EPJ Web of Conferences **251**, 03032 (2021) https://doi.org/10.1051/epjconf/202125103032
CHEP 2021

Evaluation of Portable Acceleration Solutions for LArTPC Simulation Using Wire-Cell Toolkit

Haiwang Yu[1,*], *Zhihua* Dong[2], *Kyle* Knoepfel[3], *Meifeng* Lin[2], *Brett* Viren[1], and *Kwangmin* Yu[2]

[1]Department of Physics, Brookhaven National Laboratory, Upton, NY 11973, USA
[2]Computational Science Initiative, Brookhaven National Laboratory, Upton, NY 11973, USA
[3]Scientific Computing Division, Fermi National Accelerator Laboratory, Batavia, IL 60510, USA

Abstract. The Liquid Argon Time Projection Chamber (LArTPC) technology plays an essential role in many current and future neutrino experiments. Accurate and fast simulation is critical to developing efficient analysis algorithms and precise physics model projections. The speed of simulation becomes more important as Deep Learning algorithms are getting more widely used in LArTPC analysis and their training requires a large simulated dataset. Heterogeneous computing is an efficient way to delegate computationally intensive tasks to specialized hardware. However, as the landscape of compute accelerators quickly evolves, it becomes increasingly difficult to manually adapt the code to the latest hardware or software environments. A solution which is portable to multiple hardware architectures without substantially compromising performance would thus be very beneficial, especially for long-term projects such as the LArTPC simulations. In search of a portable, scalable and maintainable software solution for LArTPC simulations, we have started to explore high-level portable programming frameworks that support several hardware backends. In this paper, we present our experience porting the LArTPC simulation code in the Wire-Cell Toolkit to NVIDIA GPUs, first with the CUDA programming model and then with a portable library called Kokkos. Preliminary performance results on NVIDIA V100 GPUs and multi-core CPUs are presented, followed by a discussion of the factors affecting the performance and plans for future improvements.

1 Introduction

The Liquid Argon (LAr) Time Projection Chamber (LArTPC) is a key detector technology that is widely used in neutrino physics [1–4]. Both scintillation light and ionization electrons are created when charged particles traverse the LAr medium. The scintillation light can provide neutrino-interaction timing information while the detection of ionization electrons affords high-resolution position and energy information.

Accurately simulating such detector responses is difficult due to the highly diverse event topologies that occur in LArTPCs. The large number of disparate data patterns is (in part) responsible for the rising popularity in the neutrino community of deep-learning algorithms, which are capable of processing such data volumes. Training such algorithms, however,

*Corresponding Author. e-mail: hyu@bnl.gov

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (http://creativecommons.org/licenses/by/4.0/).

requires very large data sets to achieve accurate performance, presenting a challenge in terms of data handling and processing. Improved computational performance for simulation is thus key to generating enough data in a timely manner.

This paper discusses some of the challenges associated with LArTPC simulation and some efforts undertaken to improve the efficiency of generating simulated detector signals. We have made various optimizations, including using symmetries to simplify computing algorithms; trying different vendor libraries for fast Fourier transform (FFT) calculations; and also using task-level parallelism with Intel's Threading Building Blocks (TBB) [5] to balance CPU and memory load. To further boost efficiency, we have explored using graphical processing units (GPUs) as accelerators with the NVIDIA CUDA [6] API and programming model. Although the initial results are promising, two considerations motivate another approach:

1. the hardware available to us may not have the specific type of accelerators (such as NVIDIA GPUs) the code is programmed for, and
2. as technology evolves, new and better accelerators may become available.

It is thus ideal to pursue a portable solution that can provide a unified user-level API while hiding lower-level, accelerator-specific, backend interactions. One such solution is Kokkos [7], a library which provides a set of unified APIs for multiple backends. In this paper, we present our experience porting the Wire-Cell LArTPC simulation to Kokkos, including the impact on the framework, the algorithm and the kernel.

This paper is organized as follows. Section 2 briefly summarizes LArTPC simulation and its implementation in the Wire-Cell Toolkit. The CUDA porting experience is described in Section 3 and the Kokkos porting experience in Section 4, where we also present preliminary benchmark results. Future plans are described in Section 5, and we summarize in Section 6.

2 LArTPC Simulation

Figure 1 illustrates the signal formation in a typical LArTPC configuration with wire readout. When ionization electrons move close to the wires, induced currents can be detected. As governed by Ramo's theorem [8], the induced current has bipolar and unipolar shapes on the induction-plane and collection-plane wires, respectively. The recorded digitized TPC signal can be modeled as a two-dimensional (2D) convolution of the distribution of the arriving ionization electrons and the impulse detector response:

$$M(t, x) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} R(t - t', x - x') \cdot S(t', x') dt' dx' + N(t, x), \quad (1)$$

where $M(t, x)$ is a measurement, such as an analog-to-digital converter (ADC) value at a given sampling time, t, and wire position, x. $R(t - t', x - x')$ is the impulse detector response, including both the field response that describes the induced current by a moving ionization electron and the electronics response from the shaping circuit. $S(t', x')$ is the charge distribution in time and space of the arriving ionization electrons, and $N(t, x)$ is the electronics noise. The goal of TPC simulation is to calculate $M(t, x)$ based on the original charge distribution S given the known detector response R in the presence of the electronics noise N. The integral in Eq. 1 is referred to as the *signal simulation* and the additive term is referred to as the *noise simulation*. Computing the signal contribution is typically more time-consuming than computing the noise simulation. References [9, 10] introduced a LArTPC detector response simulation algorithm based on the 2D convolution, which is considered the current state-of-the-art and used by multiple experiments.

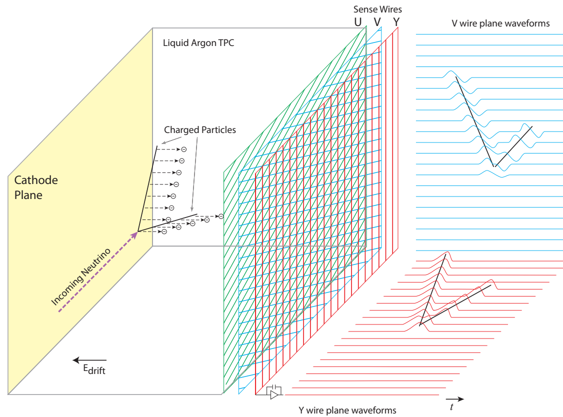

Figure 1. Diagram from Ref. [1] showing a conceptual configuration of a typical three-wire plane LArTPC and illustrating the signal formation of it. The signal in the U induction plane is omitted from the diagram for simplicity.

2.1 Wire-Cell Toolkit

2.1.1 Algorithm for signal simulation

The Wire-Cell Toolkit calculates the 2D convolution by transforming to the frequency domain, applying a multiplicative correction, and then transforming back to the time-space domain:

$$\begin{aligned} S(t,x) &\xrightarrow{\quad FT \quad} S(\omega_t,\omega_x), \\ M(\omega_t,\omega_x) &= R(\omega_t,\omega_x)\cdot S(\omega_t,\omega_x), \\ M(\omega_t,\omega_x) &\xrightarrow{\quad IFT \quad} M(t,x), \end{aligned} \tag{2}$$

where $R(\omega_t, \omega_x)$ is the pre-calculated detector response function in the frequency domain. The signal simulation process can be factored into two steps: the $S(t, x)$ calculation and the $M(t, x)$ calculation using Eq. 2. The $S(t, x)$ calculation can be further divided into two sub-steps: (1) the *rasterization* of each energy deposition into small patches ($\sim 20 \times 20$), and (2) the summation, or *scatter-adding*, of all patches to a large grid ($\sim 10,000 \times 10,000$). The exact patch and grid dimensions are determined by the actual detector configuration, governed by factors like the drifting distance, the ADC sampling frequency and the wire pitch size. Hereafter, the $M(t, x)$ calculation will be referred to as *FT* since Fourier transforms are the crucial elements in its evaluation.

2.1.2 Programming model and language

Written primarily in C++, the Wire-Cell Toolkit [11] is a software package designed according to the dataflow programming paradigm [12]. It supports a modular computing model by expressing computing tasks as nodes of a graph. These nodes are connected to form directed acyclic graphs that can be executed by various processing engines. The nodes themselves are

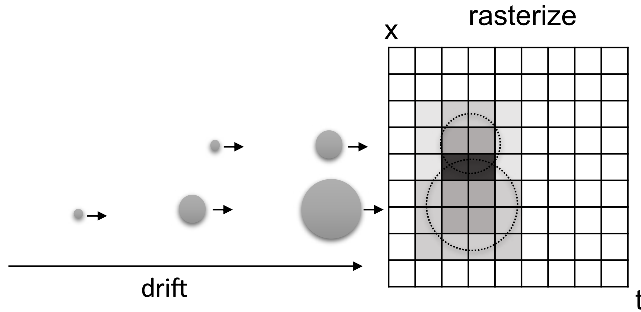

Figure 2. To illustrate the electron cloud drifting simulation and the rasterization process. In this figure, two electron clouds drift towards to the read-out plane. They expand in both transverse and longitudinal directions and overlap at the read-out plane.

polymorphic, allowing toolkit users to create and assemble concrete components according to a common framework interface. In addition to the framework infrastructure, the toolkit also provides many concrete algorithms for specific LArTPC analysis steps. The current state-of-the-art 2D convolution-based simulation is one of them, which can be run standalone or as a plugin of the LArSoft software suite [13] used by many LArTPC experiments. This simulation module is the focus of current study.

3 Initial Porting with CUDA

Before adopting Kokkos in Wire-Cell, we evaluated the potential of using NVIDIA GPUs to accelerate the signal simulation using CUDA. Performance profiling of the CPU code revealed that the most time-consuming part is in the rasterization step, which was thus ported to CUDA first. The data flow for this initial porting is shown in Figure 3 and is largely based on the original CPU code. In this example, we compute 100,000 energy depositions (depos) with a patch size of 20×20 each. The energy depositions are transferred to the device from the host one at a time, rasterized on the device and then transferred back to the host. The concurrency on the device in this scenario is very low (20×20) and only 1 GPU thread block is used. This stage represents only a partial porting and we thus expect poor performance for three reasons: (1) the data need to be transferred back and forth for the rasterization of each patch, incurring significant data transfer overhead; (2) the size of the patch (20×20) to be computed on the GPU is very small, resulting in significant under-utilization of the GPU; and (3) the scatter-adding and FT parts of the simulation are still computed serially, limiting the overall performance gain we can achieve. The effects of these limitations on the benchmarking results are presented in Section 4.3.2.

The porting above was done to minimize code changes and to familiarize ourselves with the CUDA porting process. Figure 4, however, illustrates an ideal portable solution, where all three steps in Eq. 2 are computed on GPUs. Such a solution reduces the host/device data transfers to only once at the beginning of the simulation and once at the end, and improves the overall utilization of the GPU by performing more computation on the device. This strategy, however, requires more significant code restructuring, which is in progress.

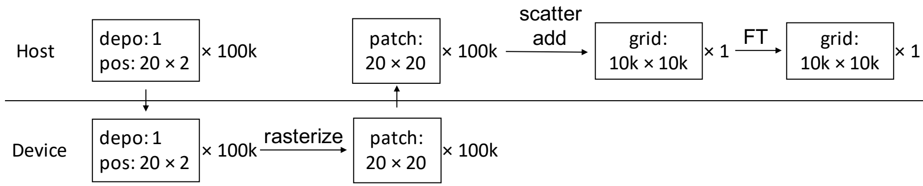

Figure 3. Data flow for current CUDA porting and first round of Kokkos porting.

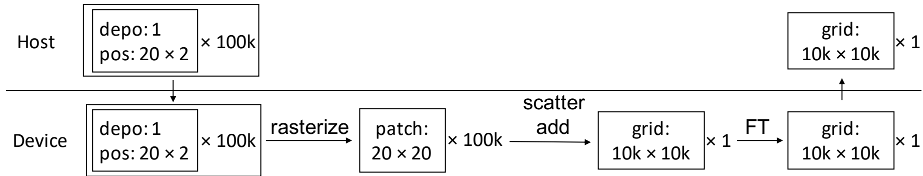

Figure 4. Planned data flow for the final Kokkos porting.

4 Portable Solution with Kokkos

4.1 Introduction to Kokkos

Kokkos [7] is a C++ library that allows developers to write code for different computing architectures and parallelism paradigms using a single API. In addition to supporting standard parallel algorithms, including a tasking model, Kokkos also supports diverse node architectures and memory models by allowing users to define their own execution and memory spaces.

The Kokkos abstraction layer maps C++ source code to the specific instructions required for the Kokkos *backends* enabled during build time. When compiling the source code, the binary code for up to three backends may be generated for a given C++ translation unit:

- *Serial* backend, which executes single-threaded on a host device.
- *Host-parallel* backend, which executes multithreaded on the host device.
- *Device-parallel* backend, which executes on an external device (e.g. a GPU).

For the host-parallel backends, Kokkos currently supports multi- and many-core CPUs through OpenMP [14] or POSIX threads (pthreads). The device-parallel backends include CUDA for NVIDIA GPUS, and HIP for AMD GPUs. In the latest version of Kokkos, experimental support for OpenMP target offloading and SyCL has also been added.

4.2 Software infrastructure to support Kokkos porting

Several software-infrastructure changes were required to facilitate building, porting and testing Kokkos along with Wire-Cell Toolkit. These include a Kokkos-enabled container environment and changes to the Wire-Cell Toolkit framework, which will be discussed in more detail below.

4.2.1 Containerized development and production

Although input data can be presented to Wire-Cell Toolkit in its standalone form via JSON serialization, more realistic data can be provided through the LArSoft library [13]. The LArSoft packages and their software dependencies require consistent versions and compatibly-built binary libraries. As the explorations described here must be performed on various platforms, a development solution portable to multiple platforms was necessary.

To achieve this, the LArSoft packages (and their software dependencies), the Wire-Cell Toolkit, the Kokkos library, and the relevant Kokkos backends were included in a CentOS-7 based Docker image (see Table 1 for some of the software packages used). Kokkos library needs to be compiled specifying the hardware architecture to make the performance optimized and some hardware features enabled. The Docker images can be readily converted to Singularity and Shifter images, the latter being required by NERSC's Cori platform. Upon starting a container of the image, an entrypoint script initializes the environment, making available all LArSoft, Wire-Cell, and Kokkos libraries needed for compiling and running (within the container) the code described in this paper.

Table 1. Software packages and their corresponding versions used for this study.

Software package	Version
GCC	8.2
Kokkos	3.3.00
CUDA toolkit	11.0.2
Eigen	3.3.9 (with patches)

4.2.2 Framework development

To develop code independently of the Wire-Cell Toolkit production code base, we created a standalone package `wire-cell-gen-kokkos` [15], which provides a shared library that can be used by Wire-Cell Toolkit as a plugin. This standalone package supports two build systems based on `waf` [16] and `cmake` [17]. Source-code files that use Kokkos are assigned a customised file extension (".kokkos") to enable the build system to identify and treat them differently from non-Kokkos files as necessary. Acquisition and release of Kokkos resources is achieved by calling `Kokkos::initialize()` and `Kokkos::finalize()`, respectively, before and after any Kokkos code executes. To insulate Wire-Cell users from having to do this explicitly, a `KokkosEnv` Wire-Cell-based object can be created, which calls `Kokkos::initialize()` in its constructor and `Kokkos::finalize()` in its destructor, following the well-known RAII C++ programming idiom.

4.3 Kokkos porting

4.3.1 Charge rasterisation kernel

As mentioned in Section 3, the current CUDA porting focuses on the rasterization of individual energy depositions as shown in Figure 3. For the first round of Kokkos porting, we followed Figure 3 and ported only the rasterization part, which already had a CUDA version, allowing a direct comparison between Kokkos and CUDA in terms of performance and development effort. The final goal is to port the entire simulation according to Figure 4.

The rasterization step, however, requires some facilities that are not provided by Kokkos, such as random numbers drawn from a normal distribution or binomial distribution. We

therefore used the Box-Muller transform [18] to generate normal-distribution random numbers from uniformly distributed ones. Similar to the CUDA implementation, we implemented a random-number pool to allow multiple threads to access the random numbers concurrently.

4.3.2 Preliminary benchmark results

We tested three LArTPC simulation implementations (original CPU, CUDA and Kokkos) on our workstation with one 24-core AMD Ryzen Threadripper 3960X CPU and one NVIDIA V100 GPU. The tests were done in the container environment as described in Section 4.2.1. The original serial CPU implementation and CUDA implementation described in Section 3 are hereafter referred to as "ref-CPU" and "ref-CUDA", respectively. For the Kokkos implementation, we tested two backends, the OpenMP backend ("Kokkos-OMP") and CUDA backend ("Kokkos-CUDA"). The input for the tests are energy depositions generated from simulated cosmic rays interacting with LAr. We used `CORSIKA` [19] as the cosmic ray generator and `Geant4` [20–22] to simulate the particle and LAr interactions. The software stack used to to generate the input files is `LArSoft` [13].

Tables 2 and 3 summarize the preliminary rasterization timing results for the CUDA and Kokkos ports following the strategy shown in Figure 3. The second column lists the total rasterization time, while the third and fourth list the timing for two major steps of the rasterization. We ran each test 5 times and averaged the results; the timing variations between runs were fairly small.

The top two rows of Table 2 show that the total rasterization time is reduced by about a factor of 3 using CUDA. That speedup, however, is mostly due to the improvement of the fluctuation calculation (the fourth column), which uses a random number generator (RNG) in the ref-CPU implementation. The current realization of the RNG for ref-CPU is `std::binomial_distribution`. In ref-CUDA, however, the RNG is factored out from the fluctuation calculation, and a pre-calculated random number pool is used instead. If we remove the RNG from ref-CPU (referred to as "ref-CPU-noRNG"), we obtain the results shown in the third row, which indicates that using a GPU actually degrades the performance. There are three reasons for this behavior:

1. Data are transferred between host and device in many blocks of a few kilobytes, unnecessarily increasing the data transfer cost;
2. The number of units of workload that needs to be parallelized (concurrency) is too small (less than 1000);
3. The amount of work per unit is too small compared to the dispatch overhead.

These results underscore the need to adopt the strategy as depicted in Figure 4 in order to improve the performance. With such a strategy, the data blocks copied to the device are batched to improve the transfer efficiency while significantly increasing GPU concurrency and reducing the relative dispatch overhead. Most importantly, this strategy allows the data to stay on the device for the scatter-adding and FT steps, further reducing data-copying time.

Table 3 shows timing results for the current Kokkos implementation, also following Figure 3. Note that the RNG has been pre-calculated, similar to the CUDA implementation, and is thus not included in the timing for the Kokkos implementation. For Kokkos-OMP, with an increased number of maximum threads, the computing time actually gets longer. This is a sign that the dispatch overhead outweighs the parallelism benefit. We think this could also be improved by adopting the Figure 4 strategy. Comparing Kokkos-CUDA and ref-CUDA, we can see that Kokkos-CUDA is about two times slower than ref-CUDA.

Analyses using the NVIDIA Nsight Systems [23] show that the causes are: (1) Kokkos `parallel_reduce()` kernels are almost 3 times slower than CUDA reduction kernels; and (2) in between kernel and API calls, Kokkos has extra `CudaDeviceSynchronization` and `CudaStreamSynchronization`. For a workflow with many small calculation kernels and data transfers, those extra steps have significant contributions to the run time.

Table 2. Preliminary timing results for the original serial CPU implementation and CUDA implementation described in Section 3. Unit is second (s). For ref-CUDA, the timing includes data transferring time between device and host. “h->d" means data transfer from host to device; “d->h" the other way around.

Description	Rasterization total [s]	2D sampling [s]	Fluctuation [s]
ref-CPU	3.57	0.07	3.42 (incl. RNG)
ref-CUDA	1.22	0.21 (incl. h->d)	0.79 (no RNG, incl. d->h)
ref-CPU-noRNG	0.18	0.07	0.03 (no RNG)

Table 3. Preliminary timing results of the first round Kokkos porting using strategy in Fig. 3. Unit is second (s). Note that the Kokkos implementation uses a pre-calculated random number pool, so the timing should be compared to ref-CUDA and ref-CPU-noRNG in Table 2.

Description	Rasterization total [s]	2D sampling [s]	Fluctuation [s]
Kokkos-OMP 1 thread	0.29	0.11	0.10
Kokkos-OMP 2 thread	0.49	0.17	0.23
Kokkos-OMP 4 thread	0.55	0.19	0.28
Kokkos-OMP 8 thread	0.66	0.24	0.34
Kokkos-CUDA	2.31	0.94	1.28

5 Future Plans

As discussed in Section 3, three major algorithms need to be ported to Kokkos: the rasterization, scatter-adding and Fourier transform. The rasterization has a CUDA implementation and porting to Kokkos is relatively straightforward, as discussed in Section 4.3.1. However, the performance is not ideal as we discussed in Section 4.3.2. We have identified the causes for the poor performance and will improve our Kokkos implementation according to the discussions in Section 4.3.2.

For the scatter-adding component, we will use the `Kokkos::atomic_add` facility. We have implemented a unit test for `Kokkos::atomic_add` in *wire-cell-gen-kokkos* to test the correctness and performance of our implementation. An initial scalability test using Kokkos with the OpenMP backend is shown in Figure 5. The test machine has a 24-core AMD Ryzen Threadripper 3960X CPU with hyperthreading enabled (2 threads per core). Two OpenMP thread affinity strategies were used: `close` and `spread`. The `close` strategy is generally faster when not all hyperthreads are in use. This may be because the `Kokkos::atomic_add` test does not perform many floating point operations, and the performance is primarily limited by the memory bandwidth, which can be alleviated by cache sharing with 2 hyperthreads per core. The linear-scaling trend breaks down after using more than 40 threads for the `close` strategy, which will be investigated further.

For the Fourier transform we use *Eigen* [24] with *fftw* [25] in the CPU code, and *cuFFT* in the CUDA code. Right now, Kokkos does not have a native portable FFT implementation or common interface for various optimized vendor FFT libraries. Until Kokkos adds support

for FFT, we will implement our own wrapper APIs over the vendor FFT libraries for different backends, similar to the approach taken by the Synergia [26] team. This work is ongoing.

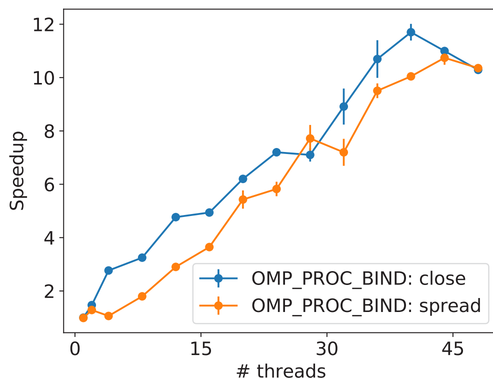

Figure 5. Scalability test for the scatter adding task using `Kokkos::atomic_add`. In this test, the grid size is 1000×6000; the patch size is 15×30; and 100,000 patches are used. The test machine has a 24-core AMD Ryzen Threadripper 3960X CPU. For each data point, the tests are run 10 times. The mean and standard deviation are shown. Two OMP thread affinity strategies are used: `close` and `spread`. The speedup numbers are calculated with regarding to the single thread `close` strategy result.

6 Summary

We have explored the feasibility of using Kokkos to implement a portable acceleration solution for LArTPC simulation in Wire-Cell Toolkit. We have ported the original serial CPU implementation to CUDA and then to Kokkos following a simple and localized strategy without significantly refactoring the original CPU code. During this process, we have learned that factoring out a random number calculation from the main loop could significantly reduce the computing time. However, this localized strategy is not enough to efficiently use the GPU accelerators due to low concurrency and high data transfer overhead. As a result, we have proposed an alternative porting strategy to better utilize the accelerators. The Kokkos implementation does run on the two different backends we tested (OpenMP and CUDA) without any change of the source code. Its performance with the CUDA backend degrades non-negligibly from the raw CUDA implementation in our test. Given that this is our initial Kokkos implementation, it is very likely that we can further optimize it.

We find that the Kokkos porting process is relatively straightforward for people who have experience with CUDA. For people without any GPU programming experience, Kokko's abstraction over architectural details is quite beneficial. While there are some limitations of Kokkos, such as architecture-matching happening at compile time rather than run time, and the lack of some library support (binomial distribution RNGs and the FFT library, for example), they can be worked around with some effort and are not major road blocks.

Acknowledgments

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) at Brookhaven National Laboratory and Fermi National Accelerator Laboratory under

B&R KA2401045. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231. We gratefully acknowledge the use of the computing resources at the Scientific Data and Computing Center of Brookhaven National Laboratory, as well as the support of the Wire-Cell team of the Electronic Detector Group in the Brookhaven National Laboratory Physics department, both of which are supported by the U.S. Department of Energy under Contract No. DE-SC0012704.

References

[1] R. Acciarri et al. (MicroBooNE), JINST **12**, P02017 (2017), 1612.05824
[2] B. Abi et al. (DUNE) (2017), 1706.07081
[3] M. Antonello et al. (MicroBooNE, LAr1-ND, ICARUS-WA104), arXiv:1503.01520 (2015), 1503.01520
[4] B. Abi et al. (DUNE) (2020), 2002.03005
[5] https://github.com/oneapi-src/oneTBB
[6] https://developer.nvidia.com/cuda-zone
[7] H.C. Edwards, C.R. Trott, *Kokkos: Enabling Performance Portability Across Manycore Architectures*, in *2013 Extreme Scaling Workshop (xsw 2013)* (2013), pp. 18–24
[8] S. Ramo, Proceedings of the IRE **27**, 584 (1939)
[9] C. Adams et al. (MicroBooNE), JINST **13**, P07006 (2018), 1802.08709
[10] C. Adams et al. (MicroBooNE), JINST **13**, P07007 (2018), 1804.02583
[11] https://github.com/WireCell/wire-cell-toolkit
[12] W.M. Johnston, J.R.P. Hanna, R.J. Millar, ACM Comput. Surv. **36**, 1–34 (2004)
[13] E.L. Snider, G. Petrillo, J. Phys. Conf. Ser. **898**, 042057 (2017)
[14] https://www.openmp.org/
[15] https://github.com/WireCell/wire-cell-gen-kokkos
[16] https://waf.io/
[17] https://cmake.org/
[18] G.E. Box, Ann. Math. Statist. **29**, 610 (1958)
[19] R. Engel, D. Heck, T. Huege, T. Pierog, M. Reininghaus, F. Riehn, R. Ulrich, M. Unger, D. Veberiˇ c, Comput. Softw. Big Sci. **3**, 2 (2019), 1808.08226
[20] S. Agostinelli, J. Allison, K. Amako, J. Apostolakis, H. Araujo, P. Arce, M. Asai, D. Axen, S. Banerjee, G. Barrand et al., Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **506**, 250 (2003)
[21] J. Allison, K. Amako, J. Apostolakis, H. Araujo, P. Arce Dubois, M. Asai, G. Barrand, R. Capra, S. Chauvie, R. Chytracek et al., IEEE Transactions on Nuclear Science **53**, 270 (2006)
[22] J. Allison, K. Amako, J. Apostolakis, P. Arce, M. Asai, T. Aso, E. Bagli, A. Bagulya, S. Banerjee, G. Barrand et al., Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **835**, 186 (2016)
[23] https://developer.nvidia.com/nsight-systems
[24] https://eigen.tuxfamily.org/index.php?title=Main_Page
[25] http://www.fftw.org/
[26] https://web.fnal.gov/sites/Synergia/