That's right, the same C++ STL asynchronous parallel code runs on CPUs & GPUs

MUHAMMAD HASEEB* and WEILE WEI*, National Energy Research and Scientific Computing Center, USA
JACK DESLIPPE, National Energy Research and Scientific Computing Center, USA
BRANDON COOK, National Energy Research and Scientific Computing Center, USA

CCS Concepts: • **Computing methodologies → Parallel programming languages**; *Distributed programming languages*; • **Software and its engineering** → *Parallel programming languages*; *Distributed programming languages*; *Runtime environments*.

Additional Key Words and Phrases: C++, stdpar, stdexec, sender/receiver, stencils, CPU-GPU computing

ACM Reference Format:
Muhammad Haseeb, Weile Wei, Jack Deslippe, and Brandon Cook. 2023. That's right, the same C++ STL asynchronous parallel code runs on CPUs & GPUs. In . ACM, New York, NY, USA, 3 pages. https://doi.org/XXXXXXX.XXXXXXX

EXTENDED ABSTRACT

High-performance computing (HPC) applications are increasingly shifting towards asynchronous parallelism to extract maximum performance from modern GPU-accelerated supercomputers. To achieve this, they employ combinations of programming models, languages, and compilers. This practice often leads to significant challenges on performance, portability, and productivity (P3) along with software engineering efforts as the underlying hardware varies across the HPC systems [2]. Recently, many programming models provide high-level parallelism APIs to HPC developers in order to hide architecture complexity, including Kokkos [16], RAJA [4], SYCL [14], OpenACC [9], C++ standard parallelism (`stdpar`) [6] and OpenMP [5] [11], [10], [1]. Although these frameworks provide different advantages, they still need to be carefully configured, installed, and used to provide decent performance and portability. Further, in most cases, the developers often need to mix multiple frameworks depending on their application's needs.

Recently, a C++ model for asynchrony has been voted into C++ 26 standard, called `std::exuecution` or `stdexec` [7], [8]. `stdexec` standardizes a C++ asynchrony API for where the compute should execute. `stdexec` contains three major abstractions: 1) schedulers - obtained from execution resources describing where to run a piece of code. 2) senders and receivers - send and receive a composition of compute work asynchronously to and from the input schedulers. 3) a set of customizable asynchronous algorithms - consume, compose, and optionally return senders.

In this work, we employ an experimental, open-source, and standard reference implementation of the `stdexec` by NVIDIA [13], to evaluate standard C++26 based asynchronous parallelism across CPUs and GPUs for multiple scientific HPC applications. These applications include ADEPT (low-level CUDA-accelerated Smith-Waterman) applications [3], Heat Equation codes adapted from AMReX [17] and Stencil codes from HPX [12]. In addition to `stdexec`, we also demonstrate several modern C++23 features including `mdspan` [15], `stdpar`, `ranges` and more, in our application implementations.

*Both authors contributed equally to this research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
Manuscript submitted to ACM

We begin with the ADEPT applications and compose its *main*-level asynchronous task flow using `stdexec::then` and `stdexec::bulk` algorithms. In each algorithm, we implement the required data partitioning and launched asynchronous parallel CUDA kernels to perform the low-level Smith-Waterman sequence alignment. Due to the unavailability of explicit parallelism controls, GPU shared memory, and warp and thread-level primitives in `stdexec`, the ADEPT CUDA kernels could not be directly ported to `stdexec` using GPU schedulers. For AMReX Heat Equations and Stencil code implementations, we employ combinations of C++23 features with `stdexec` to implement different variations (or flavors) of the codes using loops and recursions that run on CPUs and GPUs. These flavors include using `mdspan`, `mdspan` and `stdpar`, `mdspan` and `stdexec`, `mdspan`, `stdpar` and `stdexec`, and finally, directly using schedulers from `stdexec`. In all these implementations, we preserve the original task flow with minimal optimizations for fairness. For instance, we use return by value semantics in recursions for implicit data copying, instead of simply swapping pointers, to retain the explicit copy operations in the original implementations. It is worthwhile to note that our codes can be further trivially modified to run on either CPUs or GPUs by either modifying the compiler flags `-stdpar=gpu/multicore` for `stdpar` flavors or by swapping between gpu and multicore schedulers for `stdexec` flavors. Finally, we also implement a ping-pong stress benchmark to study data traffic in `stdpar` based codes with respect to the memory allocation schemes. i.e., allocated as either simple pointers such as `T *ptr = new T[N]` or STL conatiners such as `std::vector <T> vect(N)`.

Our experimental results (speedup and roofline analyses) show that the `stdexec` powered C++ codes perform similar to their original (non-`stdexec`) implementations for all applications. However, we observed that the kernel launch latencies in `stdexec` implementations were significantly higher (620us) than those in their CUDA counterparts (10us). We also observe a severe load imbalance across GPUs when using MultiGPU scheduler in `stdexec` implementations. These two issues have been reported to the NVIDIA compiler team. Our experimental results for `stdpar` data traffic are particularly interesting. We found that in case of pointers, the data communications are particularly smart and only the portions of memory accessed by one side (host or device) also needed at the other side are communicated. However, in case of STL containers, the H2D communication is highly unoptimized and may transfer entire data container to the device, even when particularly not accessed at the device. The D2H communications, on the other hand, are optimized like when using pointers. We also observed that in both cases (pointer or STL containers), the data communications are split over thousands of small transfers - average H2D Mbytes/call = 0.019, D2H Mbytes/call = 0.175 ($\sim 10\times$ H2D) - instead of single tranfer incurring additional latencies.

To summarize, this work evaluates the C++ 26 `stdexec` asynchronous model on top of several HPC applications. We use the experimental implementation of the `stdexec` model by NVIDIA, along with modern C++17 and 23 features to develop multiple HPC scientific applications. Our experimental results show identical parallel performance for `stdexec` codes. We also observed unexpectedly high launch latencies for `stdexec` codes as compared to their CUDA versions which have been reported to the NVIDIA team. We also encountered several software engineering challenges including working around NVHPC compiler limitations, compilation flags, and correctly setting up dependencies. We are currently actively developing more HPC applications involving complex task graphs and algorithmic challenges to better evaluate the application of `stdexec` in real-world science.

ACKNOWLEDGMENTS

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] Victor Artigues, Katharina Kormann, Markus Rampp, and Klaus Reuter. 2020. Evaluation of performance portability frameworks for the implementation of a particle-in-cell code. *Concurrency and Computation: Practice and Experience* 32, 11 (2020), e5640.

[2] Yuuichi Asahi, Thomas Padioleau, Guillaume Latu, Julien Bigot, Virginie Grandgirard, and Kevin Obrejan. 2022. Performance portable Vlasov code with C++ parallel algorithm. In *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, Dallas, TX, 68–80.

[3] Muaaz G Awan, Jack Deslippe, Aydin Buluc, Oguz Selvitopi, Steven Hofmeyr, Leonid Oliker, and Katherine Yelick. 2020. ADEPT: a domain independent sequence alignment strategy for gpu architectures. *BMC bioinformatics* 21, 1 (2020), 1–29.

[4] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. 2019. RAJA: Portable performance for large-scale scientific applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, Denver, CO, 71–81.

[5] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[6] Graham Lopez David Olsen and Bryce Adelstein Lelbach. 2022. Accelerating Standard C++ with GPUs Using stdpar | NVIDIA Technical Blog — developer.nvidia.com. https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/. [Accessed 03-08-2023].

[7] Michal Dominiak et. al. 2023. P2300R7: 'std::execution' — open-std.org. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2300r7.html. [Accessed 03-08-2023].

[8] Michael Garland et. al. 2023. A Unified Executors Proposal for C++ | P0443R14 — open-std.org. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0443r14.html. [Accessed 03-08-2023].

[9] Rob Farber. 2016. *Parallel programming with OpenACC*. Newnes, USA.

[10] Philipp Grete, Forrest W Glines, and Brian W O'Shea. 2020. K-athena: a performance portable structured grid finite volume magnetohydrodynamics code. *IEEE Transactions on Parallel and Distributed Systems* 32, 1 (2020), 85–97.

[11] Muhammad Haseeb, Nan Ding, Jack Deslippe, and Muaaz Awan. 2021. 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)*. IEEE, St. Louis, MO, 68–78.

[12] Hartmut Kaiser, Patrick Diehl, Adrian S Lemoine, Bryce Adelstein Lelbach, Parsa Amini, Agustín Berge, John Biddiscombe, Steven R Brandt, Nikunj Gupta, Thomas Heller, et al. 2020. Hpx-the c++ standard library for parallelism and concurrency. *Journal of Open Source Software* 5, 53 (2020), 2352.

[13] Eric Niebler. 2023. GitHub - NVIDIA/stdexec: 'std::execution', the proposed C++ framework for asynchronous and parallel programming. — github.com. https://github.com/nvidIA/stdexec/. [Accessed 03-08-2023].

[14] Ruyman Reyes and Victor Lomüller. 2016. SYCL: Single-source C++ accelerator programming. In *Parallel Computing: On the Road to Exascale*. IOS Press, USA, 673–682.

[15] Christian Robert Trott, David S Hollman, Daniel Sunderland, Mark Frederick Hoemmen, Carter Edwards, and Bryce Adelstein-Lelbach. 2019. *mdspan in C++: A Case Study in the Integration of Performance Portable Features into International Language Standards*. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States); Sandia ....

[16] Christian R Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S Hollman, Dan Ibanez, et al. 2021. Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 805–817.

[17] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, et al. 2019. AMReX: a framework for block-structured adaptive mesh refinement. *The Journal of Open Source Software* 4, 37 (2019), 1370.