Porting and optimizing the performance of LArTPC detector simulations with C++ standard parallelism

Tianle Wang[1], Mohammad Atif[1], Zhihua Dong[1], Charles Leggett[2], Meifeng Lin[1],

[1]Brookhaven National Laboratory, Upton, NY 11973, USA
[2]Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA

High Energy Physics - Center for Computational Excellence
HEP-CCE

https://www.anl.gov/hep-cce

Introduction

There is a significant expansion in the variety of hardware architectures these years, including different GPUs and other specialized computing accelerators. For better performance portability, various programming models are developed across those computing systems, including Kokkos, SYCL, OpenMP, and others. Among these programming models, the C++ standard parallelism (std::par) has gained considerable attention within the community. Its inclusion as a part of the C++ standard library underscores its significance and potential impact, and it is also supported on AMD GPU recently.

As part of the High Energy Physics Center for Computational Excellence (HEP-CCE) project, we investigate if and how std::par may be suitable for experimental HEP workflows with some representative use cases. One of such use cases is the Liquid Argon Time Projection Chamber (LArTPC) simulation which is essential for LArTPC detector design, validation and data analysis. Following our earlier work of using Kokkos, OpenMP, and SYCL to port LArTPC simulations module, we are going to present the following topics: 1). How std::par is currently supported on different architectures and compiler, and comparison with other programming models; 2). Lesson learned from optimizing kernels with std::par; 3). Advantages and disadvantages of using std::par in porting LArTPC simulation and other HEP programs.

C++ standard parallelism (std::par)

- std::execution::parallel (std::par) was first introduced in C++17, and it is part of the C++ standard, which means it will have good backward compatibility.
- Its semantics are more similar to Kokkos/SYCL, which uses lambda expression as the functor, than directive-based programming models like OpenMP.
- It introduces execution policies for STL algorithms that allow parallel (multicore, GPU, SIMD) optimizations, which guarantees portability.
- C++ does not provide explicit memory movement functionalities, so its memory model relies on the vendor's unified memory support.
- In 2020, NVIDIA introduced a compiler (nvc++), which enabled the execution of parallel execution policies on NVIDIA GPUs. It automatically migrates the dynamic memory, e.g., memory allocated by malloc and std::vector via page fault.
- Currently, it supports more compilers/architecture combinations, including oneAPI:dpl on Intel GPU and clang on AMD GPU.
- With the development of the C++ standards, it will support more features in other programming models, e.g., mdspan (multi-dimension array)
- It has a higher abstraction level than other programming models.

OpenMP vs std::par Usage

```
//With OpenMP we need to map data to device explicitly
#pragma omp target enter data map(to: data[0:N])
//With std::par, we don't need to map data explicitly
```
```
//OpenMP uses directive to modify for loop
#pragma omp teams distribute parallel for ...
for(..., ..., ...)
//std::par use lambda functor combined with std::algorithm
//like for_each, transform, reduce, ...
std::for_each(std::execution::par, data, data+N,
[](T& ele) {...});
```
```
//With OpenMP we need to map data from device explicitly
#pragma omp target enter data map(from: data[0:N])
//With std::par, we don't need to map data explicitly
```

Implementation detail

- We use the idea of CountingIterator to allow for index-based access to arrays.
- Three of the functions in LArTPC simulation are not supported: RNG, FFT, and atomic add for floating numbers.
  - FFT: We wrap over FFTW on CPU and cufft/rocfft on GPU
  - RNG: We implement our own header-only RNG library that works on both CPU and NVIDIA/AMD GPU
  - Atomic operations: We properly convert floating point numbers to integer type and invoke std :: atomic<int> on CPU (which will cause performance loss), and wrap cuda/hip atomic functions
- Some of the algorithms need to be modified as they are not supported by std::par
  - Two layers of parallelism $\rightarrow$ loop collapsing.
  - Parallel reduction inside a parallel loop $\rightarrow$ serialize either one of the parallelism layers based on actual performance.
- Different from other programming models, std::par does not have many hyper-parameters to be fine-tuned (e.g., number of blocks/threads)
- When compiling with nvc++ and running on NVIDIA GPU, latest compiler might not work correctly. We find that nvc++ with version > 23.1 fails to compile our project. Also, we need to rebuild the library and its dependency using nvc++, otherwise we will have runtime errors.

Acknowledgement

This work is supported by US Department of Energy, Office of Science, Office of High Energy Physics under the High Energy Physics Center for Computational Excellence (HEP-CCE), a collaboration between Argonne National Laboratory, Brookhaven National Laboratory, Fermilab and Lawrence Berkeley National Laboratory.

Performance comparison

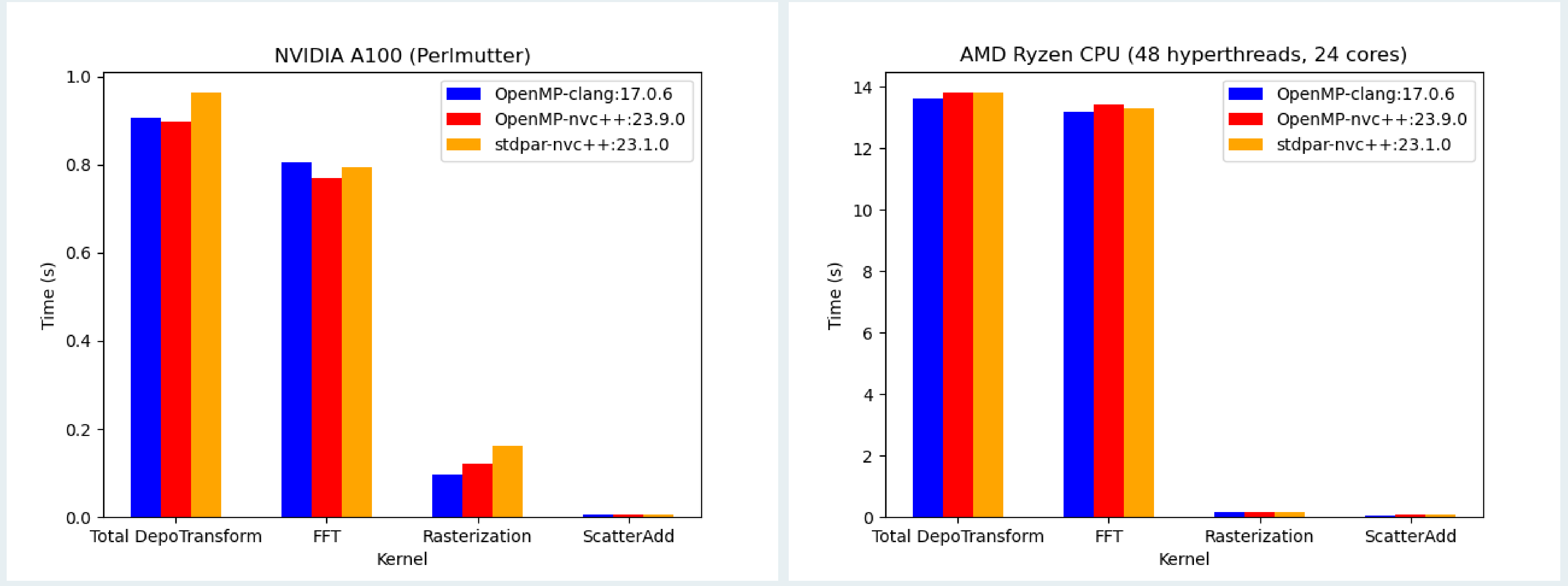


Figure 1: Running time with different compiler using OpenMP / std::par on Permultter NVIDIA-A100 GPU and multicore CPU. Currently we have not solved the compatible issue on AMD GPU.

Initial Page Migration (Prefetch)

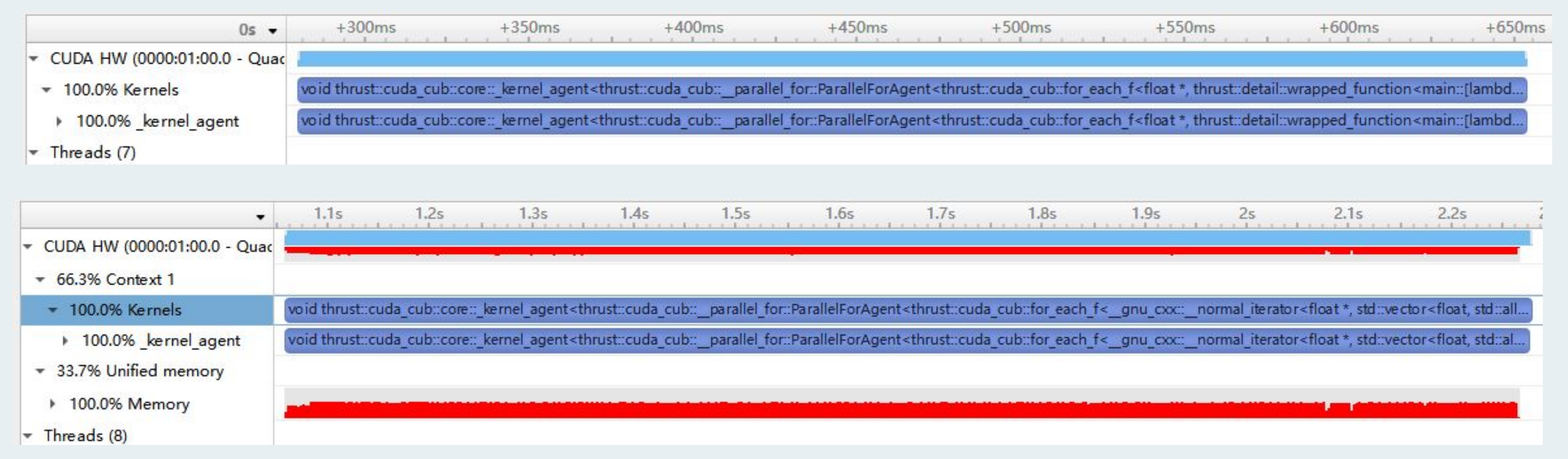
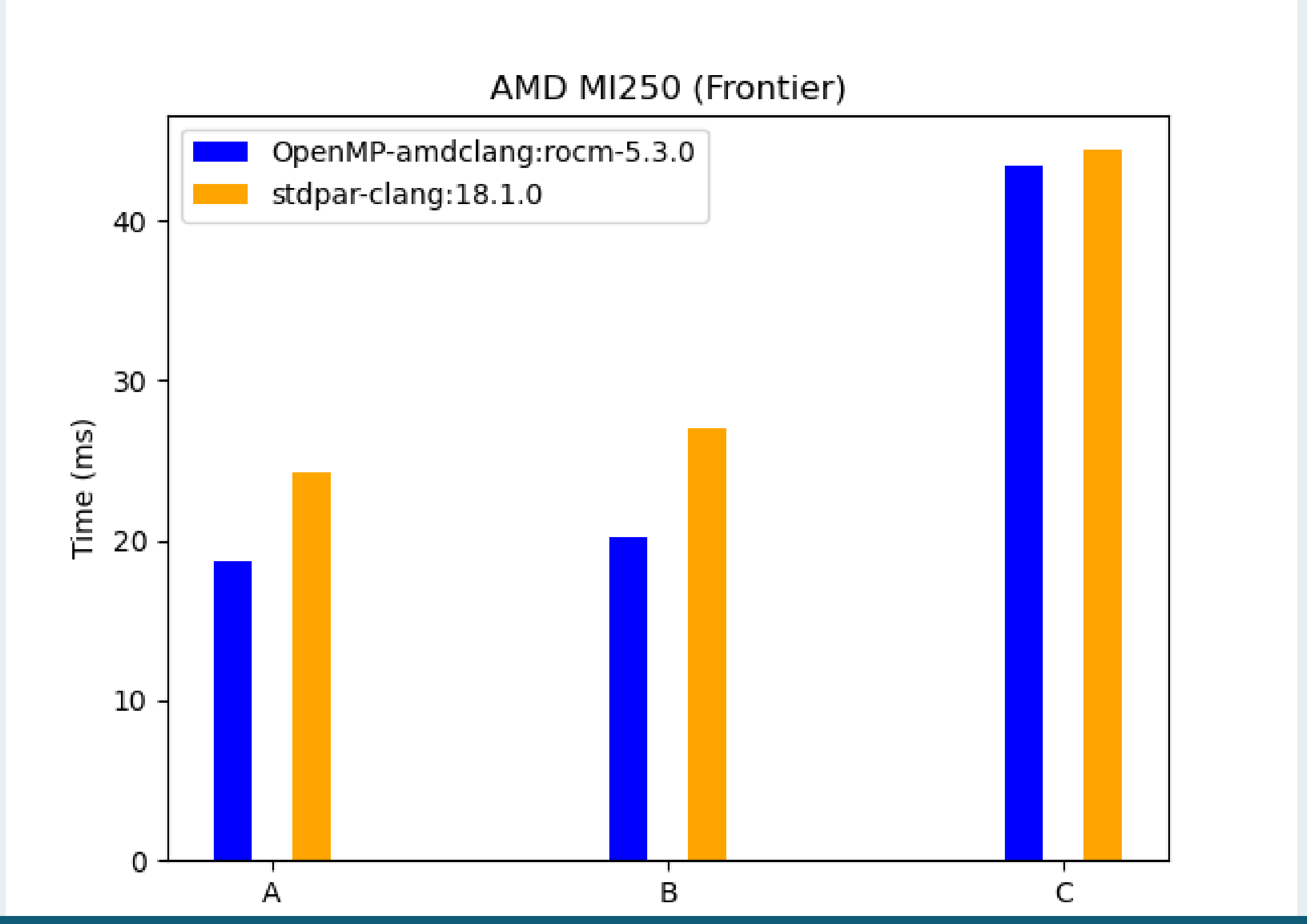

0s
+300ms
+350ms
+400ms
+450ms
+500ms
+550ms
+600ms
+650ms
CUDA HW (0000:01:00.0 - Quac
100.0% Kernels
100.0% _kernel_agent
Threads (7)
void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__parallel_for::ParallelForAgent<thrust::cuda_cub::for_each_f<float *, thrust::detail::wrapped_function<main::[lambd...
1.1s
1.2s
1.3s
1.4s
1.5s
1.6s
1.7s
1.8s
1.9s
2s
2.1s
2.2s
66.3% Context 1
33.7% Unified memory
100.0% Memory
Threads (8)
void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__parallel_for::ParallelForAgent<thrust::cuda_cub::for_each_f<__gnu_cxx::__normal_iterator<float *, std::vector<float, std::all...

Figure 2: The kernel/data movement trace of a simple axpy-like kernel, where data are initialized on GPU. Top: Use raw pointer with malloc to allocate memory. Bottom: Use std::vector to allocate memory.

For data that are initialized on GPU, one optimization is to create the object on GPU instead of CPU so that we don't need to perform data movement. When a std::vector object is created, it will automatically create the data on CPU, which later introduces unnecessary page migration. Because of that, we recommend using raw pointers with malloc for data that are initialized on GPU.

AMD GPU: first look

Here, we test the performance of the rocm-stdpar implementation on AMD GPU by comparing it with OpenMP, using three representative kernels: an axpy-like kernel (A), a reduction kernel (B), and a computationally costly kernel (C). These three kernels all have different performance bottlenecks.
For these experiments, we set HSA_XNACK=1 and compile without −−hipstdpar−interpose−alloc flag.



Future plan

- Test and Port to Intel GPU
- Experiment on the effect of XNACK on AMD GPU
- Experiment with other implementation of std::par

Reference

1 https://github.com/GKNB/test-benchmark-OpenMP-RNG
2 https://github.com/GKNB/Wire-cell-gen-stdpar
3 Lin, Meifeng, et al. "Portable Programming Model Exploration for LArTPC Simulation in a Heterogeneous Computing Environment: OpenMP vs. SYCL." arXiv preprint arXiv:2304.01841 (2023).

Work supported by US Department of Energy, Office of Science, Office of High Energy Physics under the High Energy Physics Center for Computational Excellence (HEP-CCE), a collaboration between Argonne National Laboratory, Brookhaven National Laboratory, Fermilab and Lawrence Berkeley National Laboratory.

Argonne NATIONAL LABORATORY
Brookhaven National Laboratory
Fermilab
BERKELEY LAB

CHEP-2023,
5/8/2023-5/12/2023