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::execution 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 mspan [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 containers 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 (∼10× H2D) - instead of single transfer 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


