Investigating Performance, Portability, and Productivity with a Simple Computational Fluid Dynamics Solver

Matthew A. Smith
ms3408@york.ac.uk
University of York
York, UK

Steven A. Wright
steven.wright@york.ac.uk
University of York
York, UK

ABSTRACT

Heterogenous systems containing hardware accelerators are the driving force behind the continuing increases in computing power today. However, heterogeneity is forcing developers to write propriety code for accelerators or risk underutilising the potential of the most-powerful systems currently available. This has led to the development of a number of programming models and frameworks focused on *performance portability* that aim to deliver performance across multiple platforms from a single source.

In this paper we study the performance, portability, and productivity of six different approaches currently available to developers, across six different hardware platforms. In particular, we explore three programming models with limited portability – CUDA, HIP, and MPI – and three programming models that promise extensive portability – Kokkos, and two variants of the OpenMP standard, prescriptive and descriptive. Our results highlight that the best performance on each platform is typically achieved by an approach with limited portability, and that portable approaches require divergent code branches in order to remain competitive.

1 INTRODUCTION

High performance computing (HPC) is of significant importance to the scientific community, with scientific grand challenges routinely being tackled by the most computationally powerful systems in the world. Heterogeneous systems containing a diverse range of hardware are driving this performance push with the three systems to break the ExaFLOP performance barrier – El Capitan, Frontier, and Aurora – all heterogeneous in architecture each with GPUs from two separate vendors.

Achieving high performance on these GPUs often requires vendor-specific programming models to be used by developers, however use of these may harm portability between vendors, or require duplication and specialisation of code, at the expense of developer effort [17]. This dilemma has given rise to new parallel programming models and frameworks such as Kokkos [5] which aim to provide portability between heterogeneous architectures from a single-source codebase. In turn, these parallel programming models and frameworks have been the subject of a number of studies, seeking to quantify the *performance portability* [10] and *developer productivity* [6] of these approaches across a diverse range of hardware [1, 7, 11, 13].

In this paper, we add to this corpus, providing a comparison of the performance, portability, and productivity (P3) of six approaches to scientific application development across six hardware platforms. Our study is based on a simple computational fluid dynamics (CFD) application, written in C and parallelised with CUDA, HIP, MPI, Kokkos, OpenMP 4.0+ prescriptive directives, and OpenMP 5.0+ descriptive directives. We analyse these approaches using appropriate metrics and visualisations from the P3 community [8].

Specifically, this paper makes the following contributions:

- We outline the parallelisation of a simple CFD application using six different parallel programming models, including the portability-focused Kokkos programming model, and two forms of OpenMP directives;
- We evaluate each implementation on a range of CPU and GPU hardware typically found in some of the largest systems available today;
- Finally, we analyse each implementation using the metrics of performance, portability and productivity.

2 BACKGROUND AND RELATED WORK

This paper focuses on the performance, portability, and productivity of various approaches to developing parallel software, referred to in some literature as "the three Ps" [8]. Like a number of previous studies, our analysis uses the Φ performance portability metric [10], and code divergence [6].

$$\Phi(a,p,H) = \begin{cases} \dfrac{|H|}{\sum\limits_{i \in H} \dfrac{1}{e_i(a,p)}} & \text{if } i \text{ supported } \forall i \in H \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

The performance portability (Eq. (1)) of an application a, solving problem p, on a given set of platforms H, is calculated by finding the harmonic mean of an application's efficiency ($e_i(a,p)$). Efficiency is a measure of the achieved performance against the best recorded (possibly non-portable) performance on each individual platform (i.e. *the application efficiency*), or the achieved performance against the theoretical maximum performance achievable on each platform (i.e. *the architectural efficiency*). Applications which do not run on all platforms achieve a score of zero.

$$\mathrm{CD}(a,p,H) = \binom{|H|}{2}^{-1} \sum_{\{i,j\} \in H \times H} d_{i,j}(a,p) \tag{2}$$

Code divergence (Eq. (2)) is a measure of the average "distance" between the source code required to compile an application a, and execute problem p for each pair of platforms in H, where $d_{i,j}(a,p)$ is any distance metric between two source codes.

$$d_{i,j}(a,p) = 1 - \frac{|c_i(a,p) \cap c_j(a,p)|}{|c_i(a,p) \cup c_j(a,p)|} \tag{3}$$

In this paper, we adopt the Jaccard distance (Eq. (3)), as suggested by Pennycook et al. [8], where $c_i(a,p)$ represents the set of source lines required to compile application a and execute problem p on a given platform *i*. Higher code divergence suggests additional developer effort, writing and maintaining platform-specific code.

Additionally, we visualise performance portability and developer productivity using *cascade plots* [12] and *P3 navigation charts* [8], created from recorded performance data using the P3 Analysis Library [9].

2.1 Previous P3 Studies

There are many studies evaluating the performance, portability, and productivity of different programming models, languages, and compilers [1, 4, 11]. P3 studies play an important role in the development of scientific applications as they demonstrate to developers how different approaches to a problem compare to one another, therefore helping guide development direction towards the best suited model or framework for the problem [14, 15].

Asahi et al. evaluate the performance portability of a Vlasov code using OpenACC, OpenMP, and Kokkos across a range of CPU and GPU platforms [1]. Deakin et al. track the efficiency of various approaches to developing performance portable software over a number of years, showing the growing maturity of various software stacks [4]. Rangel et al. evaluate the SYCL programming model, and include an evaluation of productivity, in addition to performance portability, showing that SYCL can deliver portability with minimal code divergence between platforms [11]. We build upon these studies through the evaluation of Kokkos and two forms of the OpenMP standard, against the MPI, CUDA, and HIP programming models.

3 PORTABLE APPLICATION DEVELOPMENT

Our study is based on a C implementation of Step 11 of Barba and Forsyth's "12 steps to the Navier-Stokes equations" [2]. The application solves a lid-driven cavity problem in 2D, and is implemented in approximately 600 lines of code. The simplicity of the microbenchmark makes it an ideal candidate for rapid evaluation of different programming models.

Our initial implementation distributes the square domain in one-dimension across processors with MPI, where each rank holds a ghost row above and below that is exchanged between ranks after each step. All other implementations operate over the entire domain, instead expressing thread parallelism. The MPI implementation is restricted to CPU architectures only, though could be used in as a communication layer in a hybrid implementation.

3.1 CUDA and HIP

CUDA and HIP provide interfaces to make use of NVIDIA and AMD GPUs, respectively. While these are two distinct programming models, their APIs are similar.

In these implementations, device kernels replace for-loops, where the kernels contain the body of the loop. All loops within the implementation are implemented in this fashion; while small single-dimension loops provide almost no speed-up, it is crucial this work is still executed on device to reduce the need to copy data between the host and device memory spaces. Further, all copying between arrays is completed through pointer swapping to eliminate the need for kernels or `cudaMemcpy`/`hipMemcpy` operations to do so.

3.2 OpenMP

OpenMP is a compiler directive based multi-threading standard originally developed to take advantage of increasingly multi-core CPUs. Recent versions of the standard (4.0+) increase the potential for portability, by adding support for offloading work to accelerator devices such as GPUs. The OpenMP implementations produced in this study incorporate both CPU and GPU offload directives.

As with the CUDA/HIP implementation, the majority of the computational work is within for-loops where each iteration is independent of others; these loops are therefore ideal candidates for OpenMPs worksharing directives. The majority of these loops are decorated with the `omp parallel for` directive to share iterations among available resources. Data is kept resident where possible using the `enter data` and `exit data` directives.

The most recent OpenMP standard contains two approaches to expressing parallelism: *prescriptive* where the developer explicitly tells the compiler every action to take; and, *descriptive* where the developer tells the compiler something can be done in parallel and the compiler decides what actions to take for a specific platform. Both approaches achieve the same goal of parallelism however may produce different results on the metrics of performance, portability, and developer productivity. This study includes an implementation in each form with the aim of providing a more comprehensive insight into application development using OpenMP.

3.2.1 Prescriptive Model. The prescriptive approach to OpenMP requires separate directives for CPU and GPU targets in order to achieve the best performance. The pre-processor then selects the appropriate directives based on the target architecture.

The CPU implementation simply employs the `omp parallel for` directive on each for-loop to divide the iterations among threads.

The offload implementation centres around the `omp target teams distribute parallel for` directive to share the iterations across GPU threads, additionally a `collapse` clause is employed on multidimensional loops to ensure the inner loop iterations also benefit from parallelism.

3.2.2 Descriptive OpenMP. The descriptive approach to OpenMP development centres around the `omp loop` directive introduced in the 5.0 standard which informs the compiler that loop iterations can be executed concurrently, with the compiler deciding the parallel implementation for a given platform target.

In our implementation, all for-loops are marked with the `omp loop` directive, even though some loops show little benefit from parallelisation. However, as before it is important that all of the computation happens on device to avoid expensive memory copy operations between device and host. As with the prescriptive approach, multi-dimensional loops are collapsed to maximise the amount of parallelism available.

At the time of writing, support for the loop directive varies between compilers. In this study, we use a range of compilers in order to target different architectures – in some cases our descriptive OpenMP implementation is not supported.

3.3 Kokkos

The final approach we evaluate in this study is Kokkos [5] – a performance portability focused programming model for C++ applications. Kokkos provides infrastructure to define multidimensional

```
Kokkos::TeamPolicy<> tpol(N-1, Kokkos::AUTO);
Kokkos::parallel_for("Work", tpol, KOKKOS_LAMBDA(const
↪  Kokkos::TeamPolicy<>::member_type &team_member) {
   int i = team_member.league_rank() + 1;
  Kokkos::parallel_for(Kokkos::TeamThreadRange(team_member,
   ↪  d_nx() -1), [=](int j) {
      completeWork(u(i,j));
   });
});
```

Figure 1: Hierarchical parallelism use for CPUs in Kokkos

arrays that can be mapped to CPUs or accelerators through the `Kokkos::View` data type, and provides functionality to define parallel for-loops using `Kokkos::parallel_for`. These for-loops can be single dimensional using a `Kokkos::RangePolicy`, or multidimensional using a `Kokkos::MDRangePolicy` (similar to collapsing a parallel for-loop in OpenMP).

In our implementation, we select the appropriate range policy for each loop, except on CPU targets where the OpenMP backend is used. For this case, we makes use of hierarchical parallelism through thread teams, as shown in Figure 1, as this was found to provide better performance.

Kokkos requires that the parallel backend is specified at compile time for a given platform target. In this paper, we use the OpenMP backend for CPUs, and the CUDA and HIP backends for NVIDIA and AMD GPUs, respectively.

4 PERFORMANCE, PORTABILITY, AND PRODUCTIVITY

Each experiment is run using exactly one GPU or one node containing two CPUs. We evaluate each implementation using a 1000×1000 problem size, over 5000 iterations, in double-precision. The runtime measurement reported includes only the main computation, ignoring all setup and teardown.

In this study we use a diverse set of hardware available on local cluster systems, as listed in Table 1. For the CPU platforms, we use GCC and OpenMPI, except in the case of the OpenMP descriptive approach, where we use NVHPC; for the GPU platforms, we use the vendor-provided NVHPC and AMD Clang compilers. Optimal values for application configuration variables are selected for each individual platform, based on an exhaustive search of possible options. These include block and grid dimensions for GPUs, and number of threads or MPI processes for CPUs.

4.1 Performance

Figure 2 shows the runtime for each of the six implementations across six hardware platforms. For the two CPU platforms, MPI provides the lowest runtime. On the Intel CPU, the descriptive OpenMP approach is more competitive with MPI, while both the Kokkos and prescriptive OpenMP implementations provide a similar runtime within 1.4× of the lowest. Differences in performance are less pronounced on the AMD CPU, with all three portable approaches within 1.14× of MPI.

Across the GPUs, the best performance is achieved by either the vendor-specified HIP/CUDA programming models, or by Kokkos,

Table 1: Hardware and compilers used in this study

Platform	Hardware	Architecture	Compilers
AMD EPYC 7643	CPU	Zen3	GCC, NVHPC
Intel Xeon Platinum 8468	CPU	Sapphire Rapids	GCC, NVHPC
NVIDIA A40	GPU	Ampere	NVHPC
NVIDIA H100 PCIe	GPU	Hopper	NVHPC
AMD MI300X	GPU	GFX942	AMD Clang
AMD Instinct MI210	GPU	GFX90A	AMD Clang

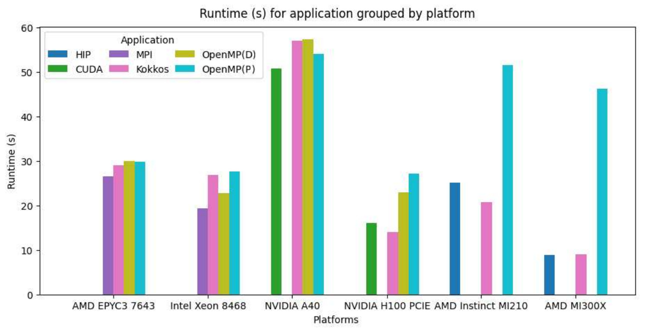


Figure 2: Achieved runtime of each implementation using a 1000×1000 grid size, grouped by platform

however the difference between Kokkos and the native programming models is negligible. On the NVIDIA A40 platform, all approaches are within 1.13× of the lowest runtime, with the general performance lagging other platforms due to the reliance on double-precision calculations not well supported by the hardware. The H100 performs significantly better, with Kokkos and CUDA providing low runtimes. Both approaches to OpenMP run successfully on this platform, with the descriptive approach being approximately 1.5× slower than the fastest approach, and the prescriptive approach being almost 2× slower. For both AMD GPUs, HIP and Kokkos perform similarly. Compiler support for descriptive OpenMP is lacking at the time of writing, while the prescriptive OpenMP approach adds considerable overhead (up to 5.2× slower).

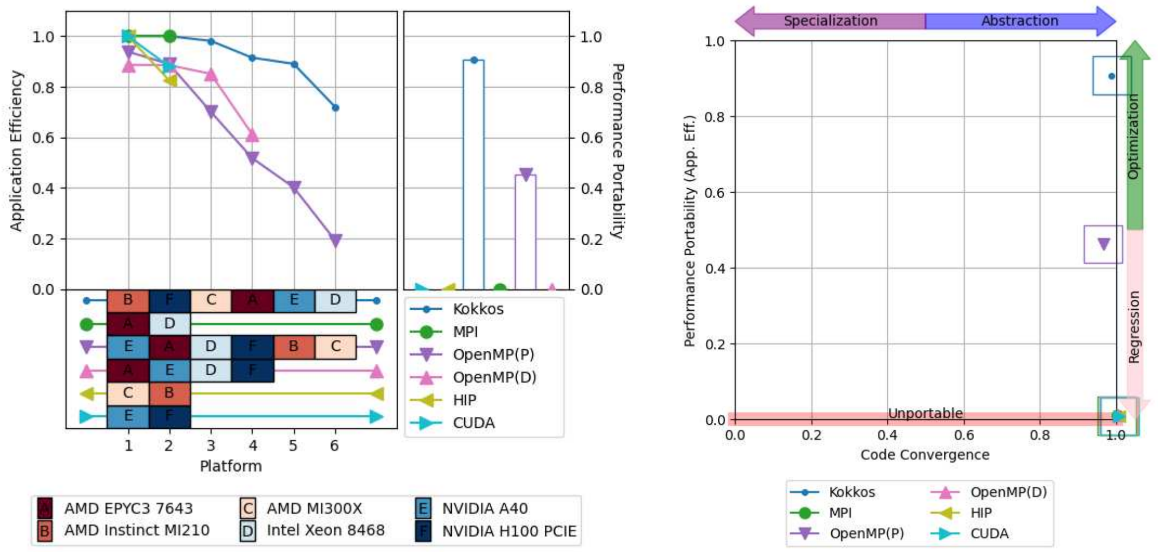
4.2 Portability and Performance Portability

Figure 3(a) presents a cascade plot visualising the performance portability of each approach to parallel application development.

Only two of the implementations outlined in this paper are *portable* according to the Pennycook metric [10]; namely, Kokkos and the prescriptive approach to OpenMP. Kokkos achieves a Φ value of 0.91, while prescriptive OpenMP achieves a value of 0.45.

Kokkos performs reasonably well across all platforms, with the biggest losses of application efficiency coming from the NVIDIA A40 GPU and Intel Xeon CPU platforms. The low Φ value of the prescriptive OpenMP implementation can be attributed to poor GPU performance, particularly on the AMD GPUs; their inclusion more than halves the Φ value, taking it from 0.72 without AMD GPUs to the final value of 0.45 when all platforms are included. This may be partially attributable to the maturity of the NVHPC compiler toolchain, compared to the AMD Clang compiler.

The descriptive OpenMP implementation offers promise as a performance portable approach, achieving a Φ value of 0.79 when the AMD GPUs are not included in the evaluation set. At the time



(a) Cascade Plot **(b) P3 Navigation Chart**

Figure 3: P3 charts for the implementations used in this study

of this study, the `omp loop` directives are only fully supported by the NVHPC compiler, which is unable to target AMD GPUs.

Two of the remaining three approaches are vendor-specified programming models, and so provide excellent performance on their target architectures but limited portability. Finally, the MPI implementation lacks portability outside of CPU targets; this is to be expected as MPI focuses on communication between processes, rather than offloading computation to accelerators.

4.3 Productivity

Developer productivity is difficult to assess objectively. In this paper, we use code divergence as a measure of the number of lines of architecture-specific source code required to achieve high performance; maintaining divergent code branches in order to target different architectures indicates developer overhead. Figure 3(b) shows a P3 navigation chart, plotting the performance portability against code divergence.

In each of our implementations, the computational work is fixed, code divergence occurs only in how parallelism is achieved. Four of our six implementations have no code divergence; the MPI, CUDA, HIP, and descriptive OpenMP implementations all target their specific platforms from a single codebase, though none are able to target all six platforms. Both fully portable implementations contain a fraction of divergence.

The prescriptive OpenMP implementation has the highest divergence of 0.04 as different compiler directives are used to target CPUs and GPUs, with additional directives present for GPU targets to move data on and off device.

While it is possible for us to target both CPUs and GPUs from an identical Kokkos implementation, we instead opt for different execution policies for CPU and GPU targets in order to achieve the highest performance; our Kokkos implementation therefore contains a small divergence of 0.02.

Our results suggest that Kokkos is an excellent choice for developers who are willing to accept a minimal loss of performance (compared to proprietary implementations) for portability across a wide range of platforms. However, divergent code may still be required in order to achieve the highest possible performance across different architectures.

5 CONCLUSION

At the time of writing, it is clear that heterogeneity is likely to be the norm in the near future, having already pushed computing power beyond the ExaFLOP barrier on El Capitan, Frontier, and Aurora. However, the movement towards ever more heterogenous systems brings with it new challenges for application developers, challenges which this study of performance, portability, and productivity has set out to explore.

In this paper, we have outlined the implementation of six different development approaches to a computational fluid dynamics application, collected runtime results for each across a diverse range of hardware, and applied the metrics of performance, portability, and productivity to these results. Our results show promise for performance portability in both Kokkos and OpenMP, but also highlight deficiencies with regards to productivity and compiler support.

In particular, Kokkos shows great performance across all platforms, close to the best for each platform. However, our Kokkos implementation does require divergent parallel execution strategies for optimal performance.

We also provide results for two alternative approaches to OpenMP, showing that the descriptive form of parallelism (`omp loop`) typically outperforms the prescriptive form, but lacks compiler support for AMD platforms. Conversely, the prescriptive form can target all architectures, but requires divergent directives in order to achieve high performance on both CPU and GPU platforms. With improved compiler support we expect the descriptive approach to rival Kokkos in terms of achieving high performance portability.

5.1 Future Work

This work has evaluated CPUs from Intel and AMD, and GPUs from NVIDIA and AMD. Although this covers a wide range of the hardware in use today, the inclusion of Intel GPUs would ensure all GPUs present in the top 5 most powerful systems are included.

Furthermore, this work features only OpenMP and Kokkos as *performance portable* parallel programming models. However, there are multiple alternative frameworks such as RAJA [3] and SYCL [16] that may provide similar or better portability and/or productivity. Including an evaluation of these frameworks could enhance our analysis further.

REFERENCES

[1] Yuuichi Asahi, Guillaume Latu, Julien Bigot, and Virginie Grandgirard. 2021. Optimization strategy for a performance portable Vlasov code. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 79–91. https://doi.org/10.1109/P3HPC54578.2021.00011

[2] Lorena Barba and Gilbert Forsyth. 2019. CFD Python: the 12 steps to Navier-Stokes equations. *Journal of Open Source Education* 2, 16 (2019), 21. https://doi.org/10.21105/jose.00021

[3] 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, 71–81.

[4] Tom Deakin, Andrei Poenaru, Tom Lin, and Simon McIntosh-Smith. 2020. Tracking Performance Portability on the Yellow Brick Road to Exascale. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 1–13. https://doi.org/10.1109/P3HPC51967.2020.00006

[5] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216.

[6] Stephen Lien Harrell, Joy Kitson, et al. 2018. Effective Performance Portability. In *IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 24–36. https://doi.org/10.1109/P3HPC.2018.00006

[7] Wei-Chen Lin, Tom Deakin, and Simon McIntosh-Smith. 2021. On Measuring the Maturity of SYCL Implementations by Tracking Historical Performance Improvements. In *International Workshop on OpenCL* (Munich, Germany) *(IWOCL'21)*. Association for Computing Machinery, New York, NY, USA, Article 8, 13 pages. https://doi.org/10.1145/3456669.3456701

[8] S. John Pennycook et al. 2021. Navigating Performance, Portability, and Productivity. *Computing in Science & Engineering* 23, 5 (2021). https://doi.org/10.1109/MCSE.2021.3097276

[9] S. John Pennycook et al. 2023. Performance, Portability and Productivity Analysis Library. 10.5281/zenodo.7733678. https://doi.org/10.5281/zenodo.7733678

[10] S. J. Pennycook, J. D. Sewall, and V. W. Lee. 2019. Implications of a metric for performance portability. *Future Generation Computer Systems* 92 (2019). https://doi.org/10.1016/j.future.2017.08.007

[11] Esteban Miguel Rangel, Simon John Pennycook, Adrian Pope, Nicholas Frontiere, Zhiqiang Ma, and Varsha Madananth. 2023. A Performance-Portable SYCL Implementation of CRK-HACC for Exascale. In *IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*.

[12] Jason Sewall, S. John Pennycook, Douglas Jacobsen, Tom Deakin, and Simon McIntosh-Smith. 2020. Interpreting and Visualizing Performance Portability Metrics. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 14–24. https://doi.org/10.1109/P3HPC51967.2020.00007

[13] Wageesha Shilpage and Steven A. Wright. 2023. An Investigation into the Performance and Portability of SYCL Compiler Implementations. *Lecture Notes in Computer Science (LNCS)* 13999 (Aug. 2023).

[14] Matthew Smith, Steven A. Wright, Zaman Lantra, and Gihan R. Mudalige. 2025. The P3 Explorer:An Open Database of Performance, Portability, and Productivity. https://doi.org/10.1109/PDP66500.2025.00079

[15] Matthew A. Smith, Steven A. Wright, Zaman Lantra, and Gihan R. Mudalige. 2024. The P3 Explorer: Exploring the Performance, Portability, and Productivity Wilderness. In *The IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis (SC24)*.

[16] The Khronos SYCL Working Group. 2020. SYCL 2020 Specification. https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf

[17] Steven A. Wright, Christopher P. Ridgers, Gihan R. Mudalige, Zaman Lantra, Josh Williams, Andrew Sunderland, H. Sue Thorne, and Wayne Arter. 2024. Developing performance portable plasma edge simulations: A survey. *Computer Physics Communications* 298 (2024). https://doi.org/10.1016/j.cpc.2024.109123