U.S. DEPARTMENT OF ENERGY | Office of Science

High Energy Physics - Center for Computational Excellence

HEP-CCE

Portable Parallelization Strategies - Evaluation of Selected Use Cases

Haiwang Yu (BNL)

On behalf of the HEP-CCE PPS Team

IRIS-HEP Topical Meeting

Aug. 4, 2021

Argonne NATIONAL LABORATORY

BERKELEY LAB

Fermilab

Brookhaven National Laboratory

Outline

Three selected use cases focusing on the Kokkos potable solution:

- FastCaloSim (ATLAS parametrized detector simulation, speedup ATLAS simulation)
  - C. Leggett, HEP-CCE Meeting Nov. 2020
  - C. Leggett, vCHEP21
- Patatrack Pixel Tracking (CMS Pixel detector pattern recognition, CMS HLT for Run 3)
  - github
  - M. Kortelainen, HEP-CCE Meeting Nov. 2020
  - M. Kortelainen, vCHEP21
- WireCell Toolkit – gen module (DUNE LArTPC detector response simulation, speedup DUNE simulation)
  - github
  - H. Yu, vCHEP21

Kokkos Porting: Building

- Build infrastructure
  - Kokkos has good **CMake** integration
  - requires **separate binaries for each device backend** (CUDA, HIP, Intel) or host parallel (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

- 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 **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

Kokkos::View: Data Structures

- Kokkos::View<T> is an N-dimensional array of type T
  - **Reference-counted**, works similar to std::shared_ptr
  - Kokkos Views can either allocate host/device memory, or **wrap existing pointers**
    - makes incremental porting of cuMalloc memory easier
  - Layout can be controlled with template parameters
    - Different default layout for host and device backends
    - Possibility for custom layouts
  - **Template parameters to enable optimizations based on intent**
    - E.g., very easy to use CUDA texture access for random-access constant data
      - Could be useful for calibration data, not tested yet though
  - Jagged multidimensional arrays not well supported

- Generic way to construct a View on the optimal host memory space for a View of device memory space
  - CUDA device memory -> CUDA pinned host memory
  - Host memory -> host memory (**shortcut, no actual copy**)

- **By default, the View initializes the content**
  - On GPU memory runs a kernel with the default constructor
  - Can be very expensive if initialization is not needed (first access is write)

- Out of the box View does not use memory pool

Kokkos: Parallelism

- High-level API: parallel_for, parallel_reduce, parallel_scan
  - Can be nested (with some restrictions)

- Details of the iteration and operations are controlled with a policy
  - RangePolicy: 1-dimensional range, all elements are independent
  - MDRangePolicy: 1-6 dimensional range, all elements are independent
  - TeamPolicy: thread teams / hierarchical parallelism (more on next slide)
    - Corresponds to CUDA's grid of blocks of threads

- RangePolicy and MDRangePolicy are simple to use when hierarchical parallelism is not needed
  - Developer does not have to think about distribution of work to threads

Kokkos: Hierarchical Parallelism

- In CUDA one has a **grid of blocks of threads**
  - Threads of a block can synchronize (e.g. barrier) and have common scratch space (shared memory)
  - Blocks themselves are independent (no synchronization available)

- Kokkos supports this model via thread teams ("**league of teams of threads**")
  - Barrier synchronization, scratch space
  - Can do reduction over the threads of a team

- Number of threads in a team is not exactly portable
  - Serial backend must have exactly 1, pthread backend can use at most the number of CPU threads, CUDA backend has the same limits as CUDA itself (128/256 are typical)
  - Can be mostly mitigated by letting Kokkos decide with Kokkos::AUTO()

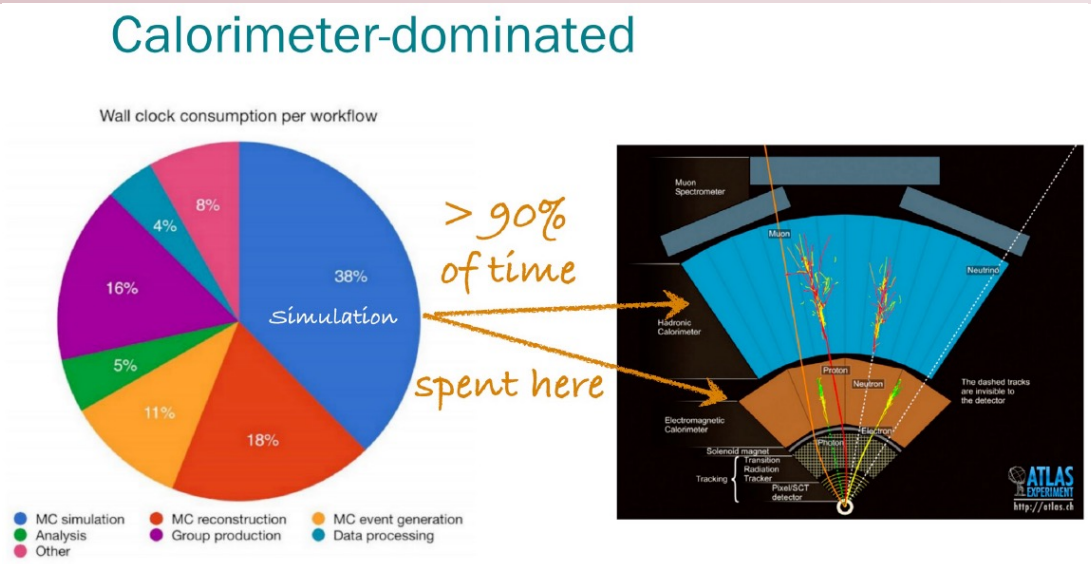
Policy	Description
TeamThreadRange	Used inside of a TeamPolicy kernel to perform nested parallel loops split over threads of a team.
TeamVectorRange	Used inside of a TeamPolicy kernel to perform nested parallel loops split over threads of a team and their vector lanes.
ThreadVectorRange	Used inside of a TeamPolicy kernel to perform nested parallel loops with vector lanes of a thread.

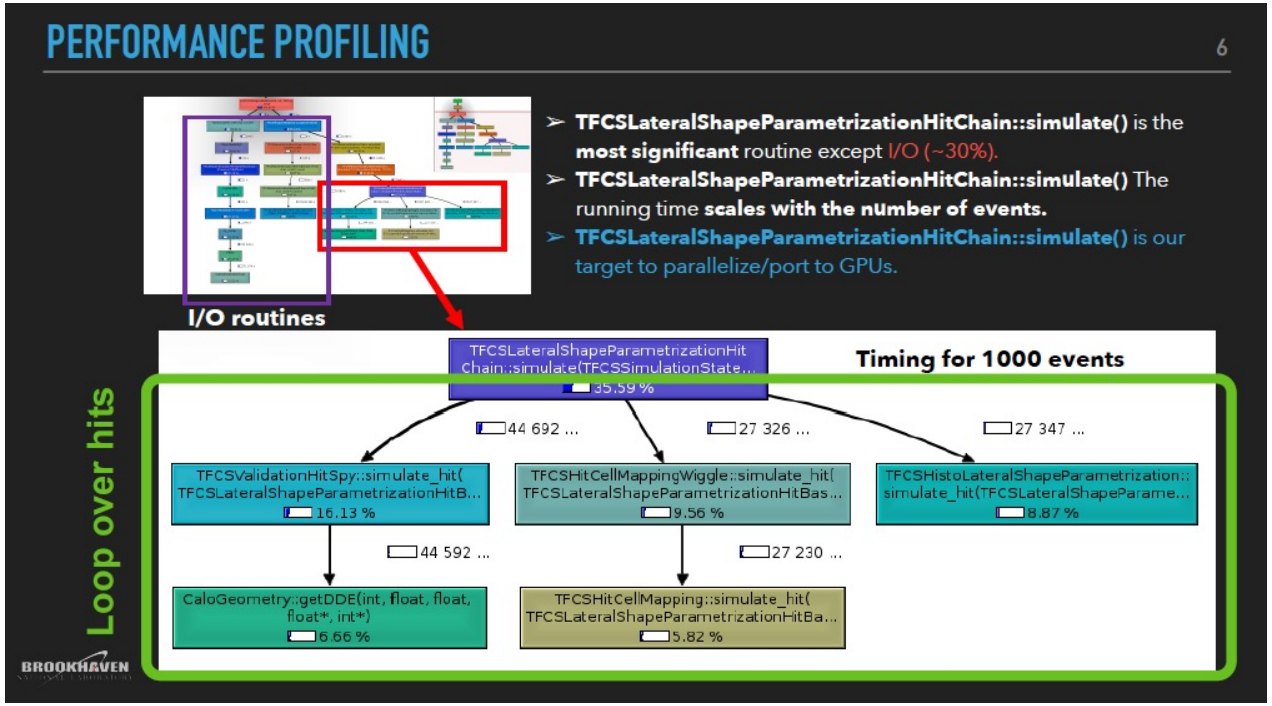
Ref: https://github.com/kokkos/kokkos/wiki/Execution-Policies

FastCaloSim: Intro

- ATLAS needs a very large amount of simulation
- A very large fraction of the simulation's computational budget is spent by the LAr Calorimeter
  - **Parametrized simulation can speed processing up enormously: FastCaloSim**
- FastCaloSim is small, self contained, few dependencies

- LAr Calorimeter has massive inherent parallelism – many independent cells and associated tasks.
- **Profiling studies identified likely hotspots that are parallelizable**
- **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:
    - memory initialization
    - main simulation
    - reduction
  - Ref. [C. Leggett, vCHEP21]





FastCaloSim Kokkos Porting: Kernel

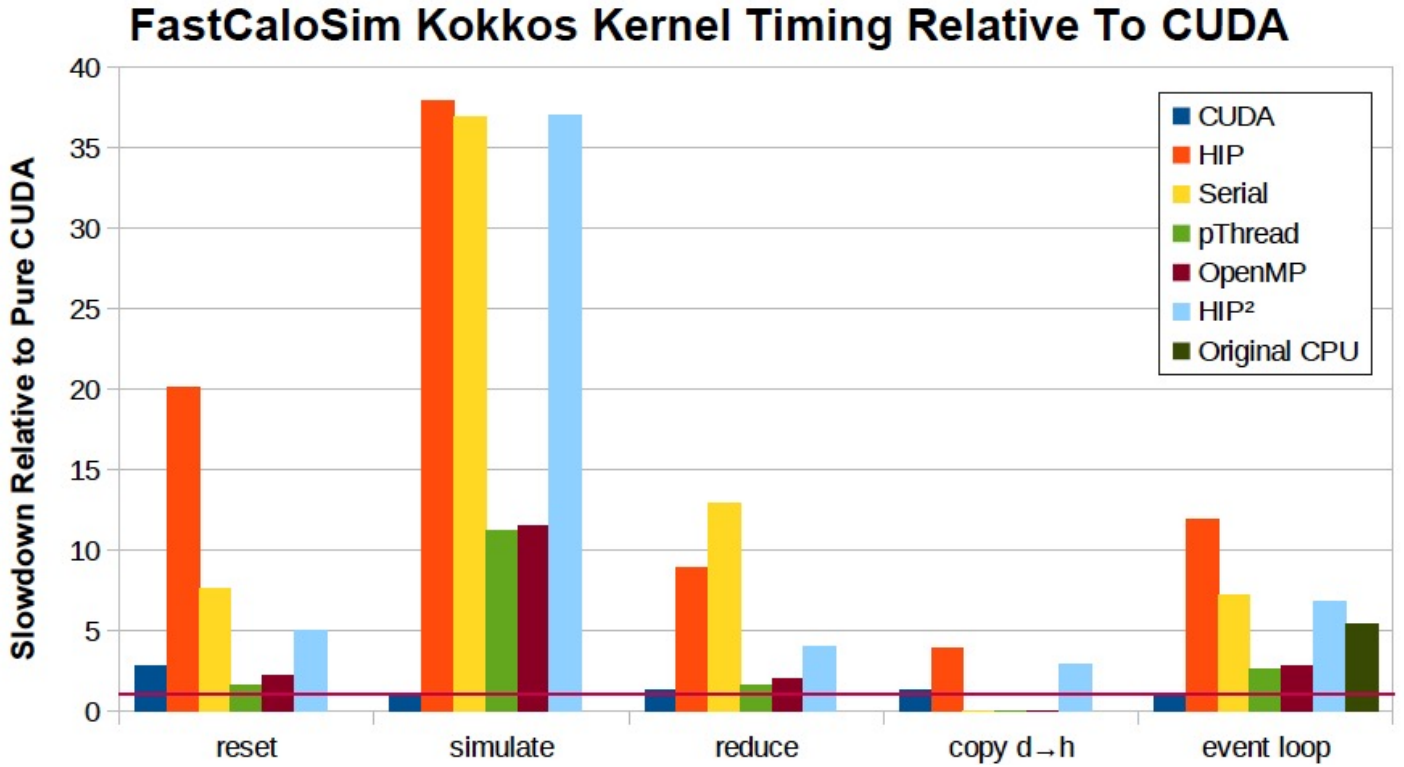
- While syntax is different from CUDA, concepts are the same
  - functions → lambdas
  - parallel_for, parallel_scan, reductions
  - atomics (but not between devices or host/device parallel execution spaces)

- Most FastCaloSim functions identical between CUDA / Kokkos
  - use a single file with #ifdef to select attributes to share as much code between version

```
#ifdef USE_KOKKOS
#  include <Kokkos_Core.hpp>
#  include <Kokkos_Random.hpp>
#  define __DEVICE__ KOKKOS_INLINE_FUNCTION
#else
#  define __DEVICE__ __device__
#endif
```

FastCaloSim Kokkos Porting: Performance

- Exercise various backends, compare to original CUDA
  - CUDA reference is NVidia 2080
  - HIP on AMD GPU (Vega56)
  - Intel XeLP GPU via OpenMP target offload, but I can't show that…
  - pThread / OpenMP best performance with ~15 threads/procs
  - HIP2 is a pure HIP port, run on AMD GPU
- **Kokkos does not handle GPU memory initialization efficiently**
- **Kokkos kernel launch penalties worse than CUDA**
- AMD Vega56 has horrible 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 optimization possible

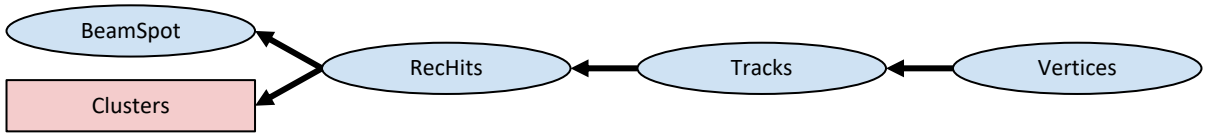


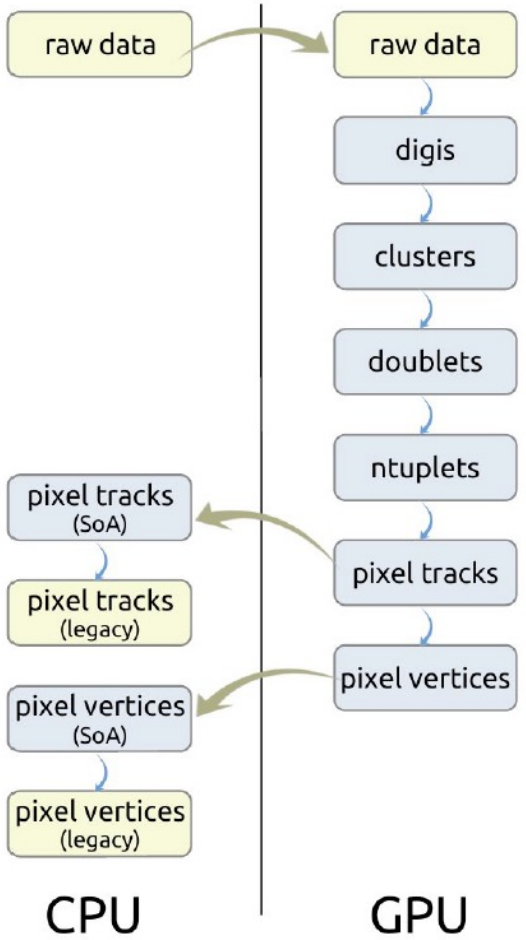
CPU Freq	CUDA	Kokkos		HIP²
		CUDA	HIP	
2200 MHz	5.6	9.4	152	88
3700 MHz	3.4	5.3	60	30

kernel launch latencies / μs

Patatrack: Intro

- Copy the raw data to the GPU (~250 kB/event)
- Run multiple kernels (39) to perform the various steps
- Copy only the final results back to the host
  - Optimized SoA format
  - ~4 MB/event for tracks, ~90 kB/event for vertices
  - Convert to legacy format if requested





- The program is in the limit of "many small GPU operations"
  - All overheads matter
  - At the peak throughput the CUDA runtime mutex is hammered at O(100 kHz)
- Key elements for performance
  - Process multiple events concurrently by using **CUDA streams**
  - Asynchronous execution: CPU does other work while the GPU is processing
    - "Continuation passing" with callback functions instead of blocking synchronization calls
- Caching allocator ("**memory pool**") to amortize the cost of CUDA memory allocation functions

- More information
  - arXiv:2008.13461 for the algorithms
  - arXiv:2004.04334 for the framework side

Patatrack Kokkos Porting: Performance (CPU)

Test	Throughput (events/s)
CPU version, 1 thread	**13.5 ± 0.2**
Kokkos version, Serial backend	**8.5 ± 0.2**
CPU version, 40 threads	539 ± 9
Kokkos with Threads backend, peak at 18 threads	28 ± 1

- On Cori GPU nodes at NERSC
- Pinned on one socket of Xeon Gold 6148 (“Skylake”)
  - 20 cores, 40 threads
- Filled the socket with CPU intensive work
- Repeated 8 times on random nodes, jobs run 5-10 minutes at a time
- Different parallelization strategies
  - “CPU version”: process events concurrently, one event per thread
  - “Kokkos Threads backend”: one event in flight, parallelize within that event

Patatrack Kokkos Porting: Performance (CPU) - Update

- General changes for better optimization by the compiler
  - Use KOKKOS_FORCEINLINE_FUNCTION
  - Use Restrict = Kokkos::MemoryTraits<Kokkos::Restrict>;
  - Pass Kokkos::View arguments as const references
- Merge kernels that were split when Kokkos did not have team-wide prefix scan (only kernel wide)
  - Allowed to allocate two Views from the scratch space (shared memory) instead of global memory
- Avoid real atomics in Serial code

```
[matti@zeus ~/pixeltrack-standalone]$ ./cudacompat
Processing 1000 events, of which 1 concurrently, with 1 threads.
Processed 1000 events in 3.873291e+01 seconds, throughput 25.8178 events/s.

[matti@zeus ~/pixeltrack-standalone]$ ./kokkos --serial
Processing 1000 events, of which 1 concurrently, with 1 threads.
Processed 1000 events in 3.930203e+01 seconds, throughput 25.444 events/s.
```

Patatrack Kokkos Porting: Performance (CUDA)

	Test	Throughput (events/s)
CUDA version	9 concurrent events and threads (peak)	1840 ± 20
	1 concurrent event	720 ± 20
	1 concurrent event, caching allocator disabled	**159 ± 1**
Kokkos version	**CUDA backend**	**115.7 ± 0.3**

- On Cori GPU nodes, with 1 NVIDIA V100 GPU, repeated 8 times
- No memory pool (not supported yet) or concurrent events (attempts so far lead to assert failures) in Kokkos version, so disable those in CUDA version for comparison

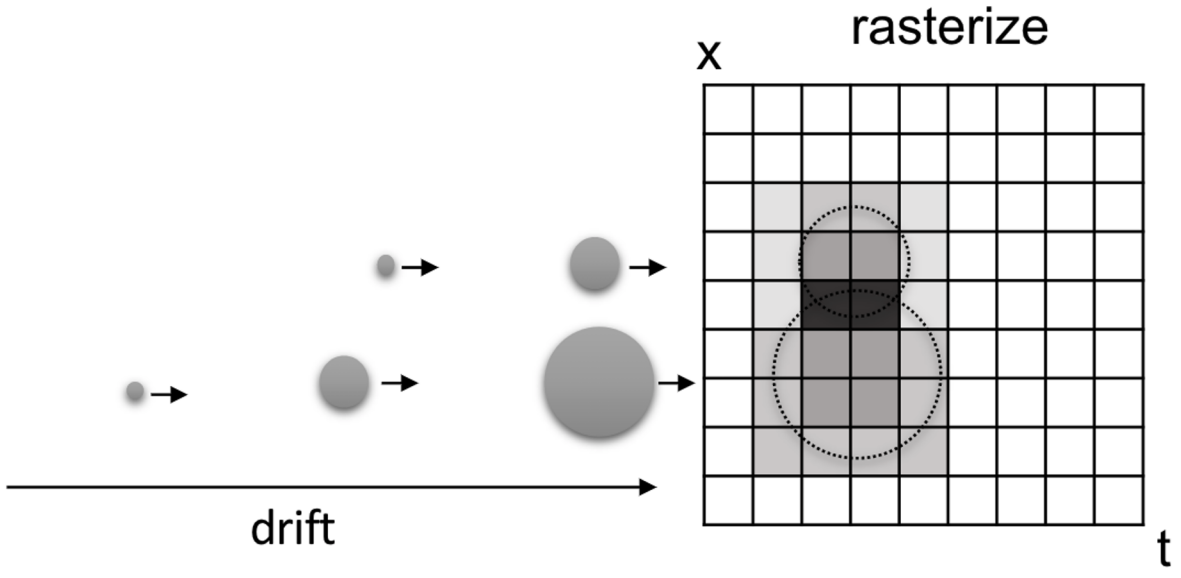
- [In-progress] Adapt the caching allocator and wrap pointers to Kokkos::Views

Wire-Cell Gen (Simulation) Major Steps

Three major steps of LArTPC simulation with Wire-Cell - a representative workflow

1. Rasterization: depositions ⟶ patches (small 2D array, ~20×20)
   - # depo ~100k for cosmic ray event
2. Scatter adding: patches ⟶ grid (large 2D array, ~10k×10k)
3. FFT: convolution with detector response

rasterization and scatter adding



Convolution theorem:
convolution in time/space domain

$$M(t,x) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} R(t-t', x-x')\cdot S(t',x')dt'dx' + N(t,x),$$

multiplication in frequency domain

$$S(t,x) \xrightarrow{FT} S(\omega_t,\omega_x),$$

$$M(\omega_t,\omega_x) = R(\omega_t,\omega_x)\cdot S(\omega_t,\omega_x),$$

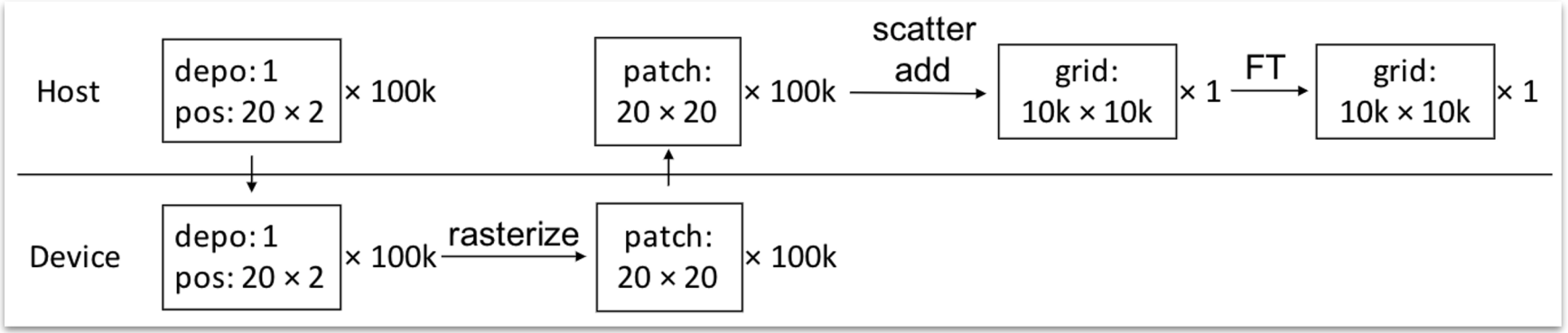
$$M(\omega_t,\omega_x) \xrightarrow{IFT} M(t,x).$$

Wire-Cell Gen Kokkos Porting: Plan

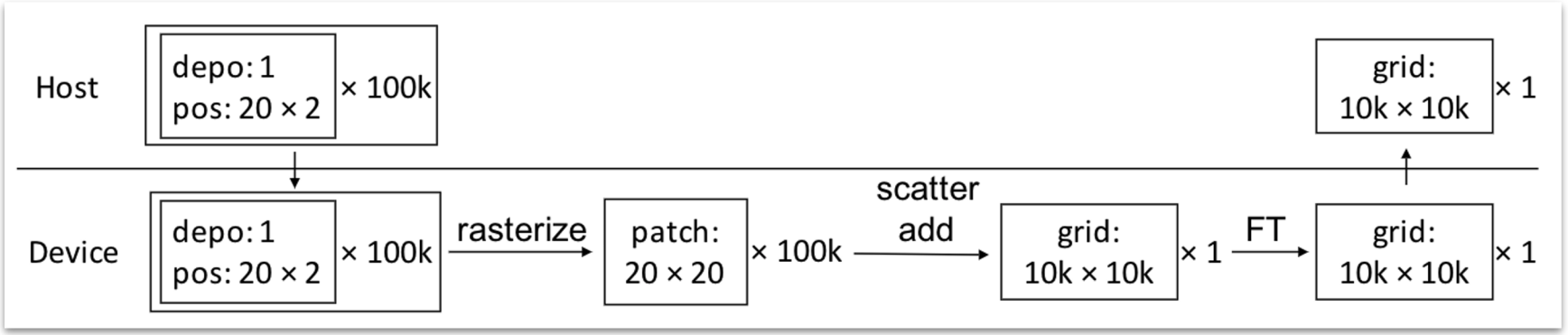
Two stage porting strategy

1. partial porting - port only step 1, rasterization
2. full porting
   a. more workloads for parallelization
   b. batched device-host data transfer

stage 1



stage 2



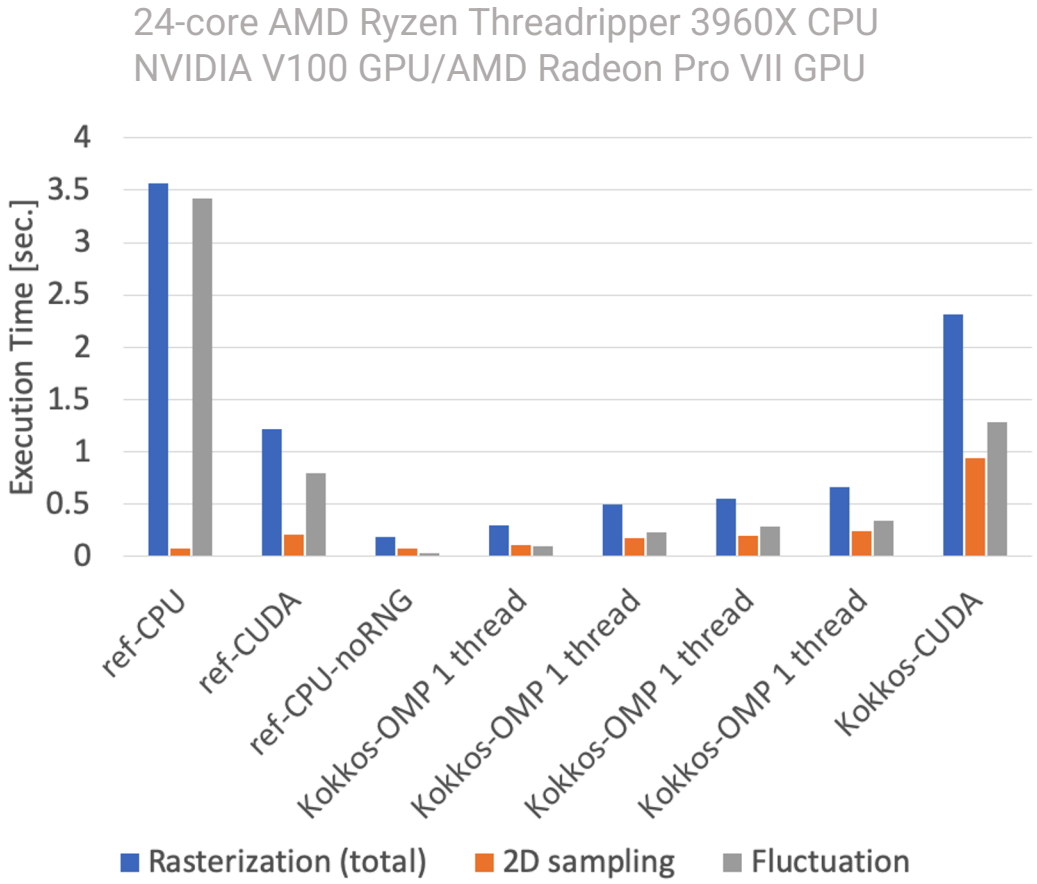
Wire-Cell Gen Kokkos Porting: Stage 1

Initial Kokkos porting follows original CUDA porting.

- no need for major refactoring
- # concurrent workloads is small ~400
- results were not ideal

Nsight timeline analysis:

1. in between kernel and API calls, Kokkos has extra CudaDeviceSynchronization and CudaStreamSynchronization
2. Kokkos parellel_reduce() kernels are almost 3 times slower than CUDA reduction kernels in this version
    - too small workload only one block



Wire-Cell Gen Kokkos Porting: Stage 2 Milestone

Boundle ~100k rasterizations together
(original total raserizations tasks become 2 parts)

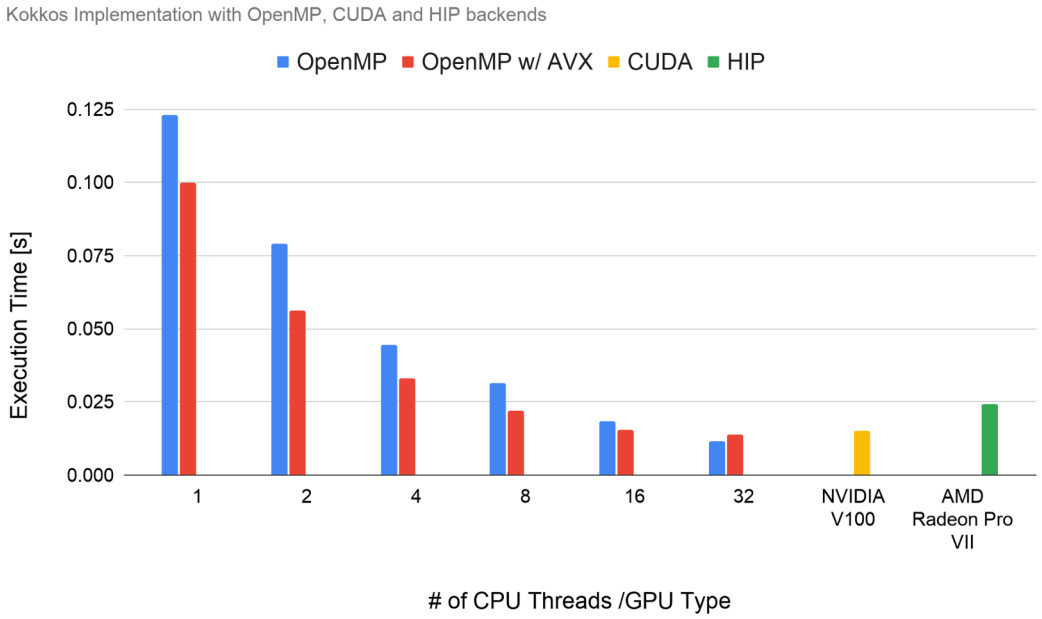
- set_sampling_pre()
  - Prepare for 2D sampling
  - Currently serial ~0.085s
  - Working on parallelizing it
  - Expect good improvement in performance

- set_sampling_bat()
  - Single kernel for 2D samplings and fluctuations.
  - Include host/device data transfer
  - Use Kokkos *TeamThreadsRange* for CUDA
  - Use Kokkos *ThreadVectorRange* to enable SIMD on OMP backend.

- CUDA backend ~10x better than before

Timing for set_sampling_bat()

24-core AMD Ryzen Threadripper 3960X CPU
NVIDIA V100 GPU/AMD Radeon Pro VII GPU



For GPUs, the actual kernel time are very small (<1ms)
Most time are on data transfers which will be absorbed in next step when scatter_add and FFTs parts are implemented on device with kokkos.

Wire-Cell Gen Kokkos Porting: Scatter Adding

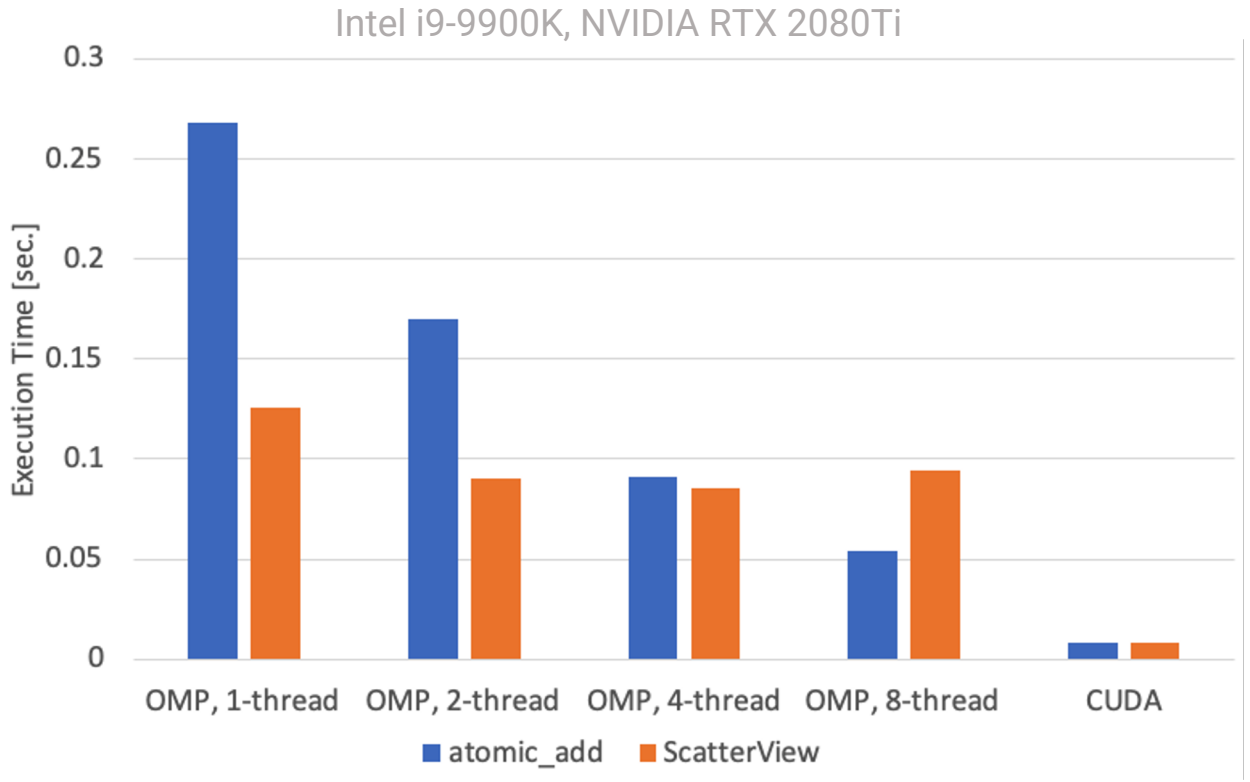
Initial attempts to do scatter adding with Kokkos

1. Kokkos::atomic_add
2. Kokkos::ScatterView

- atomic_add scales better with OMP threads
- ScatterView has better ST performance

- equal performance for CUDA

Unit test:
grid size: 1000 × 6000
patch size: 15 × 30
50k patches, avg. time for 10 executions



Wire-Cell Gen Kokkos Porting: FFT

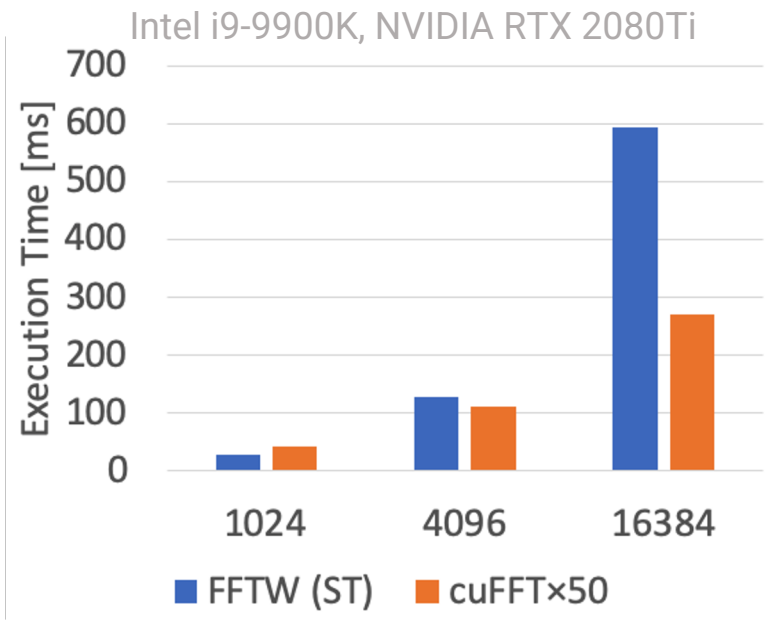
Kokkos does not have a native FFT implementation or official interface to the optimized vendor libraries (FFTW, cuFFT) ⟹ **Use wrappers**

- Thanks to the synergia group for the helpful advices

cuFFT with **cufftPlanMany** performs 20× - 100× faster than FFTW on the test platform.

- Most of the times we need to do batches of 1D FFTs
- Previous FFTW version perform each one sequentially

Kokkos wrapper for FFTW and cuFFT
Unit test: 1D FFT for 1024 arrays per operation
x-axis is length of array
using cufftPlanMany



Wire-Cell Gen Kokkos Porting: Full Stage 2 Prototype

Batched rasterization

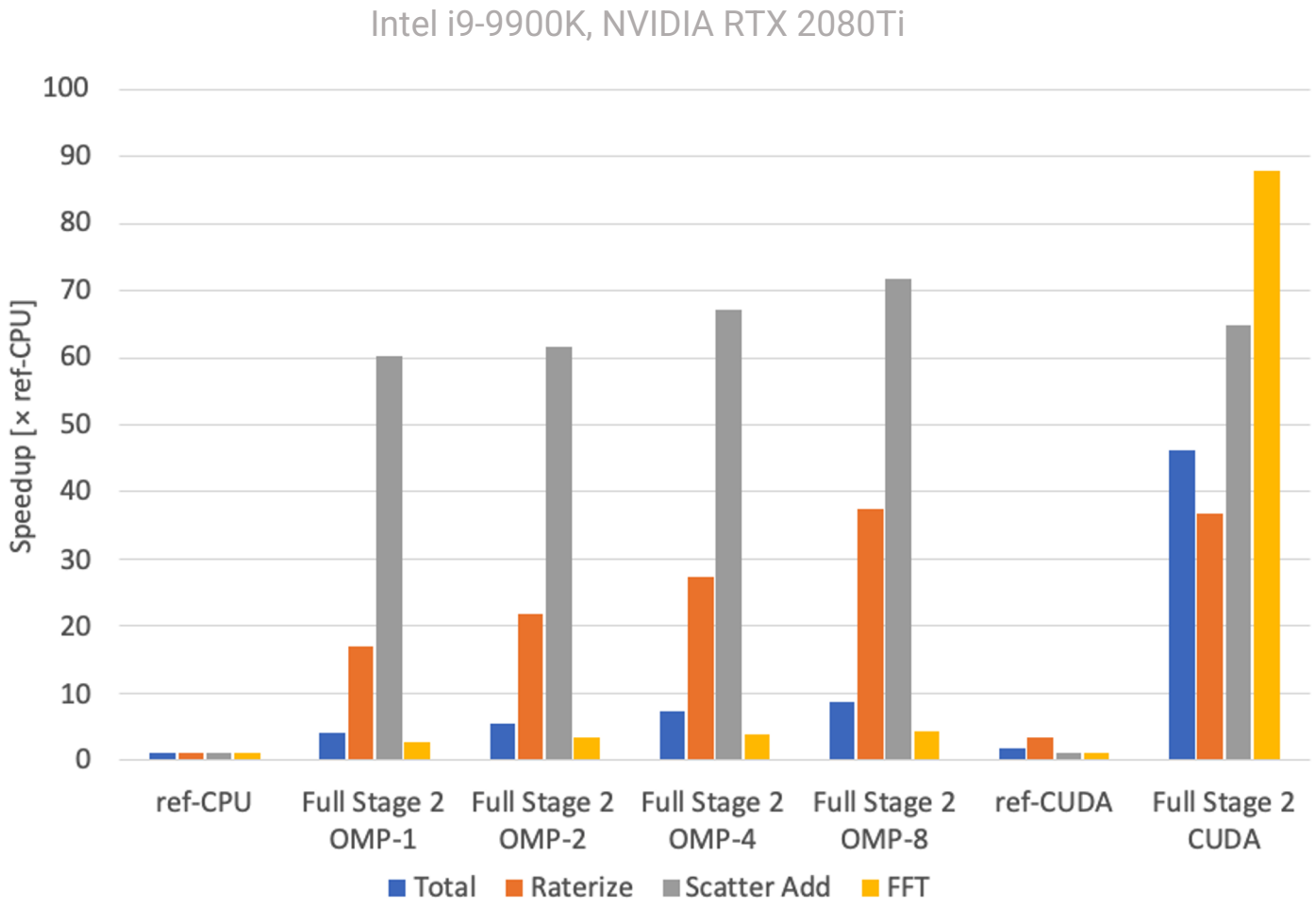
Scatter Adding: major refactoring

- sparse ⟶ dense (Kokkos::View)
- **60× speed up**
- no extra HtoD needed for FFT

cuFFT (cufftPlanMany): **88× (CUDA)**

Total speedup: **46×** (CUDA)

**Full Stage 2 porting is not completely finished yet, some tuning undergoing. But we expect the general trend should stay for the final version.*



Summary

- Focusing on one PPS, Kokkos: introduced three representative use cases (physics, algorithm, porting stage)

- Kokkos:
  - nice high-level API for basic operations
  - non-negligible overhead compared to native CUDA
    - optimization could mitigate this to some extent
  - some functionalities missing natively
    - memory pool (caching allocator)
    - FFT
    - [workaround] wrappers to other APIs essentially reduce some portability

- Improving the Kokkos porting; evaluate other use cases
- Evaluation of other PPS solutions on-going
  - SYCL/Alpaka/Parallel STL/etc.