struct Element
{
    int            atomic_number;
```

```
    units::AmuMass atomic_mass;
};

struct MatElementComponent
{
    ItemId<Element> element;
    real_type       fraction;
};

struct Material
{
    real_type   number_density;
    real_type   temperature;
    MatterState matter_state;
    ItemRange<MatElementComponent> components;
};

template<Ownership W, MemSpace M>
struct MaterialParamsData
{
    template<class T> using Items = celeritas::Collection<T, W, M>;

    Items<Element>             elements;
    Items<MatElementComponent> elcomponents;
    Items<Material>            materials;
    unsigned int               max_elcomponents;

    template<Ownership W2, MemSpace M2>
    MaterialParamsData& operator=(const MaterialParamsData<W2, M2>& other);
};
```

A `MaterialParamsData<Ownership::value, MemSpace::host>` is constructed incrementally on the host (each `Items<class>` in that template instantiation is a thin wrapper to a `std::vector`), then copied to a `MaterialParamsData<Ownership::value, MemSpace::device>` (where each `Items<class>` is a separately managed `cudaMalloc` allocation) using the templated assignment operator. The definition of that operator simply assigns each element from the `other` instance. For primitive data such as `max_elcomponents`, the value is simply copied; for `Items` the templated `Collection::operator=` performs a host-to-device transfer under the hood.

To access the data on device, a `MaterialParamsData<Ownership::const_reference, MemSpace::device>` (where each `Items<class>` is a `Span<class>` pointing to device memory) is constructed using the sample assignment operator from the `<value, device>` instance of the data. The `const_reference` and `reference` instances of a Collection are trivially copyable and can be passed as kernel arguments or saved to global memory.

The first material in a `MaterialParamsData<Ownership::const_reference, MemSpace::device> data` instance can be accessed as:

```
const Material& m = data.materials[ItemId<Material>(0)];
```

A view of its elemental component data is:

```
Span<constMatElementComponent> els = data.elcomponents[m.components];
```

And the elemental properties of the first constituent of the material are:

```
const Element& el = data.elements[els[0].element];
```

In practice, the `MaterialParamsData` itself is an implementation detail constructed by the host-only class `MaterialParams` and used by the device-compatible class `MaterialView` and `ElementView`, which encapsulate access to the material and element data. In Celeritas, `View` objects are to `Collection` as `std::string_view` is to `std::vector<char>`: Collections and vectors only store the underlying data.

2.3 States and parameters

The Celeritas data model is careful to separate persistent shared "parameter" data from dynamic local "state" data, as there will generally one independent state per GPU thread. To illustrate the difference between parameters and states, consider the calculation of the Lorentz factor γ of a particle, which is a function of both the rest mass mc^2—which is constant for all particles of the same type but is not a fundamental constant nor the same for all particles—and the kinetic energy K. It is a function of K parameterized on the mass m: $\gamma(m; K) = 1 + K/mc^2$. Celeritas differentiates shared data such as m (parameters, shortened to `Params`) from state data particular to a single track such as kinetic energy K or particle type (`State`). Inside transport kernels, the `ParticleTrackView` class combines the parameter and state data with the local GPU thread ID to encapsulate the fundamental properties of a track's particle properties -- its rest mass, charge, and kinetic energy, as well as derivative properties such the magnitude of its relativistic momentum.

In Celeritas, a particle track is not a single object nor a struct of arrays. Instead, sets of classes (Params plus State) define aspects of a track, each of which is accessed through a `TrackView` class. Table 2 shows the current track attributes independently implemented in Celeritas.

Table 2. Existing groups of per-track state and parameter data in Celeritas.

Module	State	Params
Particle	Kinetic energy and particle ID	Properties for each particle type
Material	Current material ID	Material densities, elements, etc.
Geometry	Position, direction, volume ID, NavState	Geometry description
Sim	Track ID, Event ID, time	—
Physics	Distance to interaction, cached cross sections	Models, processes, cross section tables

In addition to the per-track attributes, each hardware thread (which may correspond to numerous tracks in different events over the lifetime of the simulation) has a persistent random number state initialized at the start of the program.

2.4 On-device allocation

One requirement for transporting particles in electromagnetic showers is the efficient allocation and construction of secondary particles. The number of secondaries produced during an interaction varies according to the physics process, random number generation, and other properties. This implies a large variance in the number of secondaries produced from potentially millions of tracks undergoing interactions in parallel on the GPU.

To enable runtime dynamic allocation of secondary particles, we have authored a function-like `StackAllocator<class>` templated class that uses a large on-device allocated array with a fixed capacity along with an atomic addition to unambiguously reserve one or more items in the array. The call argument to a `StackAllocator` takes as an argument the number of elements to allocate, and if allocation is successful, it uses placement new to default-initialize each element and returns a pointer to the first element. If the capacity is exceeded during the allocation (or by a parallel thread also in the process of allocating), a null pointer is returned. Figure 1 describes the allocation algorithm, including the error conditions necessary to ensure that the size of the allocated elements is correct even in the case of an overflow.

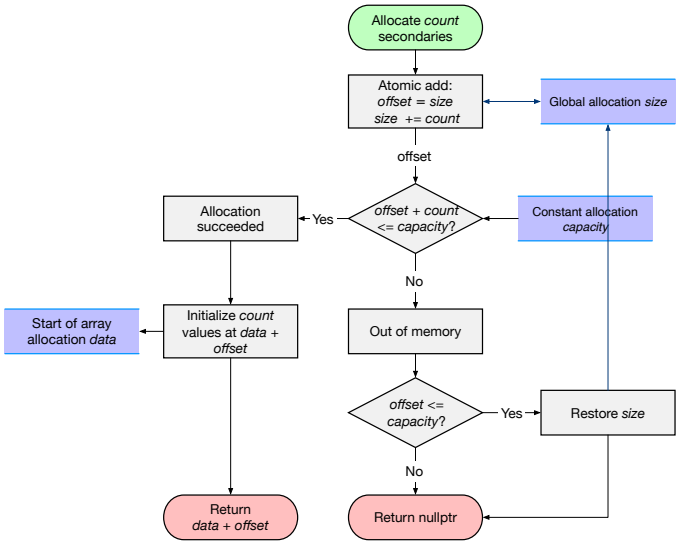

Figure 1. Flowchart of the stack allocation algorithm.

To accommodate large numbers of secondaries on potentially limited GPU memory, we define a `Secondary` class that carries the minimal amount of information needed to reconstruct it from the parent track, rather than as a full-fledged track. It comprises a particle ID (an `ItemId` into an array of particle data), an energy, and a direction.

The final aspect of GPU-based secondary allocation is how to gracefully handle an out-of-memory condition without crashing the simulation *or* invalidating its reproducibility. This can be accomplished by ensuring that no random numbers are sampled before allocating storage for the secondaries, and by adding a external loop over the interaction kernel (Fig. 2) to reallocate extra secondary space or process secondaries so that all interactions can successfully complete in the exceptional case where the secondary storage space is exceeded.

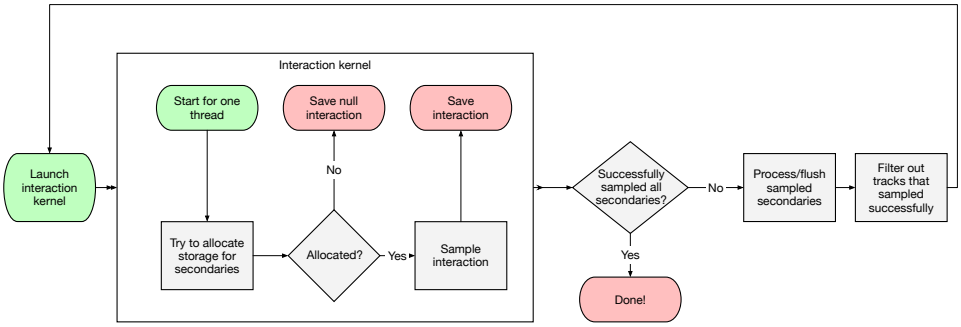

Figure 2. Flowchart for an interaction kernel wrapped in a host-side loop for processing secondaries.

3 Mini-App Results

Using the Celeritas components, we constructed a demonstration app to verify a simple test problem against Geant4 [3] results. It is the simplest physical simulation we can run, with

photon-only transport and a single Compton scattering process using the Klein–Nishina model. It has a single infinite material (aluminum) and a 100 MeV monodirectional point source.

The stepping kernel is parallel over particle tracks, with one launch per step, and "dead" tracks ignored. Interaction lengths are sampled with uniform-in-log-energy cross section calculations with linear interpolation. The particle states include position and directions that are updated with each step. Secondaries are allocated and constructed as part of each interaction, but they are immediately killed and their energy deposited locally. Each energy deposition event, whether from an absorbed electron or a cutoff photon, allocates (using a `StackAllocator<Hit>` instance) a detector hit and writes the deposited energy, position, direction, time, and track ID to global memory.

An additional kernel processes the allocated vector of detector hits into uniformly spaced detector bins. A final kernel performs a reduction on the "alive" state of particles to determine whether the simulation should terminate. These helper kernels are included in the timings reported below.

The same code components were used to build GPU and CPU versions of the same stepping process, although the CPU version steps through one track at a time (single-threaded, no vectorization) rather than many tracks in parallel. Figure 3 gives the baseline performance of the two versions. The host code uses a single core of an Intel "Cascade Lake" Xeon processor running at 2.3 GHz, compiled with GCC 8.3 and `-O3 -march=skylake-avx512 -mtune=skylake-avx512`. The device code uses a single Nvidia Tesla V100 running at 1.53 GHz, compiled with CUDA 10.1 and `-O3 -use_fast_math`.

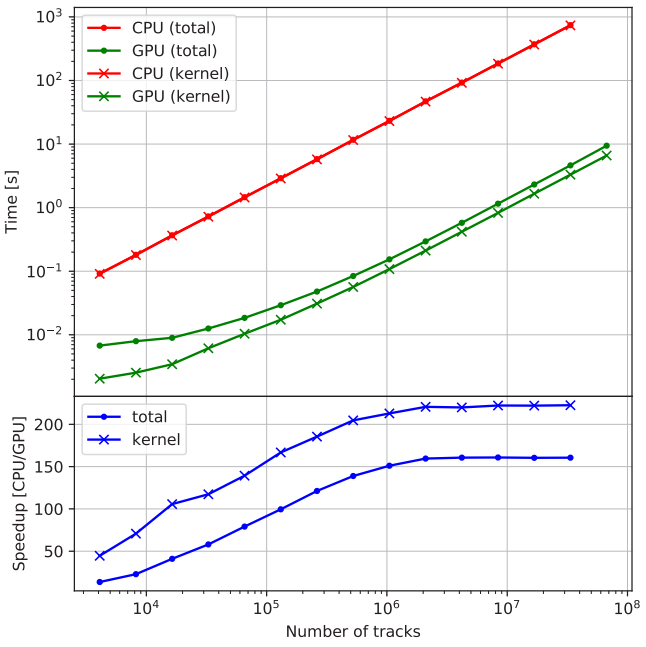

Figure 3. Performance comparison of the CPU and original GPU versions of the Celeritas code. The "total" GPU plot includes the extra kernel launches for processing detector hits and the number of living tracks.

A roofline analysis (Fig. 4) shows that the interaction kernel is not limited by raw memory bandwidth or floating point performance. Similar behavior has been observed in other GPU Monte Carlo particle transport algorithms [5], and it is characteristic of the method in general. Due to a combination of both high-latency and random-access data patterns and low arithmetic intensity, Monte Carlo particle transport will generally never achieve full memory or floating-point bounded performance on a given architecture.

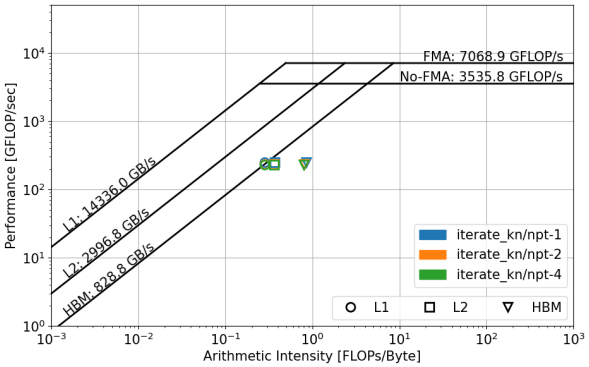

Figure 4. Roofline plot showing the interaction kernel performance plotted against the theoretical memory bandwidth and arithmetic performance limitations. The 1/2/4 points are the number of particle tracks processed by a single GPU thread.

Although this mini-app is far simpler than one that supports the full range of physics needed for EM shower simulation, it may be instructive to see how standard recommendations for performance enhancement affect the total runtime. We experimented with the demo app and underlying Celeritas components by independently making each of the following changes:

1. *Removing the "grid striding" wherein a single GPU thread can transport multiple tracks sequentially.* Preliminary studies showed decreased performance by transporting multiple tracks per thread, so in all the results shown here only one track is used per thread. Removing the grid striding should reduce the register pressure for the kernel by eliminating an unused runtime variable.
2. *Copying track states into local variables.* Operating on the position, direction, and time locally (rather than as pointers to global memory) should remove potential aliasing issues and improve potential compiler optimizations.
3. *Copying the random number generator state into a thread-local variable.* Like with the other track states, the RNG state (XORWOW) is accessed directly through global memory. The CURAND documentation notes that to increase performance, the generator state can be operated on locally but stored in global memory between kernel launches. If the increased register usage spills into local memory, then at least the memory usage will be coalesced.
4. *Using a struct-of-arrays rather than array-of-structs for particle data.* For the sake of expediency, the `ParticleTrackState` data is a struct with a particle type ID (`unsigned int`) and an energy (`double`). Conventional CUDA kernels obtain higher memory bandwidth when data accesses are "coalesced," which will be more likely when each component is a contiguous array. We should note that the `ParticleTrackView` abstraction completely hides this implementation change from the tracking kernel and the physics code.

5. *Preallocating one secondary per interaction.* Rather than using the dynamic `SecondaryAllocator` and its atomic add, preallocate a single `Secondary` as part of the `Interaction` result. Physics kernels that allocate more than one secondary per interaction will still need the dynamic allocation, but simpler kernels will no longer need the atomic, at the cost of slightly increased memory pressure.
6. *Splitting the single step kernel into two kernels, one for movement and one for interaction.* Smaller kernels tend to have lower register usage and therefore higher occupancy.
7. *Using a 32-bit instead of 64-bit integer for the stack allocator.* Smaller data reduces memory bandwidth, and CUDA operations tend to be inherently faster for 32-bit types such as the native unsigned int and single-precision floats.

Figure 5 shows the relative GPU performance of each separate change in the list above, as well as a combination of all changes.

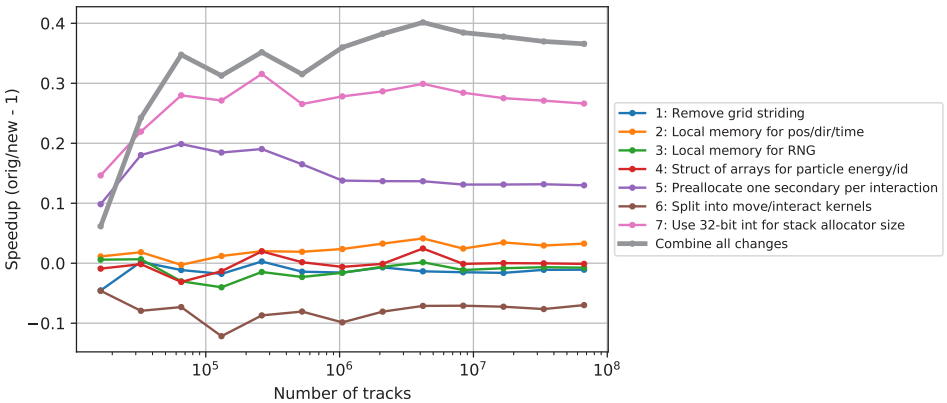

Figure 5. Incremental (thin colored lines) and cumulative (thick gray line) speed up for changes to the mini-app and data structures.

Aggregating the speedup values for cases with more than 10^6 tracks (mean and 1σ given for speedup = original/adjusted − 1, in percent):

1. *Removing the "grid striding"* had a slightly negative impact ($-1.3\% \pm 0.3\%$). Diagnostics showed that removing grid striding did in fact increase occupancy from 37.5% to 50%, but this did not translate to improved performance.
2. *Copying track states into local variables* had a small positive effect ($+3.1\% \pm 0.6\%$). Inspecting the PTX assembly code showed that with the cost of a few extra arithmetic operations, the improved kernel decreased the number of global memory loads from 41 to 29, because storing the values locally informs the compiler that there are no aliasing effects that might cause hidden dependencies between the data. However, this 25% reduction in global memory accesses resulted only in a marginal speedup.
3. *Copying the RNG states into a local variable* had essentially no effect ($-0.8\% \pm 0.5\%$). Examining the emitted PTX code shows almost no change between the original and modified version. This suggests that the extensive use of inline functions in Celeritas allows the compiler to determine that the RNG state (a struct of a type not used anywhere else in the code) cannot be aliased or modified by external functions calls, and thus does not have to be reloaded between subsequent calls to the RNG functions.

4. *Using a struct-of-arrays rather than array-of-structs for particle data* had zero significant effect. Coalescing memory access in this demo app appears unimportant: the minor reduction in memory transactions is swamped by the cost of the inherently random accesses for the cross section calculation and dynamic allocations.
5. *Preallocating one secondary per interaction* improved performance more than any other change thus far (+13.4% ± 0.3%). There was one fewer atomic operation (the detector "hits" still remained) and a decrease in global memory accesses from loading the allocated secondary to process.
6. *Splitting the single step kernel into two kernels* had the most negative effect (−7.7% ± 0.9%). This is not unexpected because each kernel launch has additional overhead, and each independent kernel has to reload data from global memory that might otherwise be stored in registered. Still, a 10% drop in performance might provide a substantial gain in code flexibility and extensibility.
7. *Using a 32-bit instead of 64-bit allocation size* had the most positive individual change (+28.0% ± 0.1%). In conjunction with the secondary preallocation result, this suggests that the atomic operations may be the single most expensive aspect of this simple demonstration kernel.

The overall speedup of +38% suggests that the faster and fewer atomic operations negate the performance drop of the atomic-heavy interaction split kernel.

Of these results, the lack of performance gain for a substantial increase in occupancy is perhaps the most surprising. Loosely stated, occupancy measures the ratio of threads that can be *active* to the maximum number of threads on a streaming multiprocessor (SM), which is the core hardware computational component of a CUDA card. Higher occupancy can hide latency to improve overall kernel performance. To explore the performance implications of higher occupancy, we recompiled the fully "optimized" kernel with the `--maxrregcount N` NVCC compiler option to constrain kernel register usage, which is the limiting factor for the CUDA SM occupancy for the demo interactor kernels. Figure 6 demonstrates that a higher kernel occupancy does not always translate to improved performance.

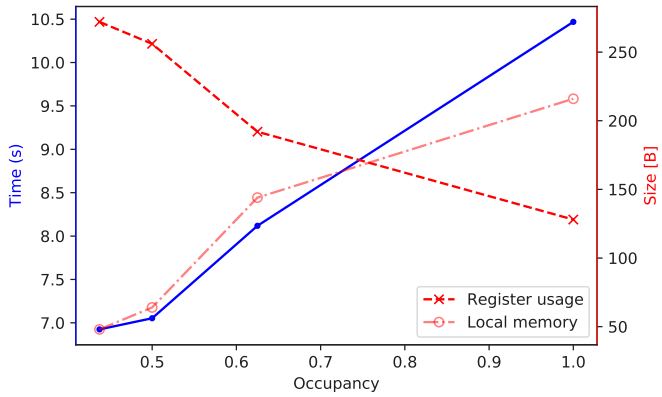

Figure 6. Performance ramifications of forcing the register size to be smaller for higher occupancy. The blue line is total solve time, the dark red line is the register usage, and the light red line is local memory usage including memory spills.

4 Conclusion

The nascent Celeritas project has established with its early developmental phase a foundation of core classes and design patterns to facilitate development of an efficient, GPU-enabled particle transport code for high energy physics. The initial results from a simple but nontrivial GPU physics simulation showed an "unoptimized" factor of about 160× performance improvement over a single-CPU version of the same simulation with identical underpinning components. A set of potential optimizations was independently examined, and their cumulative effect boosted the speedup to about 220× relative to the CPU version. Since adding additional kernels increased the overall runtime, we expect a complete EM physics app to have a smaller speedup than this mini-app. The ease of implementing and testing each change promises that the infrastructure in place will enable similar optimizations in the future full-featured physics application.

The next step in the Celeritas development will incorporate VecGeom geometry transport, multiple materials, and multiple physics models, including continuous-slowing-down processes. Future work will present the design choices made to enable these additional features, as well as more CPU comparisons and GPU performance analysis.

5 Acknowledgements

Work for this paper was supported by Oak Ridge National Laboratory (ORNL), which is managed and operated by UT-Battelle, LLC, for the U.S. Department of Energy (DOE) under Contract No. DEAC05-00OR22725 and by Fermi National Accelerator Laboratory, managed and operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy. This research was supported by the Exascale Computing Project (ECP), project number 17-SC-20-SC. The ECP is a collaborative effort of two DOE organizations, the Office of Science and the National Nuclear Security Administration, that are responsible for the planning and preparation of a capable exascale ecosystem—including software, applications, hardware, advanced system engineering, and early testbed platforms—to support the nation's exascale computing imperative.

References

[1] The HEP Software Foundation, J. Albrecht, A.A. Alves, G. Amadio, G. Andronico, N. Anh-Ky, L. Aphecetche, J. Apostolakis, M. Asai, L. Atzori et al., Computing and Software for Big Science **3**, 7 (2019)

[2] *VecGeom software project*, https://gitlab.cern.ch/VecGeom/VecGeom, accessed: 2021-02-23

[3] S. Agostinelli, J. Allison, K. Amako, J. Apostolakis, H. Araujo, P. Arce, M. Asai, D. Axen, S. Banerjee, G. Barrand et al., Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **506**, 250 (2003)

[4] T.M. Pandya, S.R. Johnson, T.M. Evans, G.G. Davidson, S.P. Hamilton, A.T. Godfrey, Journal of Computational Physics **308**, 239 (2016)

[5] S.P. Hamilton, T.M. Evans, Annals of Nuclear Energy **128**, 236 (2019)

[6] H.C. Edwards, C. Trott, *Kokkos, Manycore Device Performance Portability for C++ HPC Applications*, in *GPU Technology Conference* (San Jose, CA, 2015)

[7] Tech. Rep. DU-06716-001_v8.0, NVIDIA (2017)