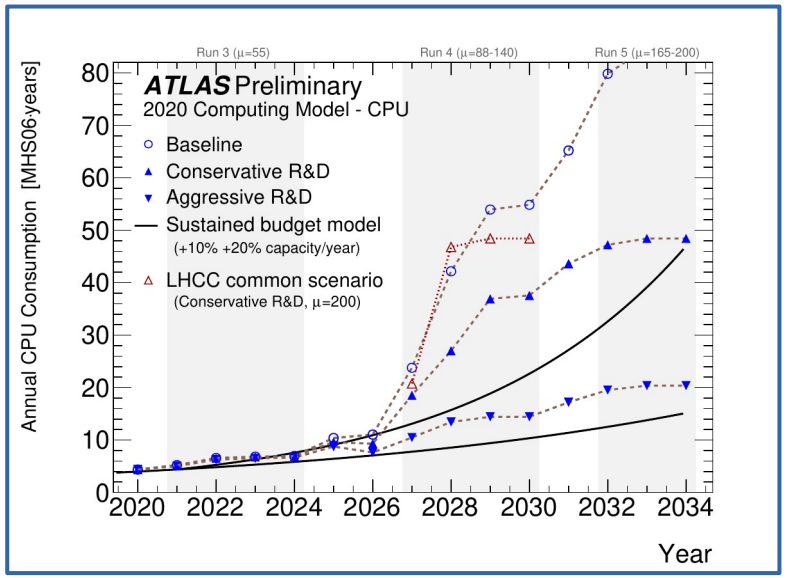
Investigating Portable Heterogeneous Solutions with Fast Calorimeter Simulation

C. Leggett, V. Pascuzzi

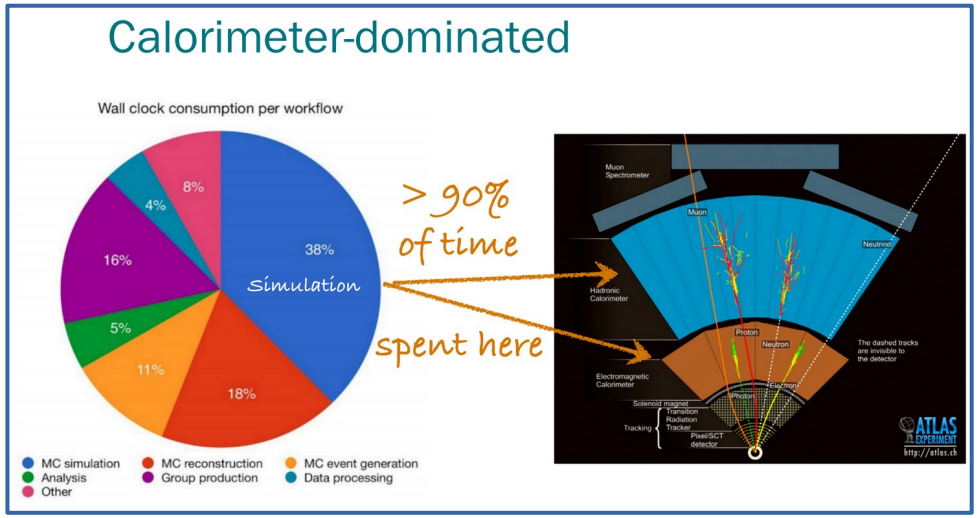
HSF/WLCG Workshop
November 24 2020

Why Fast Calorimeter Simulation for GPU Portability Studies?

- ATLAS needs lots of simulation
  - Simulation is paramount for SM and background modeling in most analyses, as well as general detector and upgrade studies
  - A significant issue in Run-2 was the lack of MC-based statistics, which will only worsen in Run-3 and beyond without faster production



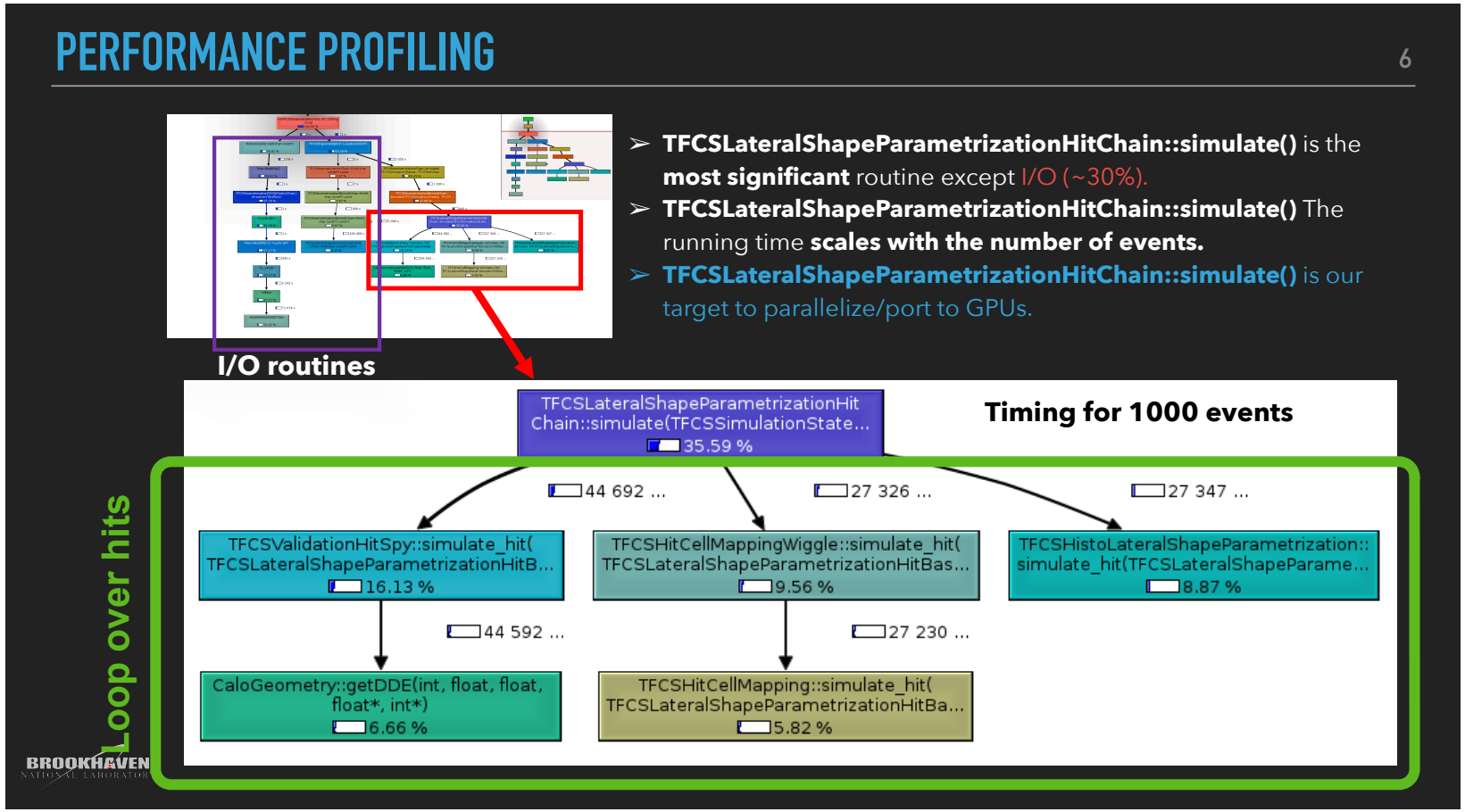
- A very large fraction of the simulation's computational budget is spent by the LAr Calorimeter
  - Parametrized simulation can speed things up enormously over full G4 Sim: FastCaloSim



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

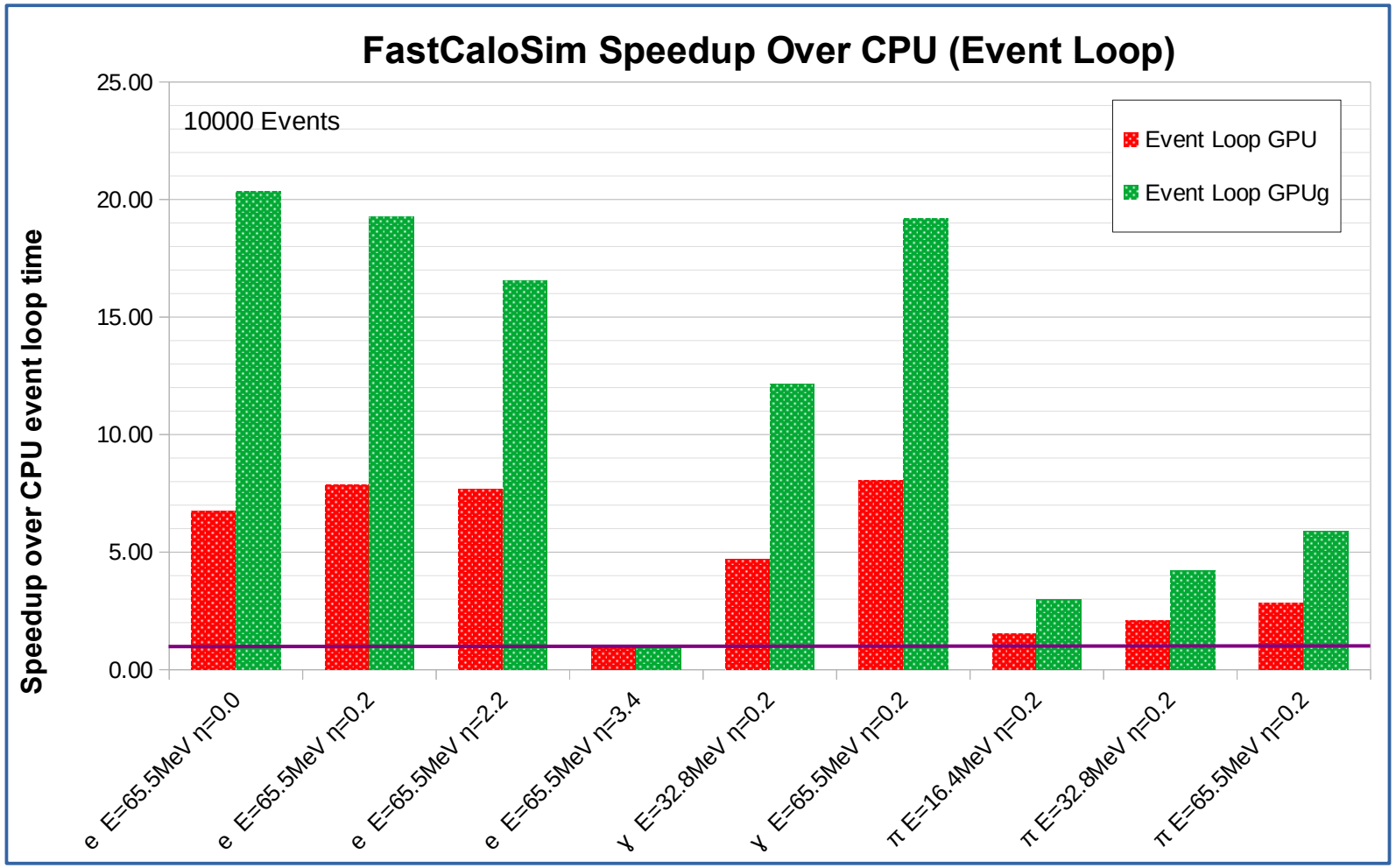
CPU Code Profiling

- LAr Calorimeter has massive inherent parallelism – lots of independent cells and associated tasks.
- Profiling studies identified hotspots that are paralellizable
- CUDA kernels created to run these parts on the GPU
  - modified data structures
  - reimplement Geometry and parametrization tables for GPU – no STL allowed
  - 3 kernels:
    - reinit memory
    - main simulation
    - reduction



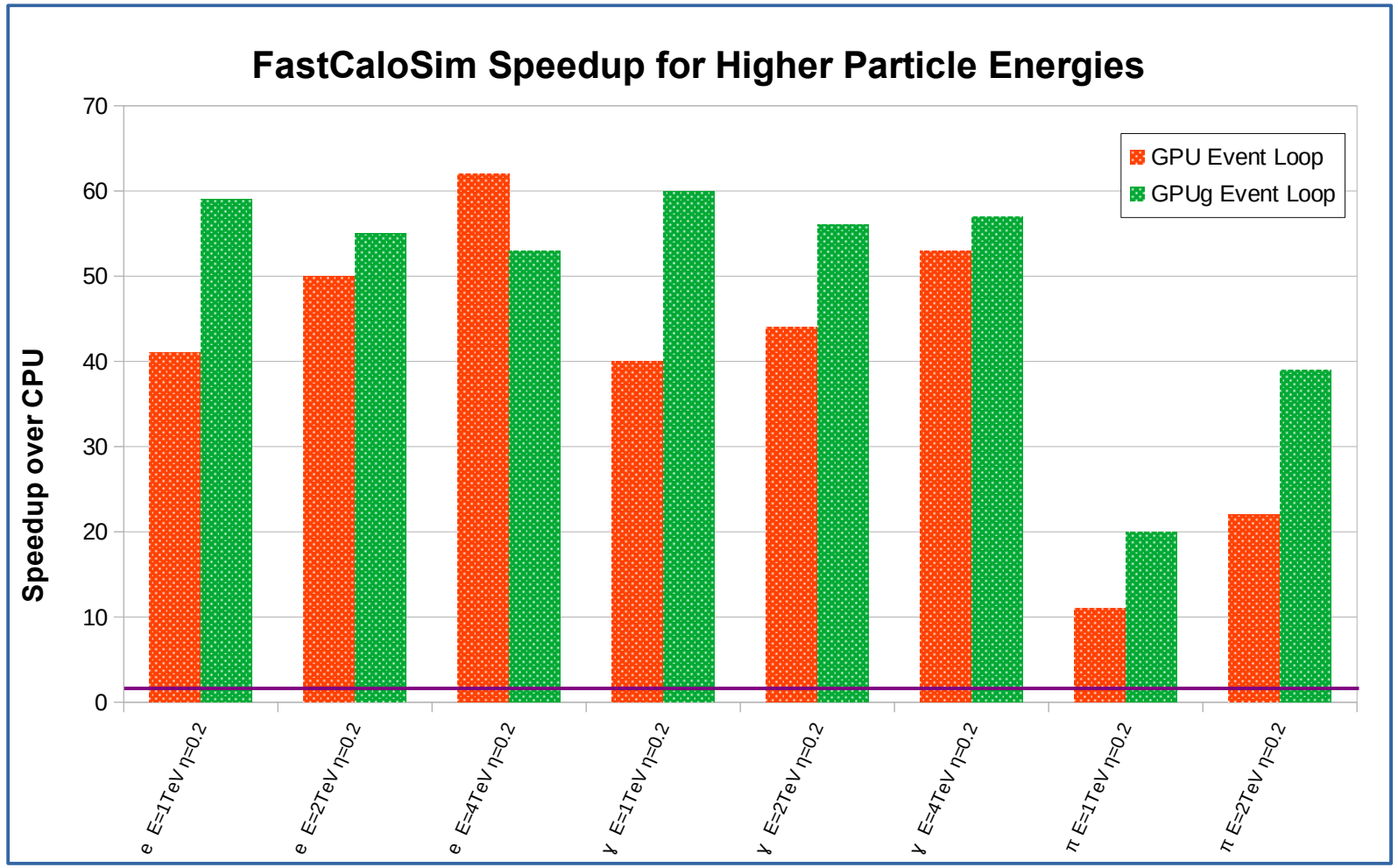
CUDA Performance Studies

- I/O to read/unpack parametrization files is expensive: ~15s of 30s
- Execution only offloaded if >500 hits, otherwise CPU is faster
  - eg, forward regions
- GPU kernels very short
  - **launch latency limited**
- Better performance if **group work** between multiple events to give more work to GPU



CUDA Performance Studies

- GPU performs better for higher energy particles (more hits = more work)
- Grouped work not as effective since regular GPU is already performant
  - also need to send extra information to GPU when work is grouped between events



Porting to Kokkos

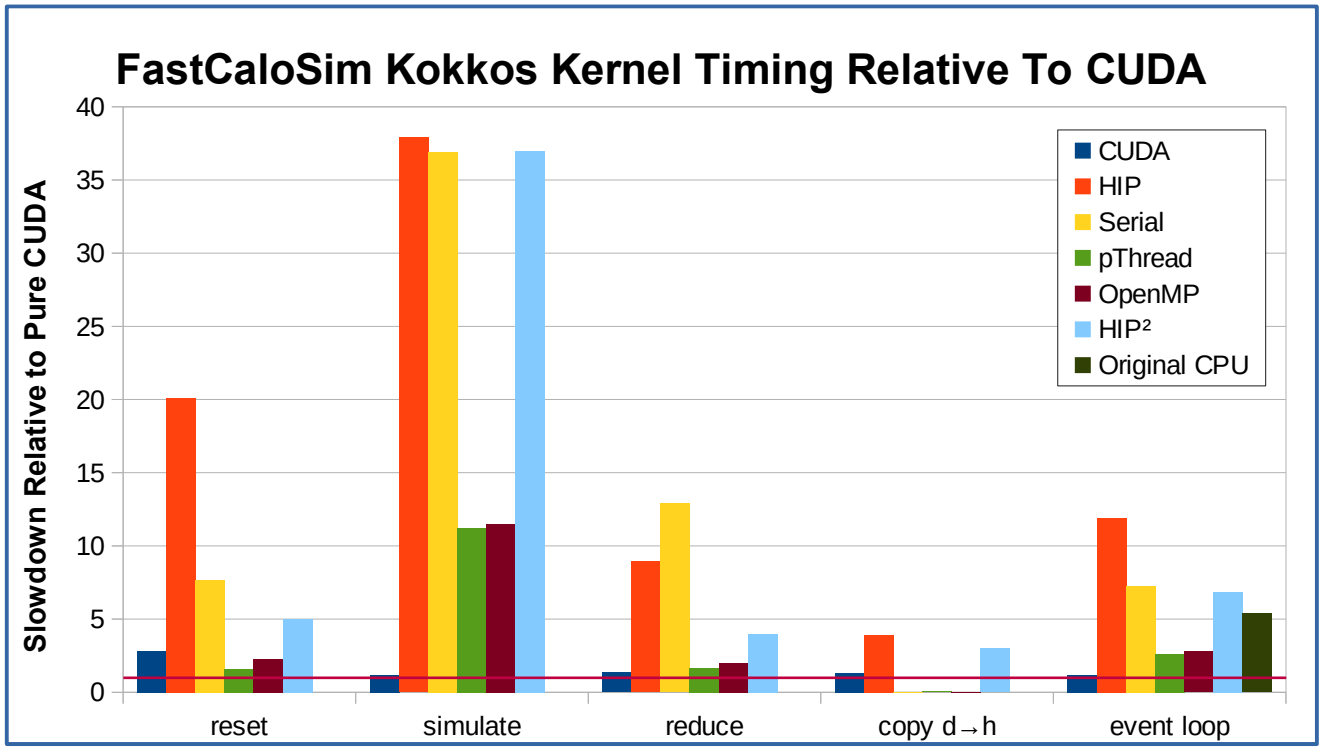
- Build infrastructure
  - Kokkos has decent CMake integration
  - requires separate binaries for each device backend (CUDA, HIP, Intel) or host parallel backend (pThread, OpenMP)
  - In theory you can run both device/host parallel backends in same code, but then you can't use the default execution space for your kernels: have to say which go where
  - cannot expose plain nvcc to Kokkos headers: need selection macros or different files
- Shared libraries not compatible with device symbol relocation
  - if you want shared libs, all symbols in a kernel must be visible to one compilation unit
    - wrap kernels in one file that does a bunch of `#include`
    - needed to do some function/file refactoring to make it all work
- CUDA backend of Kokkos interoperable with pure CUDA
  - can call CUDA functions from Kokkos kernels
  - makes incremental porting and validation much easier
- All offloaded data structures need to be converted to Kokkos Views
  - for full portability between backends

Porting to Kokkos: Data Structures and Kernels

- **`Kokkos::View<...>`** can either allocate host/device memory, or wrap existing pointers
  - makes incremental porting of cuMalloc'd memory easier
- Supports both row and column major ordering
- Jagged multidimensional arrays not well supported by Kokkos Views
  - `View<View<float>>` not meant for this
  - lots of extra boilerplate needed to make work
  - easier to flatten to 1D array, or pad to 2D
- Requires explicit Host ↔ Device memory migration
  - need to create Views on host to hold copied information
- Non-zero overhead to using Views
  - both in the extra steps for creating the host/device Views, and operations on them
- Kernels: While syntax is different from CUDA, concepts are the same
  - functions → lambdas
  - parallel_for, parallel_scan, reductions
    - some CUDA features not available in Kokkos (yet?)
  - atomics (but not between devices or host/device parallel execution spaces)

Kokkos: Performance

- Exercise various backends, compare to original CUDA
  - CUDA reference is NVidia 2080
  - HIP on AMD GPU (Vega56)
  - Serial: host serial via Kokkos
  - Intel X^eLP / X^eHP GPU via OpenMP target offload, but I can't show those results…
  - pThread / OpenMP best performance with ~15 threads/procs
  - **HIP²** is a pure HIP port, run on AMD GPU
- Kokkos/CUDA "simulate" kernel has similar performance as pure CUDA
- Kokkos does not handle GPU memory (re)initialization efficiently
- Kokkos increases kernel launch penalties
- AMD Vega56 has ***very*** large launch latencies
- HIP/AMD uses the CPU a lot more than CUDA when executing kernels on GPU
- Code was **ported** from CUDA, not rewritten
  - likely considerable room for optimization



kernel launch latencies / µs		Kokkos		
CPU Freq	CUDA	CUDA	HIP	HIP²
2200 MHz	5.6	9.4	152	88
3700 MHz	3.4	5.3	60	30

SYCL Build Infrastructure

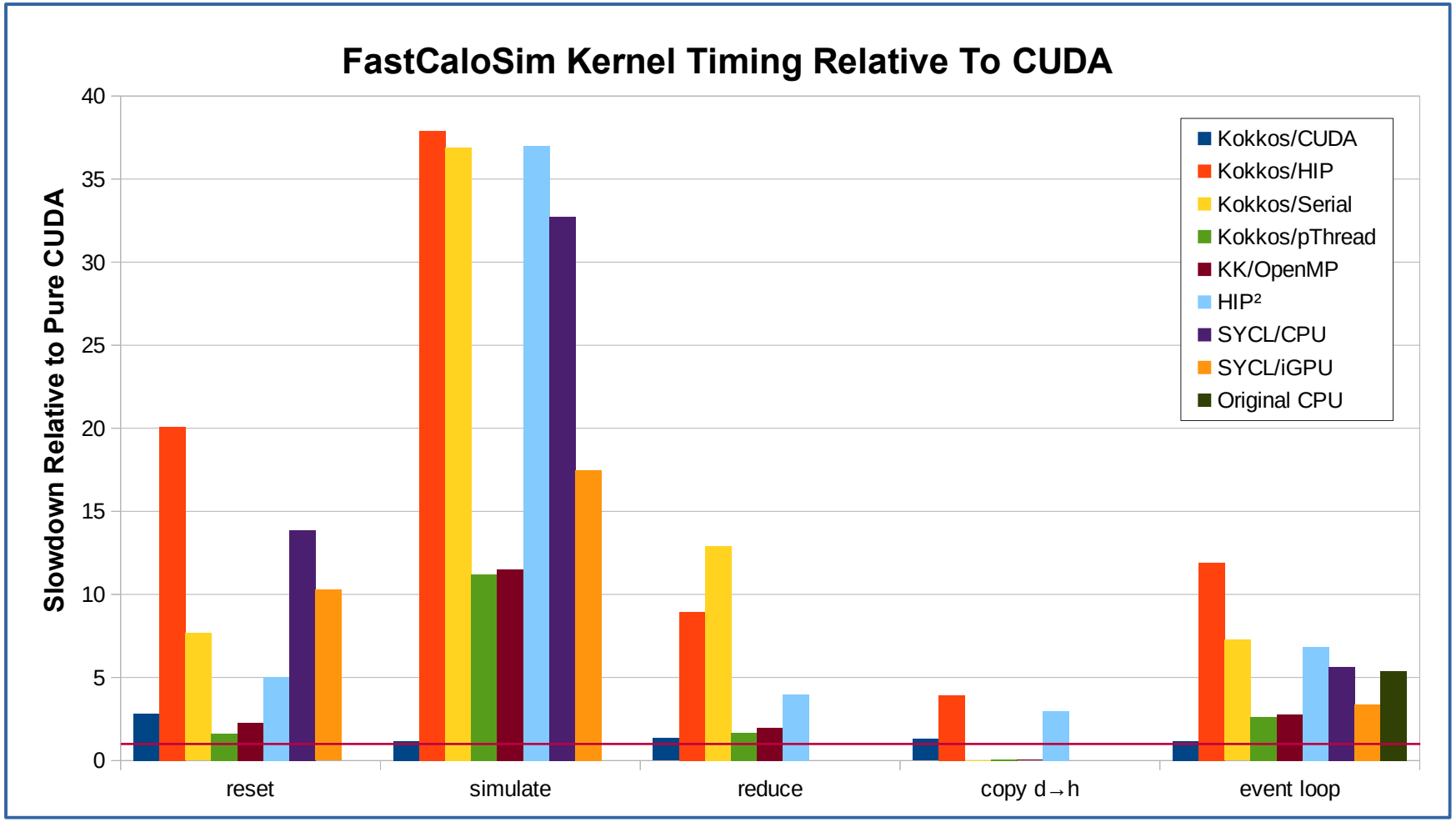
- Multiple different flavours of dpcpp/SYCL
  - Intel official “beta” releases (CPU, Intel iGPU, Intel FPGA)
  - Intel closed codedrops at Argonne for A21 development (CPU, Intel iGPU, Intel dGPU)
    - OpenCL and LevelZero backends
  - Intel/llvm git
    - CUDA backend available (selectable at compile time)
    - RNG issues: no oneMKL/cuRAND implementation
  - Codeplay (Intel, NVIDIA GPU)
  - hipSYCL (Intel, NVIDIA, AMD), triSYCL (PoCL, Xilinx FPGA)
- In theory SYCL is single source, compile once, run anywhere, select backend at runtime
  - in practice need to build with different compilers to target different hardware
  - maybe there will be convergence in the future
- Integrates well with CMake

SYCL: Performance

- Timing tests on an integrated Intel GPU (Iris Pro P580) w/ public dpcpp beta10 release

- SYCL data/buffers are normally automagically migrated between Host ↔ Device as needed, but with iGPU is all same RAM
  - uses USM
  - resetting data is not free – done with a kernel, ~180k launches

- Same code has been run on X^eLP and X^eHP discrete GPUs



Lessons Learned

- Build configuration requirements may be challenging
  - Kokkos shared libs *vs* relocatable device code: code reorganization
  - dpcpp changing (too?) rapidly, things that worked last week may not work today
- Separate binaries for different device backends
  - Kokkos explicitly, SYCL because you need different compiler flavours
  - major implications for production code distribution
- CUDA→ Portability Layer concepts translate well
  - Views / Buffers / USM memory management come with overhead / penalties
- Launch latencies for tiny kernels kills performance on all platforms
  - Portability layers make it worse
  - AMD is really bad. Will RDNA2 / CDNA2 / Instinct improve things?
- High performance single source (single core) CPU/GPU may be a pipe dream
- GPU very underutilized in FastCaloSim
  - grouping data between events helps: may require significant refactoring of frameworks
  - a single GPU can be shared between multiple processes

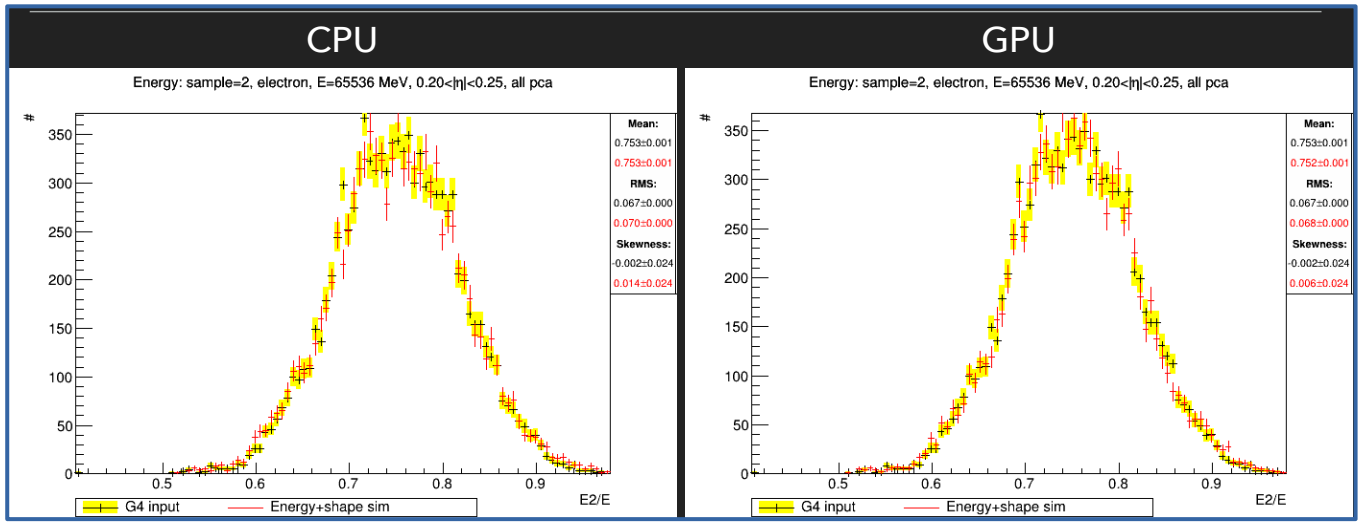
Next Steps

- Other Parallel Portability Layers:
  - OpenMP / OpenACC : very different pragma based directives
  - Alpaka : strong presence at CERN
  - Raja : will we learn anything that Kokkos didn't teach us?
- Other backends
  - SYCL w/ CUDA on NVidia : apples to apples comparison
    - Work started on cuRAND support for oneMKL
  - Intel discrete GPU (Arctic Sound/X^{e}HP and Ponte Vecchio/X^{e}HPC) via Kokkos and SYCL
    - we can already run FCS/SYCL on X^{e}LP and X^{e}HP nodes at Argonne JLSE
  - AMD RDNA2 / CDNA2
- Better understanding/evaluation/reporting of metrics
  - in coordination with other testbeds
- Update FastCaloSim to reflect what ATLAS is currently using
  - more realistic particle scenarios
  - integrate into ATLAS repositories

fin

Validation

- CUDA has a very good random number generator (cuRAND)
  - FCS needs *lots* of random numbers
    - 3 per hit x ~5k hits per event
  - much faster than generating on CPU
  - but can't do bitwise comparisons with CPU – only statistical
  - after looking at lots of histograms, results look statistically equivalent



- If we sacrifice speed, we can generate random number on CPU, and transfer them to GPU, using these for all calculations on GPU
  - compared the results of 62 million hits in the Electron 64 GeV run
  - found only 2 hits calculations that ended up in different calorimeter cell
    - slightly different float rounding policies on CPU/GPU
  - if we use double precision variables for certain calculations, difference vanishes

- Confident that GPU code does the same thing as CPU

Acknowledgments

- Really want to thank all the people who contributed to this project

- CUDA port:

  Zhihua Dong (BNL)
  Meifeng Lin (BNL)
  Kwangmin Yu (BNL)

- Kokkos port:

  Zhihua Dong (LBL)
  Charles Leggett (LBL)

- SYCL port:

  Charles Leggett (LBL)

  Vincent Pascuzzi (LBL)

- Physics Validation:

  Doug Benjamin (ANL)

  Tadej Novak (DESY)