ACAT2022, 23–28 Oct 2022

Portable Programming Model Exploration for LArTPC Simulation: OpenMP vs. SYCL

Zhihua Dong, Meifeng Lin, Vincent Pascuzzi, Brett Viren, Tianle Wang, Haiwang Yu

Brookhaven National Laboratory

Kyle Knoepfel

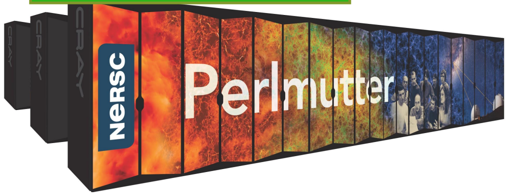
Fermilab

For HEP-CCE

Motivation

- Current and future HPC systems increasingly feature (different kinds of) compute accelerators (GPUs, FPGAs, etc.)
- There are now three major GPU vendors: NVIDIA, Intel and AMD. Hardware architectures are similar, but the native programming models supported are different:
  - NVIDIA - CUDA
  - Intel - SYCL
  - AMD - ROCM/HIP
- Future experiments (HL-LHC, DUNE, …) anticipate an **order of magnitude higher** compute and data processing needs.
- Most experimental HEP codes do not support GPU computing. CPU-only processing model may not be sufficient.
- Can we rewrite the CPU codes to be **portable** across different GPU architectures?

NVIDIA GPUs



Perlmutter (NERSC, US)

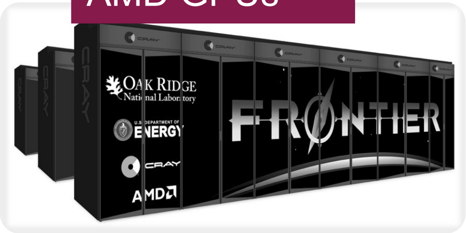
AMD GPUs

LUMI (CSC, Europe)

Intel GPUs

Aurora (ALCF, US, upcoming)

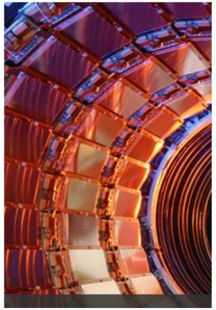
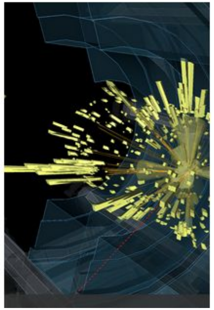
AMD GPUs



Frontier (OLCF, US)

HEP-CCE and Portable Parallelization Strategies

- **Kokkos:** a C++ abstraction layer (library) that supports parallel execution for different **host** and **accelerator** architectures.
- **SYCL:** a **specification** for a cross-platform C++ abstraction layer.
- **OpenMP/OpenACC**: Directive-based programming models for different **host** and **accelerator** architectures
- **Alpaka:** C++ abstraction layer similar to Kokkos
- **std::par:** language-based parallelism from C++ Standard
- **HIP:** originally an abstraction layer for CUDA and ROCM. Being extended to also support OneAPI.

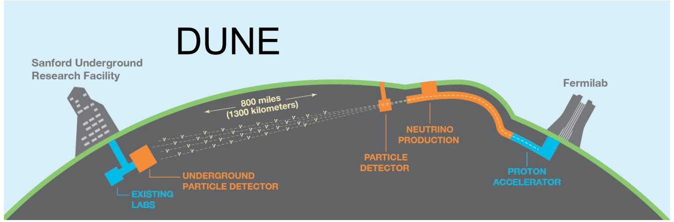

More details: CCE-PPS Overview Poster on **Wednesday:** https://indi.to/k7Bsk

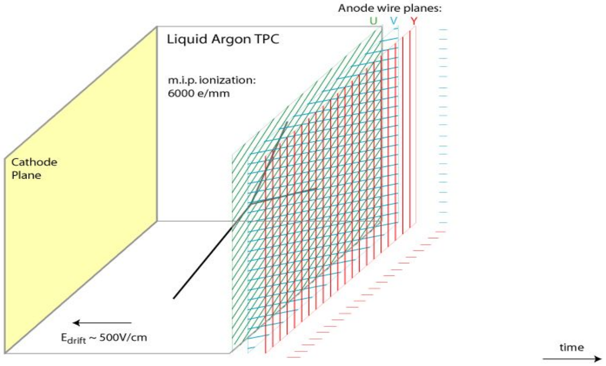
HEP-CCE involves four US labs, six experiments.
Salman Habib (ANL) PI, Paolo Calafiura (LBNL) co-PI

Liquid Argon TPC (LArTPC) and Wire-Cell Toolkit

LArTPC is a key detector technology for many next-gen neutrino experiments

- rich and precise topology info.
- calorimetry info.





LArTPC Signal Formation

Wire-Cell Toolkit (WCT) is a software package initialized for LArTPC

- algorithms: **simulation, signal processing**, reconstruction and visualization.
- data-flow programming paradigm
- modular design; can port different modules relatively independently
- works in both standalone mode and as plugin of LArSoft
- https://github.com/WireCell/wire-cell-toolkit

Wire-Cell

LArSoft is a C++ software framework for many neutrino experiments using LArTPCs

- modular design
- infrastructures + algorithms
- central hub of the LArTPC software community
- https://larsoft.org/

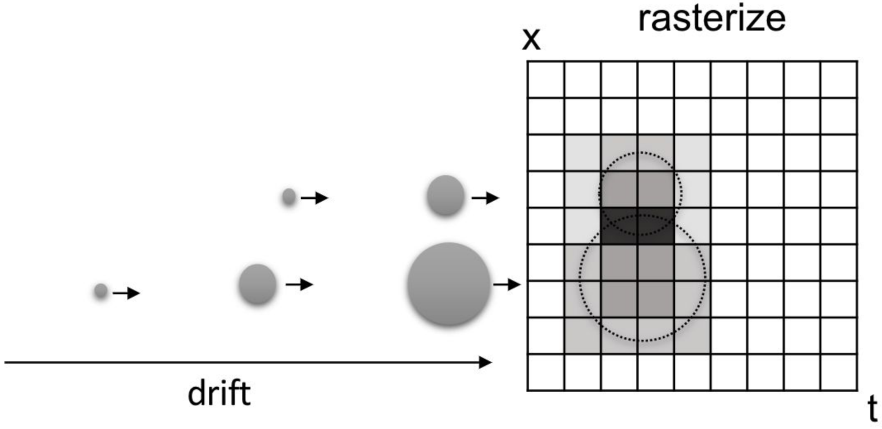
LAr Soft

Wire-Cell 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

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

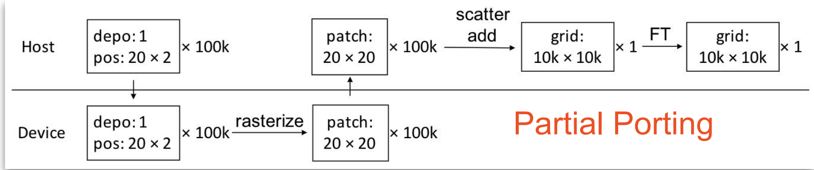
multiplication in frequency domain

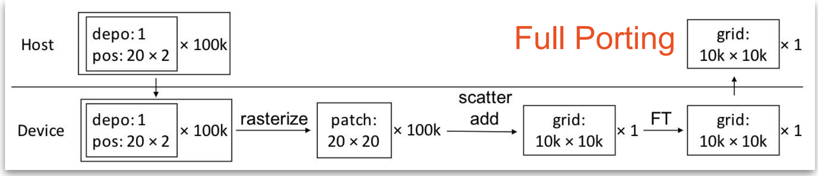
$$S(t,x) \xrightarrow{FT} S(\omega_t, \omega_x),$$
$$M(\omega_t, \omega_x) = R(\omega_t, \omega_x) \cdot S(\omega_t, \omega_x),$$
$$M(\omega_t, \omega_x) \xrightarrow{IFT} M(t,x).$$

Recap: Kokkos Porting Strategies and Results

Two stage porting strategy

1. Started with partial CUDA porting [1]: Only **rasterization part was ported to GPU**
   a. Feasibility test and baseline performance
   b. Not performant
2. Full porting [2]: All three components ported to GPU
   a. more workloads for parallelization
   b. batched device-host data transfer





Benchmarking: The same code was tested on three different architectures: 24-core AMD Ryzen Threadripper 3960X for reference CPU implementation (CPU-ref) and Kokkos with OpenMP backend running 48 threads (Kokkos-OMP48), NVIDIA V100 GPU for Kokkos-CUDA and AMD Raedon Pro VII for Kokkos-HIP.

Computation [secs]	CPU-ref	Kokko-CUDA	Kokkos-HIP	Kokkos-OMP48
Rasterization	10.45	0.05	0.04	0.15
ScatterAdd	1.14	0.0006	0.007	0.013
FFT	5.44	0.71	2.50	13.3
Total Time	18.04	0.99	2.77	13.7

Table 1: Timing for the main computational tasks on different architectures averaged over 10 runs each.

- FFT is not parallelized for CPU-ref or Kokkos-OMP48. The slowdown may be due to implementation difference. Detailed investigation is ongoing.
- We get an overall speedup of $18\times$ on V100, and $7\times$ on Raedon Pro VII.
- The GPUs are still under utilized and can be shared by several parallel processes to gain further speedup using *e.g.* CUDA MPS.

References:

[1] Z. Dong, K. Knoepfel, M. Lin, B. Viren, H. Yu and K. Yu, vCHEP 2021, arXiv: 2104.08265

[2] Z. Dong, K. Knoepfel, M. Lin, B. Viren, H. Yu and K. Yu, ACAT 2021 poster, arXiv:2203.02479

Kokkos, SYCL and OpenMP

SYCL is a programming model and (Khronos) standard that brings support for heterogeneous programming to C++ . Single source , different backend enable parallel execution on a range of hardwares CPUs GPUs, DSPs, FPGAs…

OpenMP is an API for multithreading, and it starts to support "target offloading" on heterogeneous architectures since OpenMP 4.0. It now supports several programming and memory models, including shared-memory parallelism, task parallelism, and host-device heterogeneous computing.

SYCL

```
#include <CL/sycl.hpp>

int main() {
 cl::sycl::queue Queue;
 unsigned long N=1024*1024 ;
 float a_h[N] ;
 auto a_d=cl::sycl::malloc_device<float>(N,Queue) ;
 Queue.parallel_for(
      cl::sycl::range<1>(N), [=]( auto item) {
           int id = item.get_id(0) ;
           a_d[id] = cl::sycl::sqrt((float)id ) ;
      });
 Queue.wait() ;
 Queue.memcpy(a_h, a_d, N*sizeof(float)).wait() ;
 ....
}
```

Kokkos

```
#include <Kokkos_Core.hpp>

int main() {
 Kokkos::initialize( argc, argv );
 {
 unsigned long N=1024*1024 ;
 typedef Kokkos::View<double*>   ViewVectorType;
 ViewVectorType a_d( "A", N );
 Kokkos::parallel_for("A2" , N, KOKKOS_LAMBDA (int i ) {
             a_d[i] = sqrt((double)i ) ;
       });
 Kokkos::fence();
 auto a_h = Kokkos::create_mirror_view( a_d );
 Kokkos::deep_copy(a_h, a_d, N*sizeof(double)) ;
 ....
 }
 Kokkos::finalize() ;
}
```

OpenMP

```
#include<omp.h>
#include<math.h>

int main()
{
  unsigned long N = 1024 * 1024;
  double *a = (double*)malloc(sizeof(double) * N);
#pragma omp target enter data map(to:a[0:N])
#pragma omp target teams distribute parallel for
  for(auto i=0; i<N; i++)
    a[i] = sqrt(a[i]);
#pragma omp target exit data map(from:a[0:N])
}
```

Porting Wire-Cell-Gen to SYCL

Zhihua Dong

Strategies:

- SYCL syntax is very similar to Kokkos. Porting from Kokkos is straightforward.
- Create Array1D, Array2D classes (pointer and sizes with a few methods) to replace KokkosArray (wrapper of Kokkos::view) —-> Minimum code change.

```
Kokkos::deep_copy(sps_f, spf_h);
auto  sp ts =
KokkosArray::idft_cr(sp_fs,1) ;
```

→

```
sp_fs.copy_from(sps_h);
auto  sp ts =
SyclArray::idft_cr(sp_fs,1) ;
```

- SYCL does not have a portable RNG as Kokkos.
  - We wrote a wrapper for optimized libraries (cuRAND,rocRAND,random123 for CPU) https://github.com/GKNB/test-benchmark-OpenMP-RNG.git **(Tianle Wang)**
- FFT part is similar to Kokkos. We wrote a wrapper for vendor-optimized FFT libraries (cuFFT,rocFFT, host original based on FFTW)

Things to be careful about:

- SYCL kernels and memory operation always async by default, e.g. sycl::memcpy()
- Some Kokkos functions put extra fence e.g. deep_copy()

Porting Wire-Cell-Gen to OpenMP

Tianle Wang

Strategies:

- When porting using OpenMP, we simply add #pragma for data movement and kernel, we don't need to change the CPU code a lot.
- However, the original CPU code is not suitable for GPU, so we port from the Kokkos implementation.
- Use one dimensional array to represent all the data.
- Manually perform data movement using #pragma omp target data map to remove unnecessary data movement and lower peak memory usage.
- OpenMP does not support GPU scan (prefix) operation, so we do padding, at the cost of small extra memory.
- OpenMP does not have RNG as Kokkos, we use wrapper (cuRAND,rocRAND,random123) https://github.com/GKNB/test-benchmark-OpenMP-RNG.git
- For FFT, similar to Kokkos and SYCL, we use a wrapper for cuFFT,rocFFT and FFTW.
- Use #pragma omp atomic for scattering add.

Benchmarking SYCL vs. Kokkos vs. OpenMP

Hardware Platform: SYCL, Kokkos and OpenMP versions of the code were run on the same workstation

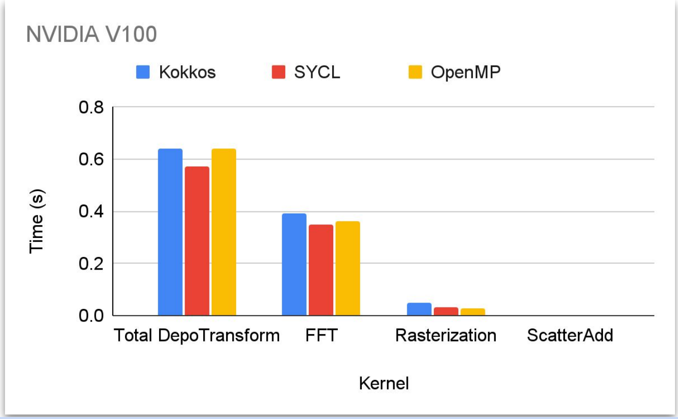
- 24 core AMD Ryzen Threadripper 3960X, 48 Hyperthreads
- NVIDIA V100 GPU
- AMD Raedon Pro VII

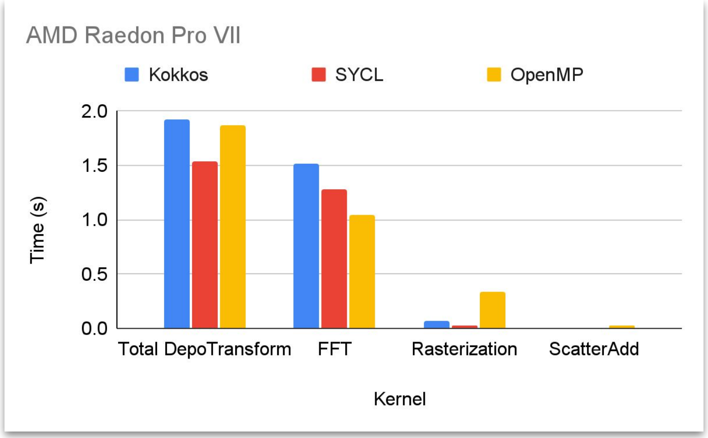
Compilers Used:

Implementation Target Architecture	SYCL	Kokkos	OpenMP
NVIDIA GPU	intel/llvm sycl-nightly20 220425	GCC9.3.0 Kokkos 3.3.01	llvm/clang 15.0.0
AMD GPU	intel/llvm sycl-nightly 20220425	GCC9.3.0 Kokkos 3.3.01 rocm-4.5.2	clang-rocm (rocm-4.5.2)
CPU multithreading	HipSYCL v0.9.3 Clang 15.0 Intel oneAPI 2022.0.2	GCC9.3.0 Kokkos 3.3.01	clang 13.0.1

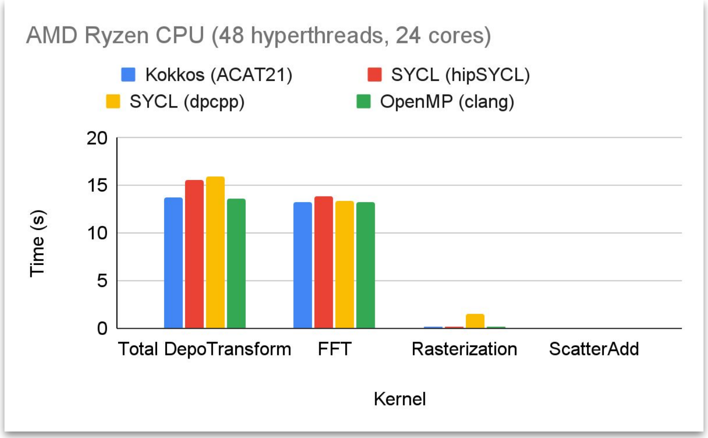
- Not one compiler that works or is optimized for all the target architectures
- Varying issues with different compilers; Lots of trial and error.
- Performance results represent the best performance obtained from our experimentation.

Performance Comparison

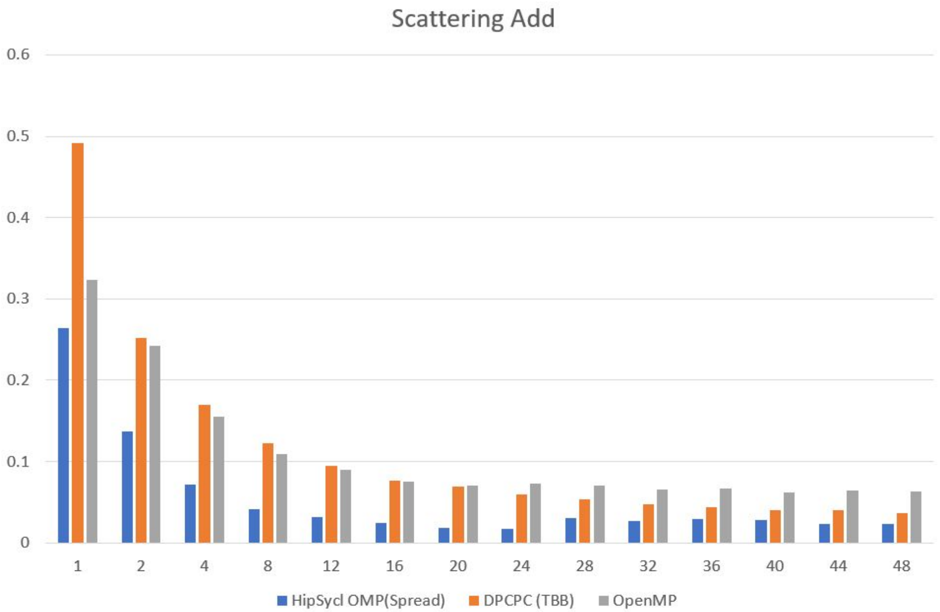


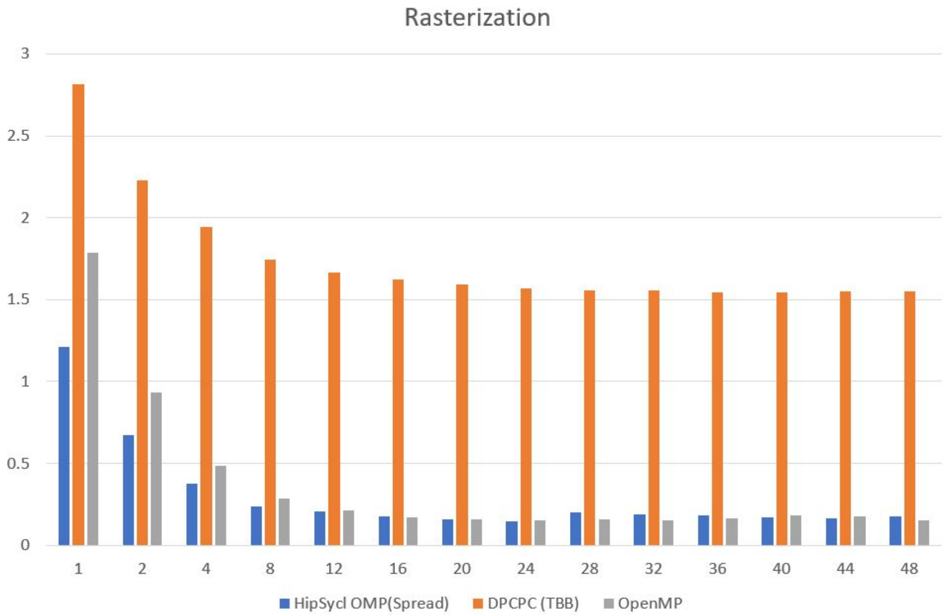


- **FFT :** SYCL/OpenMP performs better than Kokkos due to further optimizations in FFT normalization etc.
- **Rasterization:** SYCL/OpenMP performs better than Kokkos on NVIDIA GPUs mainly due to RNG . Other kernels are similar in timing
  - OpenMP HIP backend has large RNG overhead
- **ScatterAdd:**
  - SYCL 5x slower than Kokkos for CUDA backend
  - OpenMP also 5x slower than Kokkos for CUDA backend
  - SYCL same as Kokkos for HIP(AMD)



CPU Scaling Comparison (SYCL vs. OpenMP)





- Not using multithreaded FFTW at the moment. FFT dominates the CPU time.
- For Scatter-Add and Rasterization, performance is saturated at around 24 threads (24 hardware cores).
- Changing OpenMP thread-core binding can further improve the performance.

- Example of how the choice of compiler can affect performance significantly.
- Dpcpp has extra 1s+ time in the 1st kernel.

Summary

Ease of Porting:

- Porting the Kokkos implementation of WCT to SYCL was relatively straightforward due to similarities of their syntaxes.
- Porting to OpenMP was also relatively easy, but getting the most of the performance required a bit more optimization work.

Performance:

- Compiler support for both SYCL and OpenMP is still under development. Performance across different architectures is variable.
- NVIDIA GPUs are the best supported (in terms of performance) by all three programming models.

Common Issues:

- Lack of a universal API for portable optimized libraries (such as FFT and RNG).
- Code written for serial CPU processing needs to be restructured for the best parallel performance.

Recap: Kokkos port challenges and code changes

- Wire-Cell Toolkit and its associated tests (relied ton LArSoft) have many dependencies
  - Use **Docker containers** to package the dependencies, compilers and Kokkos builds
- Wire-Cell uses task-based programming model: need to retain the flexibility that some tasks may not run on the GPUs
  - Added a C++ **KokkosEnv** context manager component to initialize and finalize Kokkos
- Kokkos does not provide a wrapper API for optimized vendor FFTs (FFTW, cuFFT, etc.)
  - Implemented own FFT wrapper similar to the Synergia group
- Numerous code refactoring and reorganization to make it more GPU friendly
  - Improved RNG usage (also improved CPU performance significantly)
  - Data layout transformation to use dense matrix representation instead of sparse vectors.
  - …

Porting Wire-Cell-Gen to SYCL

Zhihua Dong

Issues:

- Various SYCL Compilers some fast developing, each have some issues for WireCell Gen code
  - Intel distributed oneAPI (dpcpp) → does not support AMD/Nvidia GPU
    - Host backend use tbb by default
    - Long delay ~1s for 1st kernel launch (?)
  - github.com/intel/llvm —> works for Nvidia/AMD,
    but hostbackend not full feature supported. (e.g group level collective like scan,sum)
  - hipSYCL —> We only got OMP backend work, others have run time error, or build error
    - Also strict in Syntax

```
q.parallel_for(N0, [=] ( auto i0) { ptr[i0] /= N0 ; } ) ;  // Won't compile on hipSYCL
```

- Due to lack of working standalone Wirecell running environment , we have to run under LarSoft framework which we need container for run on different platforms.
  Have not successfully build intel/llvm compiler for AMD backend within container.

Porting Wire-Cell-Gen to OpenMP

Tianle Wang

Issues:

- For most of the kernels, a simple porting can give a decent performance on both CPU and GPU. However, there is one kernel (set_sampling_bat) that needs different parallelism pattern.
- Different from Kokkos, it is difficult to change the data layout. Also there is no easy-to-use data structure (e.g. multi-dimensional array).
- Compiler is still developing for better performance (e.g. atomic operation).
- Currently can not compile the project with nvc++ compiler.
- Currently can not find GPU inside container, so we can not test our code on other platforms which requires container.
- Data movement speed is 1.5-3 times slower than that in Kokkos.
- We don’t need to initialize and finalize OpenMP like Kokkos, but the first data allocation on GPU requires a long time (~60 ms).
- OpenMP usually use more registers and generate more instructions than CUDA for the same kernel. This behavior becomes more observable for small kernels.