Application of Portable Parallelization Strategies for GPUs on track reconstruction kernels

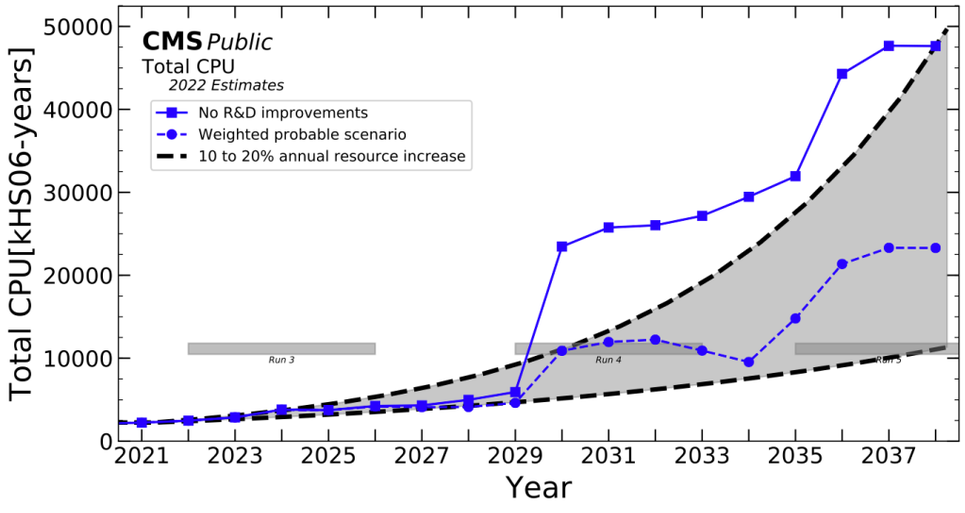
Martin Kwok, Matti Kortelainen, Giuseppe Cerati, Alexei Strelchenko, Oliver Gutsche(Fermilab)

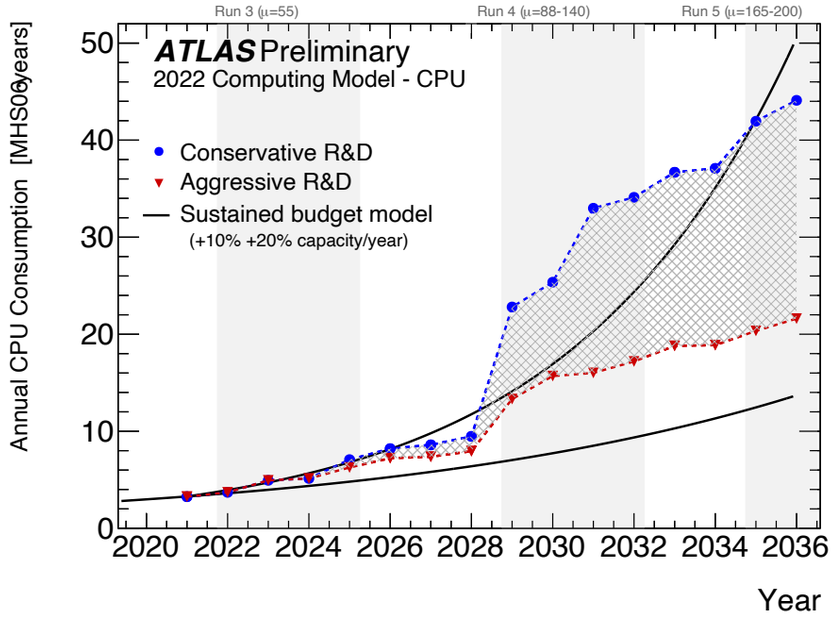
ACAT22
27 Oct, 2022

HEP-CCE

Performance portability

- Heterogenous computing is one of the key to meet the HL-LHC computing challenge
- Challenges of HEP computing:
  - Hundreds of computing sites (grid clusters + HPC systems + clouds)
  - Hundreds of C++ kernels (several million line of code, no hot-spots)
  - Hundreds of data objects (dynamic, polymorphic)
  - Hundreds of non-professional developers (domain experts)
- Portability:
  - Support multiple accelerator platforms with minimal changes to code base
- Performance portability:
  - Efficient use of CPU and GPU





Portability: HPC landscape

- Accelerator architectures are proliferating
  - Main GPU manufacturers: NVidia, Intel, AMD
  - FPGA is possible, but is used less in scientific computing
- Up coming flagship HPC systems has a mixture of architectures and GPU vendors
  - CPU+GPU is common NOW, but future system could be different
    - Shared physical memory between CPU and GPU
    - ARM, FPGA
  - Ability to use the GPU resources may be a prerequisite to the access of these machines

Perlmutter NERSC, 2020
AMD CPU, Nvidia Tesla GPU

Frontier ORNL, 2021
AMD CPU, AMD GPU, 1.5 ExaFlop

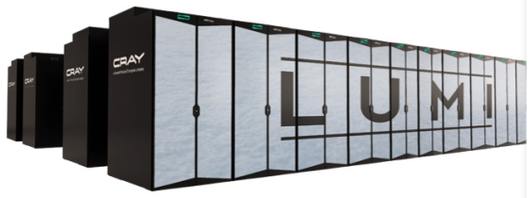
Leonardo, Cineca, 2021
Intel CPU, NVIDIA GPU, 200+PFlops

Alps, CSCS, 2023
NVIDIA Arm CPU+GPU

Aurora Argonne, 2022
Intel CPU, Intel Xe GPU, > 1 ExaFlop

El Capitan LLNL, 2023
AMD CPU, AMD GPU, > 1.5 ExaFlop

LUMI, CSC, 2021
AMD CPU, AMD GPU, 550 PFlops



Fermilab

Portability: Software landscape

HEP-CCE

- Rapidly changing ~O(month) portability solutions
  - New features/compiler supports/New backend
- Different approaches:
  - Compiler pragma-based approach
  - Libraries
  - Language extension
- HEP-CCE: Joint effort of major U.S. National labs involved in HEP
  - Investigate different portability solutions in HEP context

Focus of today - with an example test bed application

Software →

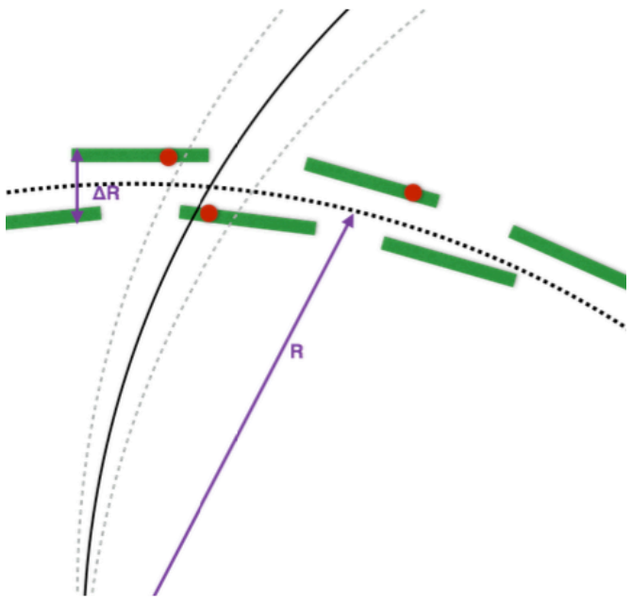
Hardware ↓

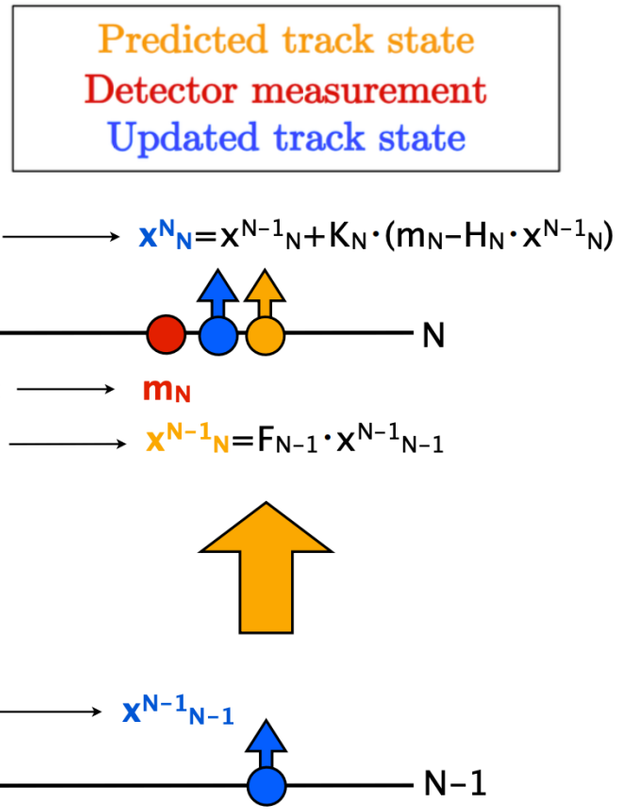
	CUDA	HIP	OpenMP Offload	Kokkos	dpc++ / SYCL	alpaka	std::par
NVidia GPU					*codeplay and intel/llvm*		*nvc++*
AMD GPU				*feature complete for select GPUs*	*via hipSYCL and intel/llvm*		
Intel GPU		*HIPLZ: early prototype*		*native and via OpenMP target offload*		*prototype*	*oneAPI::dpl*
multicore CPU							*g++ & tbb*
FPGA						*via SYCL*	

Fermilab

P2R introduction

- Track reconstruction is one of the most computational intensive task in collider experiments such as the LHC at CERN
- P2R is a standalone mini-app. to perform core math of parallelized track reconstruction
  - Build tracks in radial direction from detector hits (propagation +Kalman Update)
  - Lightweight kernel extracted from a more realistic application (mkFit, vectorized CPU track fitting)
  - Together with P2Z (a sister project) forms the backbone of track fitting kernels





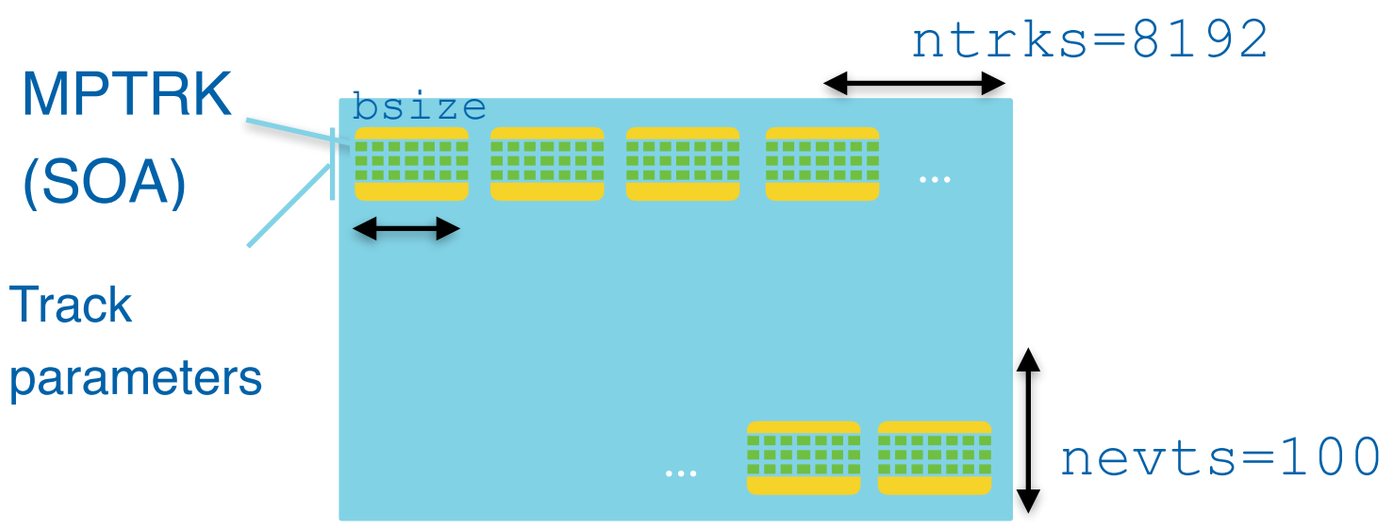
mkFit: https://arxiv.org/abs/2006.00071

p2r: https://github.com/cerati/p2r-tests

p2z:https://github.com/cerati/p2z-tests

P2R program overview

- Simplified program workflow:
  - Fixed set of track parameters
  - Fixed number of events (`nevts`)
  - Fixed number of tracks in each event (`ntrks`)
  - Single GPU kernel:
    - Prepare data on CPU
    - Transfer to GPU compute
    - Transfer track data back to CPU
- P2R uses Array-Of-Structure-Of-Array (AOSOA) as the main data structure
  - Total work of `ntrks` x `nevts`, tracks in an event are grouped into batch of `bsize`
  - Batch of tracks are put into the same data structure (MPTRK)



Fermilab

Alpaka programming model

https://github.com/alpaka-group/alpaka

- Single-source, header only C++ library
  - No additional runtime dependency introduced
- Initially started out as a thin abstraction layer over CUDA
  - API level similar to CUDA
  - Added an abstraction layer for portability (work division, memory operations etc.)
- Kernels are templated with an accelerator
  - Easy switching

Alpaka accelerator

```
// Example: CPU accelerator
using Acc = acc::AccCpuOmp2Blocks<Dim, Idx>;

// Example: CUDA GPU accelerator
using Acc = acc::AccGpuCudaRt<Dim, Idx>;

// Example: HIP GPU accelerator
using Acc = acc::AccGpuHipRt<Dim, Idx>;
```

Example Memory operations

```
// Allocate buffers
auto bufCpu = mem::buf::alloc<float, Idx>(devCpu, extent);
auto bufGpu = mem::buf::alloc<float, Idx>(devGpu, extent);

/* Initialization … */

// Copy buffer from CPU to GPU - destination comes first
mem::view::copy(gpuQueue, bufGpu, bufCpu, extent);

// Execute GPU kernel
queue::enqueue(gpuQueue, someKernelTask);

// Copy results back to CPU and wait for completion
mem::view::copy(gpuQueue, bufCpu, bufGpu, extent);
```

Alpaka kernel

```
struct HelloWorldKernel {

    template <typename Acc>
    ALPAKA_FN_ACC void operator()(Acc const & acc) const {

        using namespace alpaka;

        uint32_t threadIdx = idx::getIdx<Grid, Threads>(acc)[0];

        printf("Hello, World from alpaka thread %u!\n", threadIdx);
    }
};
```

Kokkos programming model

https://github.com/kokkos/kokkos

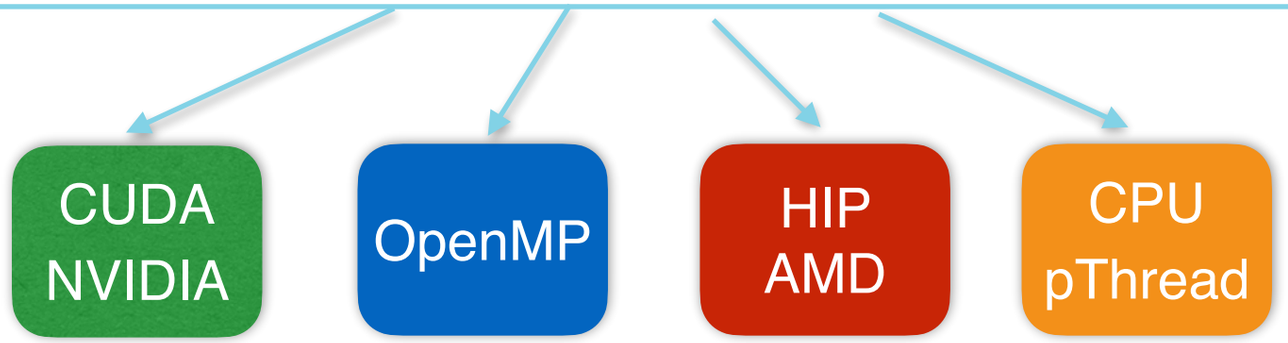
- Single source C++ template library
- Aims to be descriptive, not prescriptive
  - Developers express algorithm in general parallel programming *concepts*
  - Kokkos handles mapping to hardware
  - Abstraction layer provides handles for efficient data layout for both GPU/CPU
- No explicit mapping of loop iterations to threads
  - Could map to hardware that is not a close match of GPU-centric model

Pattern: nature of work
Execution Policy: How and where computations are executed
Body: Unit of work

Kokkos Abstraction

```
Pattern                    Policy
for (element = 0; element < numElements; ++element) {
  total = 0;                                          Body
  for (qp = 0; qp < numQPs; ++qp) {
    total += dot(left[element][qp], right[element][qp]);
  }
  elementValues[element] = total;
}
```

Parallel execution backends



Fermilab

Kokkos programming model

https://github.com/kokkos/kokkos

- Single source C++ template library
- Aims to be descriptive, not prescriptive
  - Developers express algorithm in general parallel programming *concepts*
  - Kokkos handles mapping to hardware
  - Abstraction layer provides handles for efficient data layout for both GPU/CPU
- No explicit mapping of loop iterations to threads
  - Could map to hardware that is not a close match of GPU-centric model

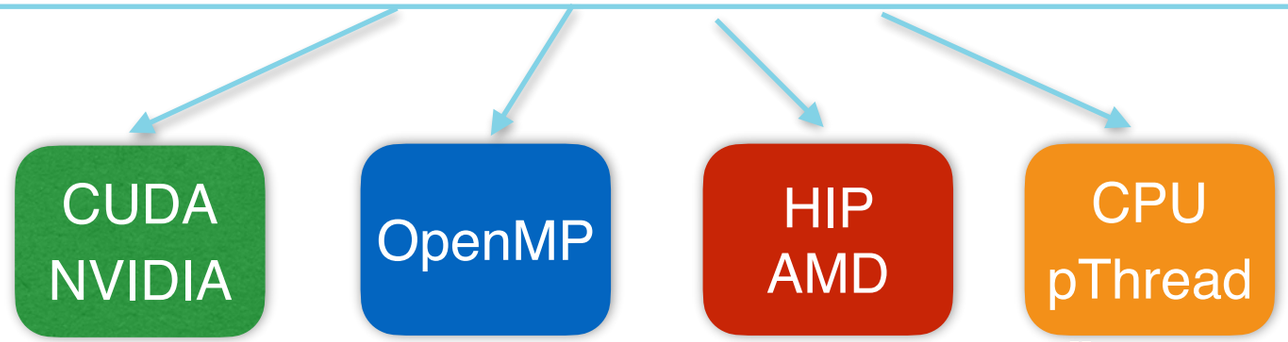
Kokkos Abstraction

Pattern: nature of work
Execution Policy: How and where computations are executed
Body: Unit of work

Custom

```
parallel_for("Label",
  RangePolicy< ExecutionSpace >(0,numberOfIntervals),
  [=] (const int64_t i) {
    /* ... body ... */
  });
```

Parallel execution backends

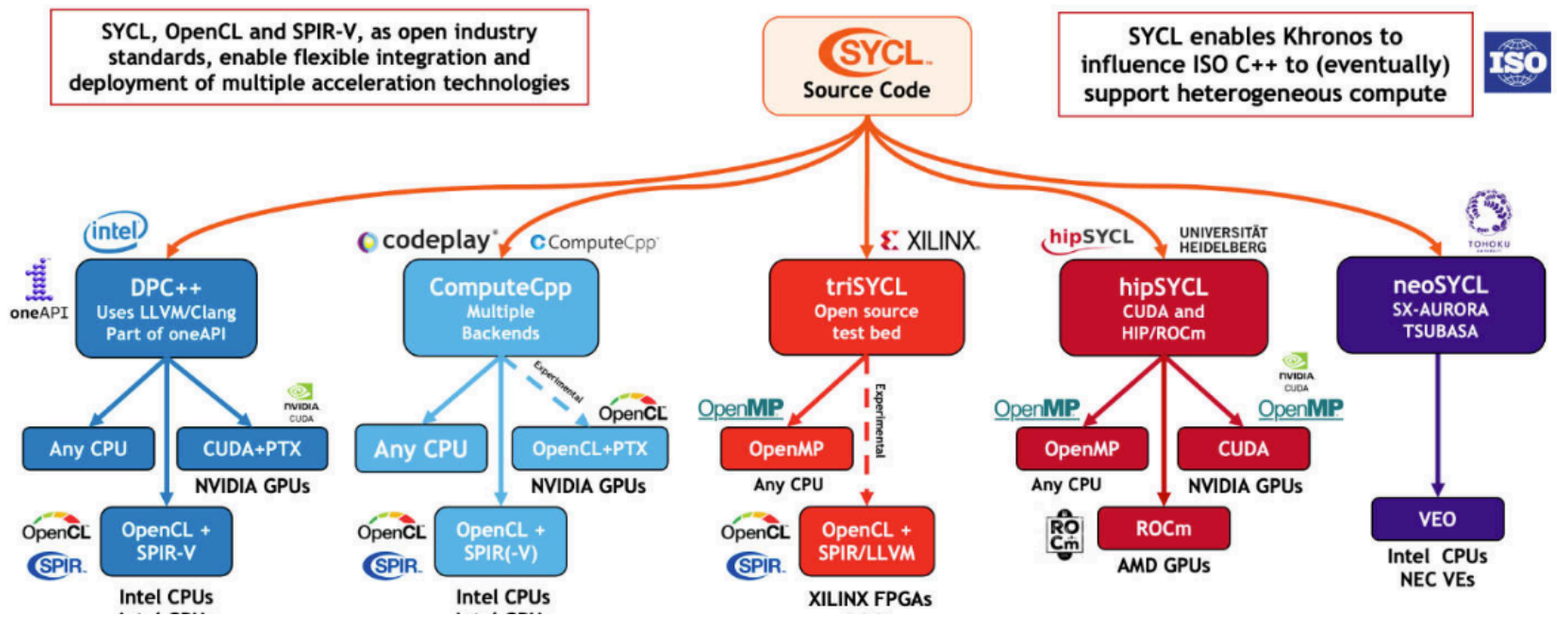


Fermilab

SYCL

- SYCL is a *specification* of single-source C++ programming model for heterogeneous computing
  - Different compilers/libraries implement the specification
- Data Parallel C++ (Part of Intel's OneAPI)
  - Implements SYCL standard + extension features
  - Support Intel's hardware (CPU/GPU/FPGA) + Nvidia/AMD GPUs
    - Open source, maintained by Intel in an branch of LLVM
    - AMD GPUs supported via hipSYCL(Heidelberg U) as well
- Compilers are rapidly evolving
- Both Alpaka/Kokkos are developing SYCL backend to support intel hardware
  - Kokkos is almost "feature complete"
  - Alpaka has experimental support since v0.9

Different implementations of SYCL



Fermilab

std::par

- Standard parallelization since C++17
  - Express parallel algorithm with standard language
- Plain C++ code
- Current limitations:
  - Memory operation via unified shared memory
  - No async operations
  - Cannot specify kernel launch parameters
- Compiler supports:
  - NVIDIA GPU via closed source compiler(nvc++) for NVIDIA GPUs
  - Intel GPU support requires adding oneAPI libraries(code change)
  - Still in early development

C++17

Parallel algorithms:

`std::execution`

`std::for_each`

C++20

Concurrency features:

`std::atomic<T>`

`std::atomic_ref<T>`

C++23/Beyond

Data structure:

`std::mdspan/mdarray`

Range-based parallel algorithm

Support for async. Execution

Porting experience

- Started with vectorized CPU(TBB) implementation
- Convert to CUDA
  - Profiling to define operating parameters
- Convert to portability solution
  - Core kernel code largely remains the same
  - Change of API: data handling, kernel launching
- Compilation
  - Configuration (SW stack) to compile for different back ends
  - Profiling to understand implementation

	CUDA	HIP	OpenMP Offload	Kokkos	dpc++ / SYCL	alpaka	std::par
NVidia GPU					*codeplay and intel/llvm*		*nvc++*
AMD GPU				*feature complete for select GPUs*	*via hipSYCL and intel/llvm*		
Intel GPU		*HIPLZ: early prototype*		*native and via OpenMP target offload*		*prototype*	*oneAPI::dpl*
multicore CPU							*g++ & tbb*
FPGA						*via SYCL*	

Porting experience

- Started with vectorized CPU(TBB) implementation
- Convert to CUDA
  - Profiling to define operating parameters
- Convert to portability solution
  - Core kernel code largely remains the same
  - Change of API: data handling, kernel launching
- Compilation
  - Configuration (SW stack) to compile for different back ends
  - Profiling to understand implementation

More time consuming to understand if we have converted optimally

	CUDA	HIP	OpenMP Offload	Kokkos	dpc++ / SYCL	alpaka	std::par
NVidia GPU					*codeplay and intel/llvm*		*nvc++*
AMD GPU				*feature complete for select GPUs*	*via hipSYCL and intel/llvm*		
Intel GPU		*HIPLZ: early prototype*		*native and via OpenMP target offload*		*prototype*	*oneAPI::dpl*
multicore CPU							*g++ & tbb*
FPGA						*via SYCL*	

Porting experience

- Started with vectorized CPU(TBB) implementation
- Convert to CUDA
  - Profiling to define operating parameters
- Convert to portability solution
  - Core kernel code largely remains the same
  - Change of API: data handling, kernel launching
- Compilation
  - Configuration (SW stack) to compile for different back ends
  - Profiling to understand implementation

Less well supported for AMD/Intel backends - often requires developer's support

	CUDA	HIP	OpenMP Offload	Kokkos	dpc++ / SYCL	alpaka	std::par
NVidia GPU					*codeplay and intel/llvm*		*nvc++*
AMD GPU				*feature complete for select GPUs*	*via hipSYCL and intel/llvm*		
Intel GPU		*HIPLZ: early prototype*		*native and via OpenMP target offload*		*prototype*	*oneAPI::dpl*
multicore CPU							*g++ & tbb*
FPGA						*via SYCL*	

Fermilab

Measurement on JLSE

- JLSE = Joint Laboratory for System Evaluation (JLSE)
  - HPC Testbed system hosted at Argonne National Lab
- Measured repeated 10 times
  - Does not include time for data-transfer (~3x kernel time on a A100 GPU)
- All versions compiled with the same p2r parameters
  - Perform computation on ~800k tracks, repeated 5 times
- Showing results on A-100/MI-100 GPUs:

		Peak FP64	Peak Memory	N cores
AMD	*MI-100*	*11.5 TFLOPS*	*1.2 TB/s*	*7680*
NVIDIA	*A100*	*9.7 TFLOPS*	*1.9 TB/s*	*6912*

JLSE hardware

NVIDIA GPU (A100)

Gigabyte G242-Z11
AMD 7532 32c 2.4Ghz
DDR4-3200 256GB (8x32G DIMMs) RAM
1x Nvidia A100 40GB PCIe 4.0
Mellanox ConnectX-6 EDR
Intel P4510 2TB NVMe

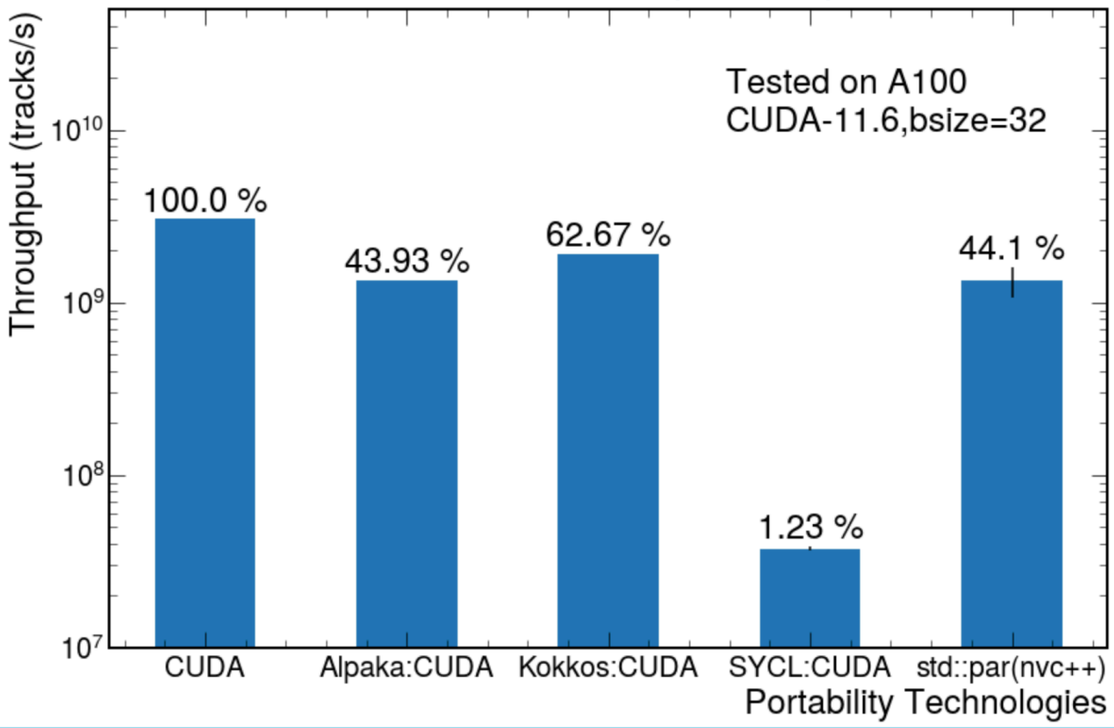
AMD GPU (MI-100)

- 2x AMD EPYC 7543 32c (Milan)
- 4x AMD MI100 32GB GPUs
- Infinity Fabric
- 512GB DDR4-3200

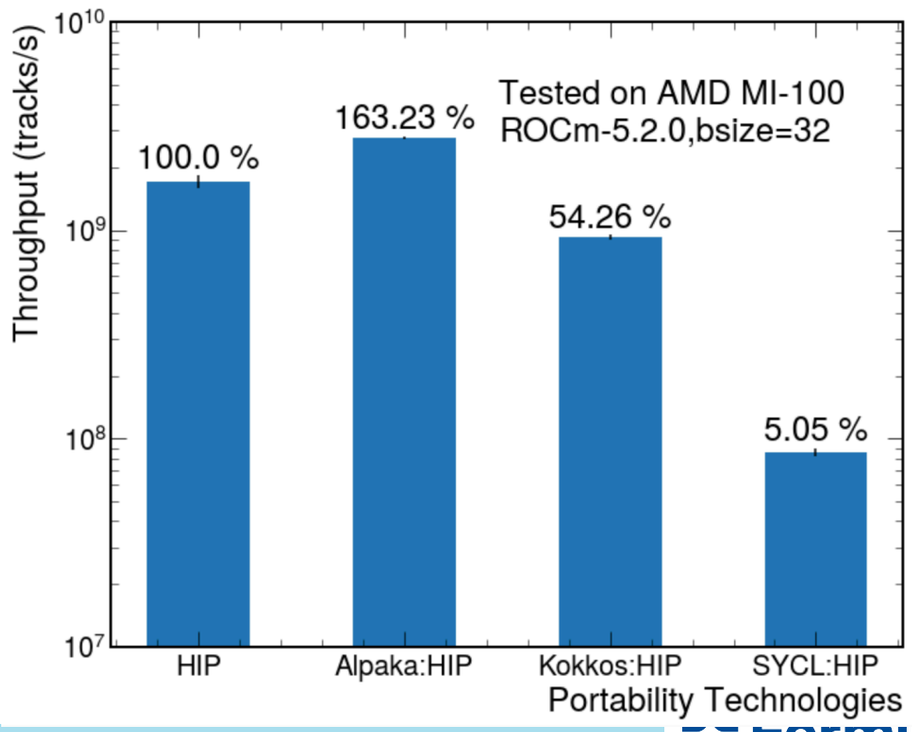
Results

- Result obtained with same code-based and compared with platform-native implementation
- Observed a large variation of slow-down
  - "Out-of-the-box" performance [requires detailed studies]
    - Alpaka/SYCL's result is slower than expectation in A100 (contrary to the experience with other CCE codes)
    - Would be good to understand attribution of Kokkos's overhead & Alpaka::HIP result
- Was able to run on Intel GPUs as well
  - Kokkos, SYCL, std::par (adding one::dpl libraries)
  - Alpaka::SYCL backend is being developed
  - Currently under NDA

NVIDIA GPU (A100)



AMD GPU (MI-100)



Summary and outlook

- Early round of results using JLSE
  - latest versions of Kokkos/Alpaka/Clang
  - Understanding the results
- Further studies:
  - CPU multicore measurements:
    - Efficient CPU AND NVidia GPU backend will be very relevant in near terms
  - Effect portability layers on memory transfer
- Longer term goal:
  - Contribute towards HEP-CCE final report
  - “Best” solution will probably depend on application/situation
    - Platforms to run on
    - Kernel bottlenecks

Acknowledgement:
We thank the Joint Laboratory for System Evaluation (JLSE)
for providing the resources for the performance measurements
used in this work.