Accelerating LBM with C++ STL Asynchronous Parallel Model

Ziheng Yuan[1] and Takashi Shimokawabe[2]

[1] Department of Electrical Engineering and Information Systems
Graduate School of Engineering, The University of Tokyo
Hongo 7-3-1, Bunkyo-ku, Tokyo 113-8656, Japan
Zihengyuan22@g.ecc.u-tokyo.ac.jp
[2] Information Technology Center, The University of Tokyo
6-2-3 Kashiwanoha, Kashiwa-Shi, Chiba, 277-0882, Japan
shimokawabe@cc.u-tokyo.ac.jp

Abstract. Asynchronous computation is an important optimization technique in scientific computation. The upcoming C++26 standard introduces a new asynchronous execution framework, stdexec, enabling the development of high-performance code using only standard C++. This paper explores the parallelization of single-GPU and multi-GPU lattice Boltzmann method computations using stdexec and further optimizes performance through its asynchronous execution model. Experimental results show that asynchronous stdexec achieves approximately 83.5%-105.4% of the performance of C++ stdpar. These results suggest potential for further optimizations in the future, providing additional options for high-performance computing development in pure C++.

Keywords: Parallel Computing · High Performance Computing · GPU · stdexec · lattice Boltzmann method

1 Introduction

The parallel programming libraries can currently be categorized into two approaches. The first approach is hardware-specific libraries designed for particular hardware platforms, characterized by their ability to provide fine-grained control over hardware. Representative examples include CUDA [1], OpenCL [2] and HIP [3]. The second approach libraries provide high-level APIs to abstract hardware details. Typical examples of this category include Kokkos [4], OpenMP [5] and OpenACC [6]. To enable programming using pure C++ and enhance code compatibility, two new features have been or will be introduced into the C++ standard, referred to as C++ standard language parallel model (stdpar) [7] and C++ standard model for asynchronous execution (stdexec) [8], following the second approach. stdpar has already been integrated into C++17. As an extension to the existing algorithm, stdpar enables parallel execution support for specific function in standard algorithm library. stdexec provides the asynchronous execution framework, which is not supported by stdpar. Before stdexec is integrated

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_30

into C++26 in the future. a prototype of stdexec library provided by NVIDIA is available for testing. This paper focuses on evaluating the feasibility of performing parallel programming using C++ stdexec, analyzing its performance and comparing it with other parallel programming labraries. Lattice Boltzmann method (LBM) [9] is selected as the benchmark problem for evaluation due to its wide applicability in the field of fluid dynamic and its suitability for parallel computation.

2 Background Knowledge

2.1 C++ stdexec

The stdexec sender/receiver model is defined by three critical components: executor, sender/receiver, and scheduler. The role of the executor is to provide a uniform task execution interface by abstracting hardware resources. The sender/receiver is responsible for supplying tasks to the executor. The scheduler serves to provide an abstract interface for the management of hardware resource [10]. It is important to note that one scheduler can only manage one hardware resource associated with it, this hardware could be CPU or GPU. Listing 1.1 and 1.2 illustrates the structure of C++ STL parallel model with an example.

```
auto A = stdexec::just()
 | exec::on(sched, stdexec::bulk(n, GPU_task1))
 | stdexec::then(CPU_task)
 | stdexec::let_value([&]{return stdexec::just()
 | exec::on(sched, stdexec::bulk(N, GPU_task2));});
```

Listing 1.1: C++ stdexec synchronous example

```
auto A = stdexec::when_all(stdexec::just()
| exec::on(sched_low,   stdexec::bulk(N, GPU_task2)),
stdexec::just()
| exec::on(sched_high, stdexec::bulk(n, GPU_task1))
| stdexec::then(CPU_task));
```

Listing 1.2: C++ stdexec asynchronous example

In this scenario, three tasks executed on either the CPU or GPU: `CPU_task`, `GPU_task1` and `GPU_task2`. Among them, both `CPU_task` and `GPU_task2` are waiting the result from `GPU_task1`. Listing 1.1 and 1.2 represent different approaches to executing the same tasks. `stdexec::bulk()` is the stdexec equivalent of a for-loop, utilizing the resources allocated by `stdexec::on()` to execute tasks [11]. `stdexec::when_all()` provides synchronization functionality. Listing 1.1 implements basic synchronous execution. `GPU_task1`, `CPU_task` and `GPU_task2` is executed in serial order. Listing 1.2 implements asynchronous execution by assigning schedulers with different priority levels. The program assigns high priority scheduler `sched_high` to `GPU_task1` and low priority scheduler `sched_low` to `GPU_task2`. While `GPU_task2` is being continuously executed,

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_30

`GPU_task1` with higher priority, will complete its execution first and subsequently proceed to execute `CPU_task`, overlap the execution time.

2.2 Lattice Boltzmann method

LBM is a fluid dynamic algorithm that simulates fluid by emulating the streaming and collision of virtual fluid particles in mesoscale. Particles motion is characterized by discrete velocity set according to the Boltzmann equation [9]. The LBM uniformly divides the computational domain into lattice like orthogonal grids [12]. This experiments use D3Q27 model as the simulation model. Necessary physical values include density ρ, velocity u and equilibrium distribution function f_i are computed from equation (1)-(3).

$$\rho = \sum_i f_i \qquad u = \frac{1}{\rho}\sum_i c_i f_i \tag{1}$$

$$f_i(x + c_i \Delta t, t + \Delta t) = f_i(x, t) \tag{2}$$

$$f_i(x, t + \Delta t) = f_i^*(x, t + \Delta t) + \Omega(x, t) \tag{3}$$

In these equations, c_i represent for velocity set, Δt for time step, Ω for collision model. Equation (1) shows density and velocity computation method. Equation (2) calculates streaming step. Equation (3) calculates the collision step. In this experiment, we use the classic BGK collision model [13] and bounce-back boundary condition [14] for LBM.

Boundary computation and LBM streaming-collision computation is the two main steps for LBM computation. The boundary code computes the numerical values at the boundaries based on the condition, storing the results for use in the computation of streaming-collision step. For multi-GPU experiments, data exchange between multiple GPUs is based on MPI. The performance benchmark used is Mega Lattice Updates Per Second (MLUPS), which represents the number of LBM lattice updates (computations) per unit time. The specific calculation method is outlined as follows:

$$MLUPS = \frac{lattice\ size \times number\ of\ iterations}{total\ time\ consumption} \tag{4}$$

3 Experiment and result

3.1 Experiment condition

Hardware resource The experiments were conducted on a Wisteria-Aquarious HPC platform provided by The University of Tokyo, utilizing A100 40GB GPUs. Each computation node in Wisteria-Aquarious contains 8 GPUs. The compilation was performed using the nvidia/24.1 compiler with the C++20 standard.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_30

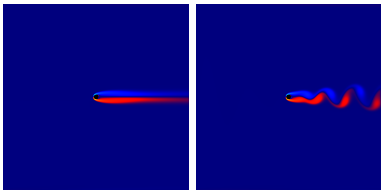

Fig. 1: Visualization of simulation result. The content of these images represents the variation of the velocity norm along the z-axis over time.

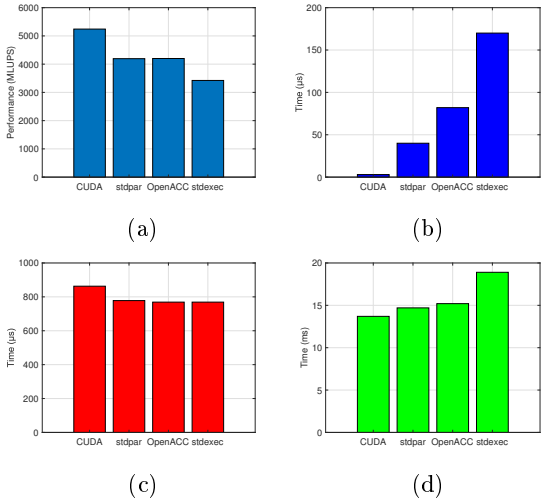

Fig. 2: Performance of single GPU execution and time breakdown comparison of stdexec, CUDA, stdpar and OpenACC in single GPU case. (a): Performance of single GPU execution. (b): Kernel launch time consumption. (c): Boundary computation time consumption. (d): LBM streaming-collision computation time consumption.

LBM condition The code presented in this article is applied to the benchmark simulation case of 3D flow around a cylindrical obstacle. A solid cylinder is fixed along the z-axis. A uniform fluid flow enters the computational domain along the x-axis. The computational domain has a size of $512 \times 512 \times 256$ for both single GPU and multi GPU conditions. Reynolds number $\mathrm{Re} = 1000$. Figure 1 illustrates the evolution of the flow field over iterations.

3.2 Single GPU result

First, the performance of the code is tested under a single GPU scenario. The implementation of stdexec code is similar to Listing 1.1, where `GPU_task1` cor-

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_30

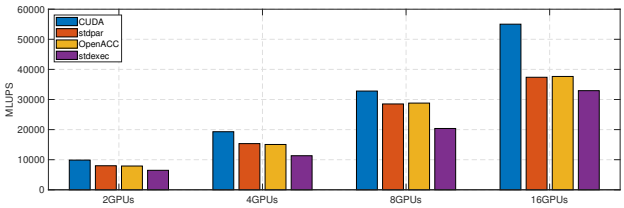

Fig. 3: Performance of multi GPU execution.

responds to boundary computation and `GPU_task2` corresponds to streaming-collision computation. However, the code does not include the `CPU_task` part, as it represents the function required by multi-GPU communication. Figure 2 (a) illustrates the performance of different codes when solving the LBM problem under the same condition described in Section 3.1. It can be observed that CUDA achieves the best parallel computing performance. OpenACC and stdpar exhibit similar performance, while stdexec shows the lowest performance. The performance of stdexec reaches 65.2% of CUDA and 81.6% of stdpar. This result leads to the conclusion that stdexec does not offer a performance advantage when relying solely on `stdexec::bulk()` for parallel computation. An analysis of the function's execution time is required to find the reason.

According to Figure 2(b), Compared to the fastest CUDA implementation, the kernel launch time of stdexec is approximately 60 times longer [15]. As shown in Figure 2(c), for boundary kernels, which involve small amounts of data, the performance across different implementations is similar. However, Figure 2(d) reveals a more significant performance gap for large data size LBM streaming collision computation, with stdexec showing a execution time consumption difference of approximately 38.0% compared to CUDA.

3.3 Multi GPU result

The performance of the code is then tested in a multi-GPU scenario. The implementation of stdexec code is similar to Listing 1.1, with `CPU_task` corresponds to boundary data exchange function based on MPI, this introduces additional communication cost. Figure 3 presents the performance of different codes when solving the LBM problem under the same condition described in Section 3.1.

Similar to the results from the single-GPU tests, the performance of stdexec in the multi-GPU scenario still shows a significant gap compared to CUDA according to Figure 4, while OpenACC and stdpar exhibit similar performance. In the 2 GPU case, the performance of stdexec is approximately 65.5% of CUDA and 81.0% of stdpar. In the 4 GPU case, the performance of stdexec is approximately 58.7% of CUDA and 73.8% of stdpar. In the 8 GPU case, the performance of stdexec is approximately 62.1% of CUDA and 71.4% of stdpar. In the 16 GPU case, the performance of stdexec is approximately 59.9% of CUDA and 88.1% of stdpar.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_30

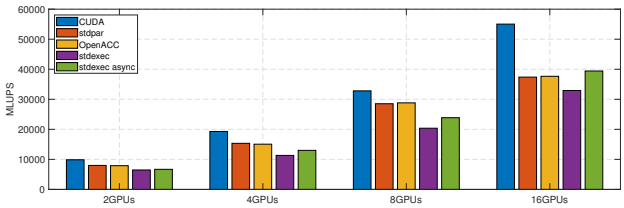

Fig. 4: Performance of multi GPU asynchronous execution. To compare the performance of stdexec async, the results for CUDA, stdpar, OpenACC, and stdexec are the same as those presented in Figure 3.

3.4 Multi GPU asynchronous result

Finally, the performance of asynchronous computation in a multi-GPU setting is evaluated. Figure 4 presents the performance of different implementations when solving the same LBM problem, with experimental conditions remaining the same as before. In this experiment, stdexec async utilize the methods introduced in Listing 1.2. The high-priority GPU kernel for boundary computation completes first and initiates MPI communication, effectively overlapping the MPI communication time with the execution of the low-priority scheduler GPU kernel for streaming-collision. In the 2 GPU case, the performance of stdexec async is approximately 67.8% of CUDA, 83.7% of stdpar and 100.3% of stdexec. In the 4 GPU case, the performance of stdexec async is approximately 67.3% of CUDA, 84.7% of stdpar and 114.8% of stdexec. in the 8 GPU case, the performance of stdexec async is approximately 72.6% of CUDA, 83.5% of stdpar and 117.0% of stdexec. in the 16 GPU case, the performance of stdexec async is approximately 71.6% of CUDA, 105.4% of stdpar and 119.7% of stdexec.

By analyzing the results, it can be observed that compared to stdexec without asynchronous computation, the asynchronous version of stdexec achieves up to a 19.7% performance improvement. To determine the source of performance improvements, it is necessary to analyze the execution time of individual kernels. Figure 5 presents the kernel execution times for a multi-GPU stdexec program without asynchronous computation, as well as for async, when running on 16 GPUs. Figure 5(a) illustrates the boundary computation time consumption per iteration, showing that performance of the two is nearly identical. A similar trend is observed for both LBM streaming collision computation and MPI communication shown in Figure 5(b) and Figure 5(c). The only notable difference lies in the overall kernel execution time shown in Figure 5(d), which is the combination of MPI communication of LBM streaming collision compution time consumption. The performance of 16 GPUs in stdexec async is higher because inter-node communication is hidden, as expected. Since the overall computational domain does not change as GPU number increases, the size allocated to each GPU decreases. At the same time, the size of boundary remains unchanged, resulting in a gradual increase in the proportion of communication, which creates room

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_30

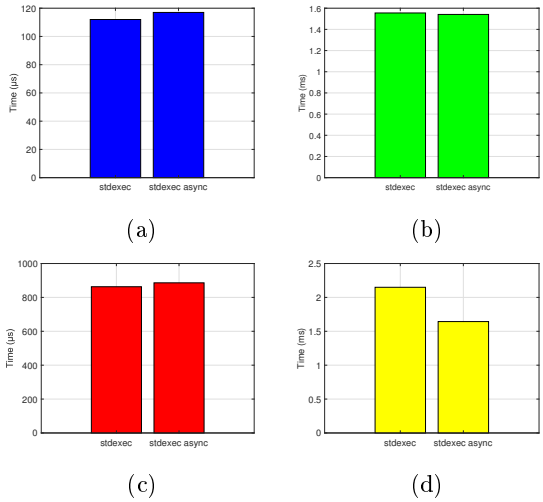

Fig. 5: Time breakdown comparison of sync and async stdexec in 16 GPUs case. (a): Boundary computation time. (b): LBM streaming-collision computation time. (c): MPI communication time. (d): Overall kernel execution time.

for optimization.This shows the importance of asynchronous communication and may be explained as the strength of stdexec's ability to introduce asynchronous communication into pure C++ code.

Observations indicate that stdexec async achieves 23.2% reduction in execution time compared with stdpar. 95.7% of the communication time was hidden in stdexec async. This reduction is attributed to the asynchronous execution of MPI communication and LBM streaming collision computation. However, due to the high kernel launch overhead at the start of each iteration (approximately 170 ms), part of the performance gains achieved by stdexec async is lost. As a result, the overall reduction in program runtime is smaller than the reduction in kernel execution time.

4 Conclusion

In summary, this paper evaluates the application of the C++26 stdexec prototype in the LBM. The performance of stdexec is compared with other programming languages and implementation approaches, and kernel analysis is conducted to explain the observed performance differences. The test code is based on the prototype provided by NVIDIA and utilizes advanced C++ syntax introduced in C++17 and later versions. The experimental results demonstrate the baseline performance of stdexec, showing that its performance is comparable to stdpar in some cases. This suggests that stdexec provides an alternative option for developing high-performance C++ code. We also encountered software engineering challenges related to compilers and libraries, and shared information

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_30

with NVIDIA engineers. We will continue to track the progress of the C++26 standard and explore further optimizations for the code in the future.

5 Acknowledgments

This work was partly supported by JSPS KAKENHI Grant Number JP24K02947. This work was also partly supported by JHPCN projects jh240052 and jh250037.

References

1. NVIDIA CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit. Last accessed 2025/02/10
2. OpenCL Official guide. https://github.com/KhronosGroup/OpenCL-Guide. Last accessed 2025/02/15
3. Diederichs, D.: Available now: new HIP SDK helps democratize GPU computing. https://community.amd.com/t5/instinct-accelerators/available-now-new-hip-sdk-helps-democratize-gpu-computing/ba-p/621029. Last accessed 2025/02/10
4. Trott, C.R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., et al.: Kokkos 3: Programming model extensions for the exascale era. In: IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 4, pp. 805–817. IEEE (2021)
5. Haseeb, M., Ding, N., Deslippe, J., Awan, M.: Evaluating performance and portability of a core bioinformatics kernel on multiple vendor GPUs. In: 2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 68–78. IEEE, St. Louis, MO (2021)
6. Farber, R.: Parallel programming with OpenACC. Newnes, USA (2016)
7. Lopez, G., Olsen, D., Adelstein Lelbach, B.: Accelerating Standard C++ with GPUs Using stdpar. NVIDIA Technical Blog — https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/. Last accessed 2025/02/10
8. Garland, M., et al.: A Unified Executors Proposal for C++ | P0443R14. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0443r14.html. Last accessed 2025/02/10
9. Kruger, T.: The Lattice Boltzmann Method: Principle and Practice. ISBN 978-3-319-44647-9 (2017)
10. Arutyunyan, R.: P2500R0 C++17 parallel algorithms and P2300 Published Proposal. ISO/IEC JTC1/SC22/WG21 (2022)
11. Dominiak, M., et al.: std::execution Published Proposal. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2300r7.html. Last accessed 2025/02/10
12. Lagrava, D., Malaspinas, O., Latt, J., Chopard, B.: Advances in multi-domain lattice Boltzmann grid refinement. J. Comput. Phys. **231**(14), 4808–4822 (2012). https://doi.org/10.1016/j.jcp.2012.03.015
13. Qian, Y.H., et al.: Lattice BGK Models for Navier-Stokes Equation. Europhys. Lett. **17**(6), 479 (1992). https://doi.org/10.1209/0295-5075/17/6/001
14. Ladd, A.J.C.: Numerical simulations of particulate suspensions via a discretized Boltzmann equation. J. Fluid Mech. **271**, 285–309 (1994). https://doi.org/10.1017/S0022112094001783
15. Haseeb, M., Wei, W., Deslippe, J., Cook, B.: That's Right – The Same C++ STL Asynchronous Parallel Code Runs on CPUs and GPUs. SC23, Denver, USA (2023)

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_30