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

Charles Leggett **for HEP-CCE**

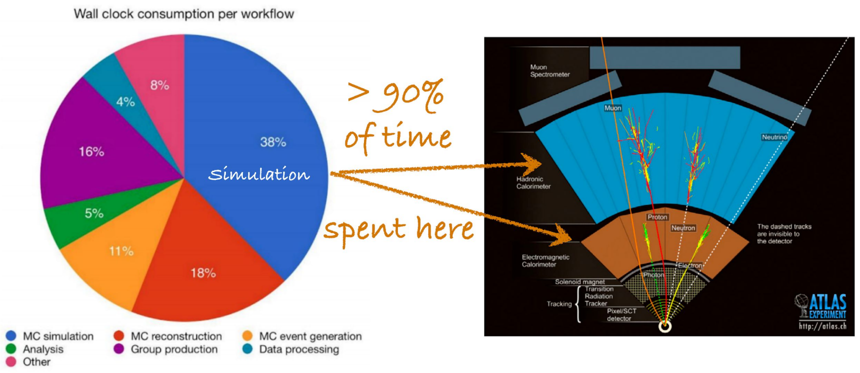
CHEP 2023
May 9, 2023

Fast Calorimeter Simulation for GPU Portability Studies

ATLAS needs lots of simulation

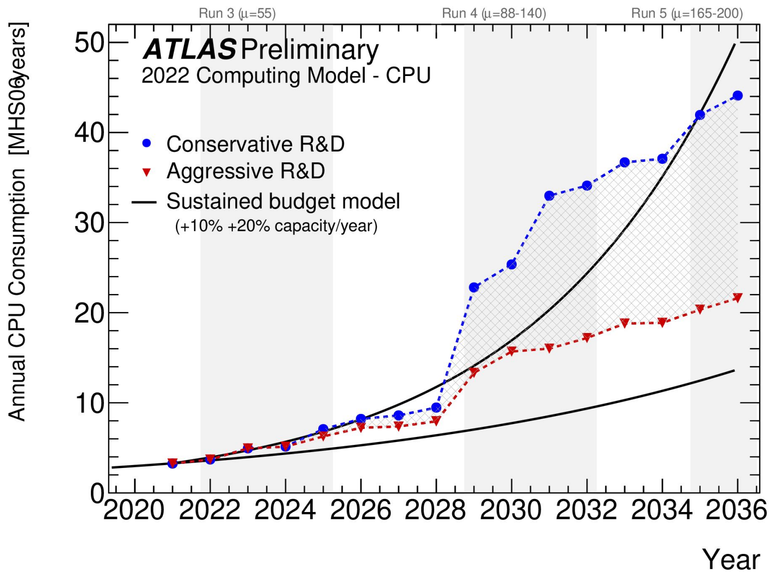
- Simulation for background modeling is paramount for precision physics
- Lack of MC-based statistics limited results in Run-2
  - will be worse for Run-3 and beyond

Calorimeter-dominated



A very large fraction of the simulation's computational budget is spent in the LAr Calorimeter

- Parametrized simulation is enormously faster than full Geant4 simulation (complex detector geometry)



FastCaloSim is small, self-contained, has few dependencies, and already has a CUDA port

- Offloading simulation to GPUs can help stay within ATLAS's compute budget
- 3 "kernels": workspace reset, simulate, reduce plus small data transfers from device to host
- Code organized to share maximum functionality between all implementations

Portability Solutions

GPUs are becoming dominant source of computing power in HPCs

- Multiple competing architectures: NVIDIA, AMD, Intel
- Different programming languages for each architecture
- Experiments lack human resources to re-code for each architecture

- **products are rapidly evolving**
- some hope of seeing emergence of industry standards at the language level

Investigate portability APIs as part of HEP-CCE/PPS's mission

- Kokkos
- SYCL
- OpenMP
- alpaka
- std::execution::parallel

	CUDA	Kokkos	SYCL	HIP	OpenMP	alpaka	std::par
NVIDIA GPU			*intel/llvm compute-cpp*	*hipcc*	*nvc++ LLVM, Cray GCC, XL*		*nvc++*
AMD GPU			*openSYCL intel/llvm*	*hipcc*	*AOMP LLVM Cray*		
Intel GPU			*oneAPI intel/llvm*	CHIP-SPV: *early prototype*	*Intel OneAPI compiler*	*prototype*	*oneapi::dpl*
x86 CPU			*oneAPI intel/llvm computecpp*	*via HIP-CPU Runtime*	*nvc++ LLVM, CCE, GCC, XL*		
FPGA				*via Xilinx Runtime*	*prototype compilers (OpenArc, Intel, etc.)*	*protytype via SYCL*	

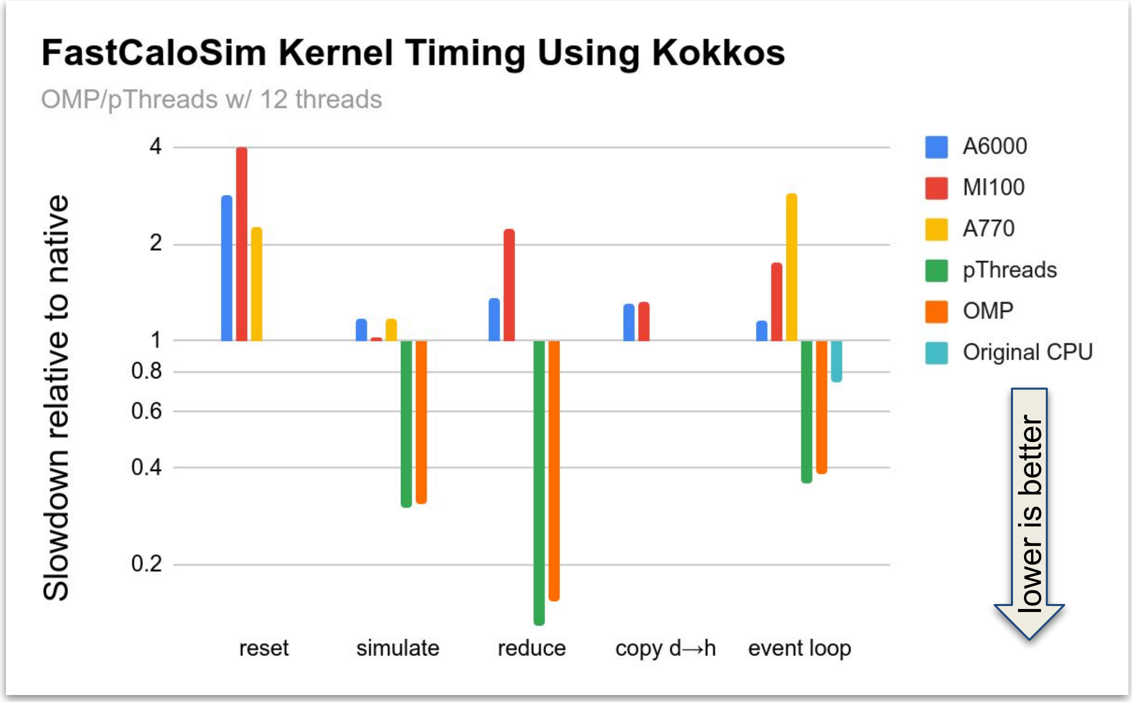
Kokkos

C++ abstraction layer that supports parallel execution and data management for different host and accelerator architectures

- host and device parallel backends must be explicitly specified at library compile time
- exercised NVIDIA, AMD, Intel, serial CPU and multi-core CPU backends

Kokkos performs similarly to "native" for simple computational kernels

- overheads from initialization of Kokkos::Views and extra launch latencies
- multicore: 2.5x perf w/ 12 threads
- requires explicit initialization and finalization
- no support for jagged arrays
- excellent developer community and support



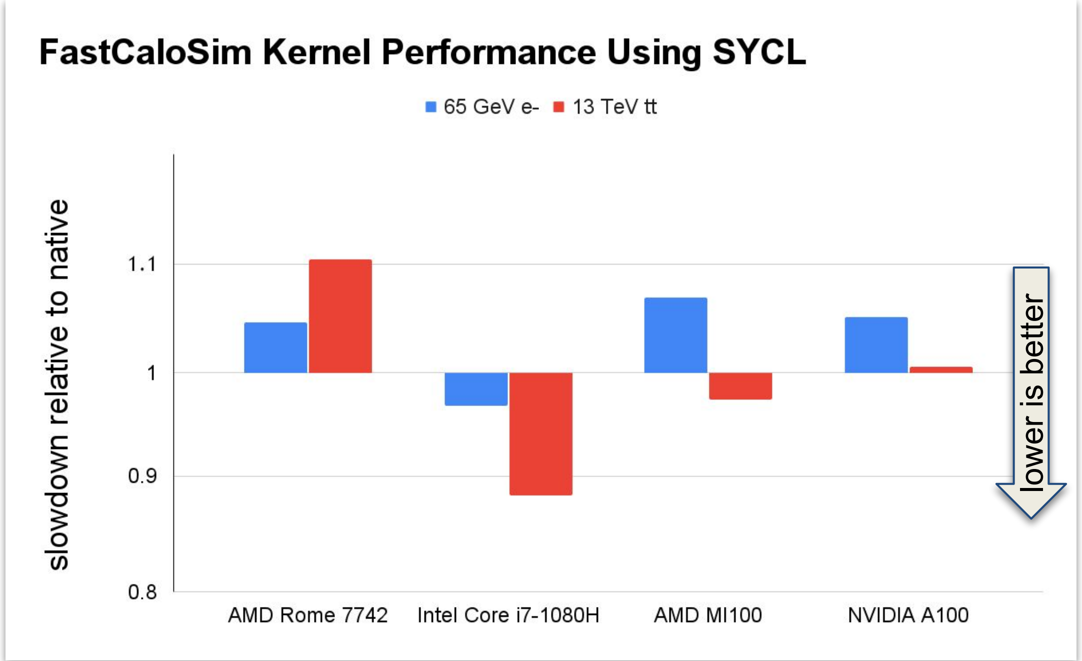
SYCL

Created by Khronos group, supported by Intel, cross platform C++ specification

- rapidly evolving implementations over the past 3 years
- different backends may require different compilers (Intel / llvm / openSYCL)
- more verbose than CUDA, though similar to Kokkos for memory management when using buffers
- DAG-based runtime satisfies inter-kernel data dependencies (buffers)
  - USM requires more explicit control from developer, but generally more performant

Near native performance

- Intel's CUDA→SYCL migration tool somewhat useful for ideas and boilerplate
- exercised Intel, NVIDIA, AMD backends



U.S. DEPARTMENT OF ENERGY | Office of Science
Argonne NATIONAL LABORATORY
Brookhaven National Laboratory
Fermilab
BERKELEY LAB Bringing Science Solutions to the World

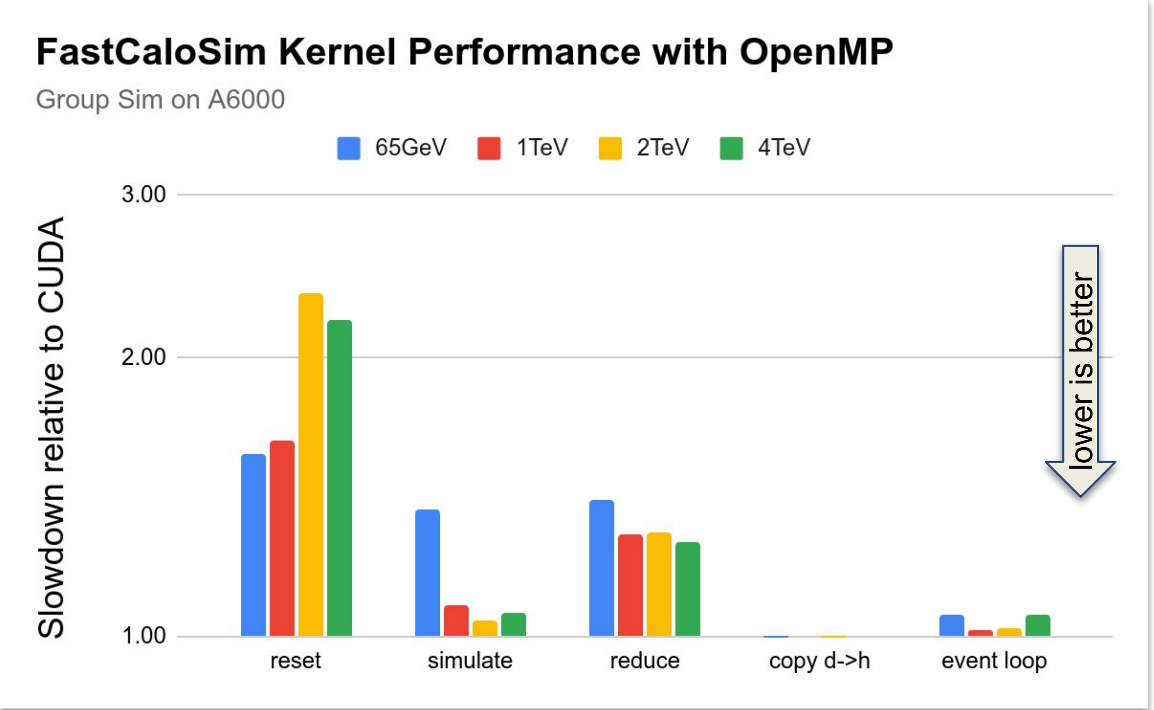
OpenMP

Directive-based programming models.

- Specifications for parallel execution on different host and accelerator architectures
- “Target offload” model adopted by many community (LLVM Clang, GCC) and vendor compilers (Nvidia, AMD, Intel)

Performance varies across compilers and hardware

- Easy to implement, does not require major changes to the C++ code
- Extracting performance requires fine tuning
- Specialized operations (e.g. atomic) less performant than CUDA
- under active development



Related poster: Porting ATLAS FastCaloSim to GPUs with OpenMP Target Offloading

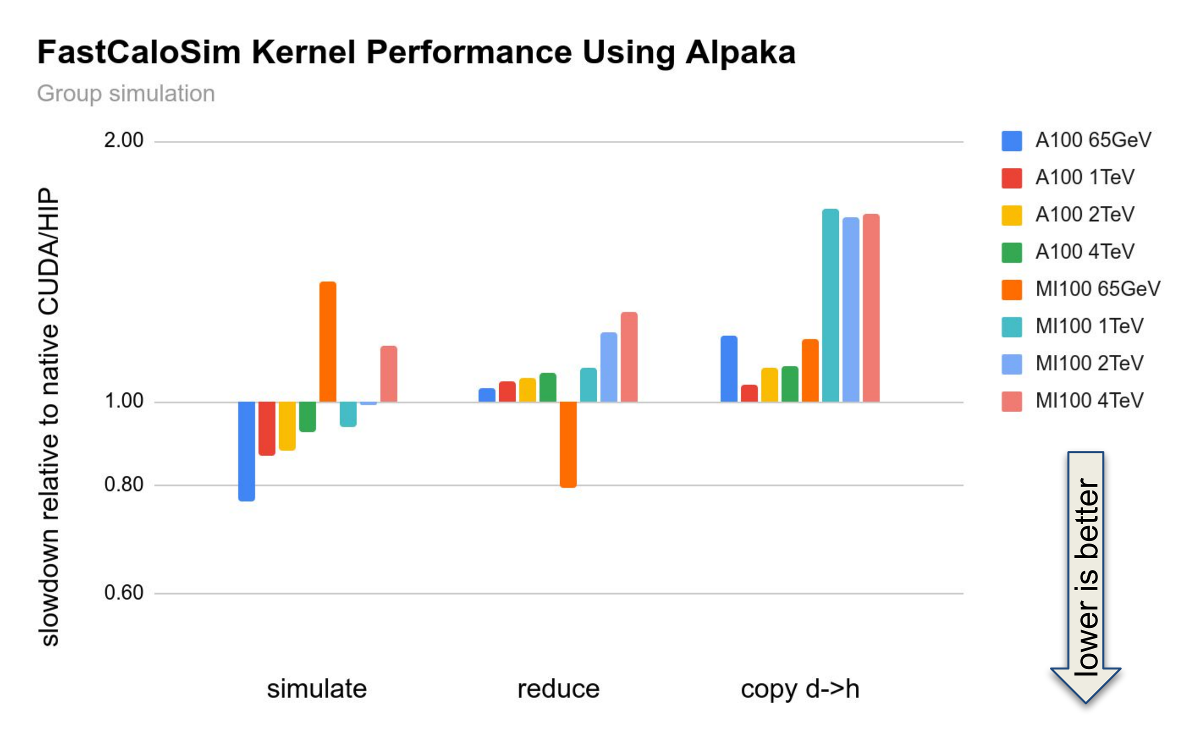
alpaka

Abstraction layer similar to Kokkos. C++ header-only library, supports wide range of compilers (g++, clang, MS Visual Studio), portable across platforms (Linux, MacOS, Windows)

- Host and device parallel backends must be specified at library compile time
- Supports CPU (C++ Threads, Intel TBB, OpenMP) and GPU (CUDA, HIP) backends

alpaka kernel performance is comparable with native implementations

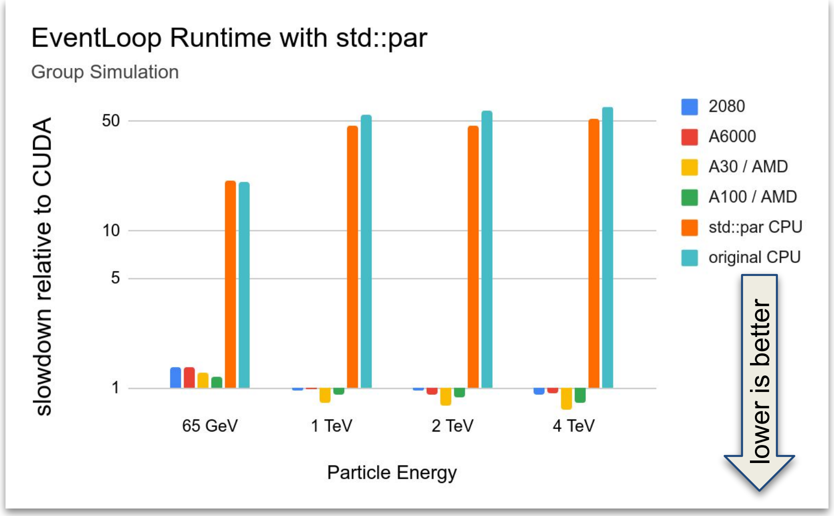
- Kernels must be wrapped into alpaka function objects (minimal overhead)
- Memory operations are performed using reference-counted smart memory buffers
- Task parallelism is implemented using blocking and non-blocking queues



Related poster: Porting ATLAS FastCaloSim to GPUs with alpaka and std::par

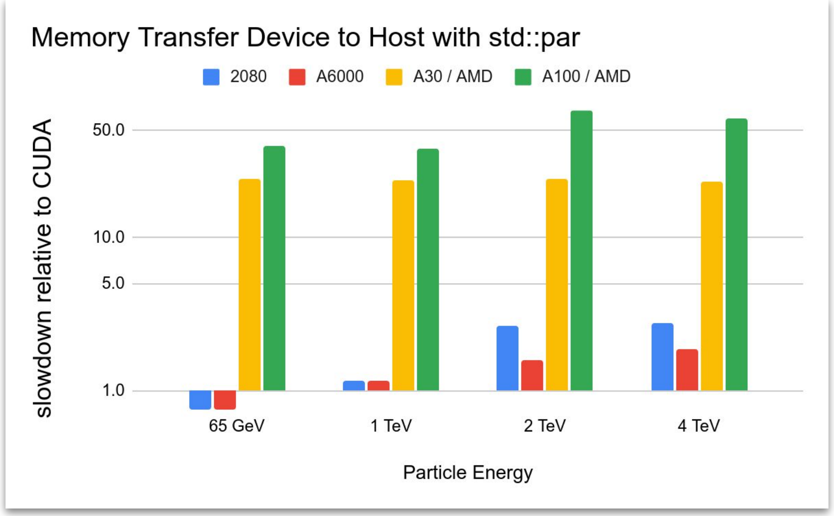
std::execution::parallel

C++17 standard for parallel execution of algorithms
- no low level control of device
- not intended to be a CUDA replacement, but a stepping stone to GPU usage



Not a true "portability layer". Yet.
- NVIDIA (via nvc++) and Intel (dpl)
- can't compile ROOT: use g++
  - complex compiler wrapper and linker issues

Data automatically migrated to device on page faults
- re-copy memory allocated with g++
- odd behaviour with AMD CPUs



For C++17, requires a "`CountingIterator`" for indexed access to containers.

Uses Thrust to implement code on GPUs
- excellent performance for "simple" kernels with adequate workloads, but poor for small workloads

Conclusions

FastCaloSim is a simple testbed that can be used to explore different APIs

- Simplicity also hides issues that more complex project would expose
- Can achieve bitwise reproducibility with appropriate compiler flags, floating point precision, and choice of RNG

There are strengths and weaknesses for each portability layer

- For simple projects, they all perform equally well (with tuning)
- Interaction with external libraries should be considered
- Excellent support for Kokkos and SYCL
  - alpaka has good support, but limited user base
- OpenMP is most widely supported API, and broadly used on HPCs
  - highest variability with compiler flavours
- std::par / ISO C++ is rapidly evolving, offers very low entry bar to usage
  - best chance to embrace a standard, functionality will grow with C++26 (std::async)

fin

This work was supported by the U.S. Department of Energy, Office of Science, Office of High Energy Physics, High Energy Physics Center for Computational Excellence (HEP-CCE). This research used resources at Argonne Leadership Computing Facility, NERSC and BNL Scientific Data and Computing Center.