T/AW088/22
Software Support Procurement

Report 2067270-TN-02-04

Developing an Exascale-Ready Fusion Simulation
Revision 7.0

Steven Wright, Chris Ridgers, Ed Higgins, Ben Dudson, Peter Hill, and David Dickinson

University of York

Gihan Mudalige, Zaman Lantra, Ben McMillan, and Tom Goffrey

University of Warwick

December 5, 2023

Contents

Glossary

AVX Advanced Vector eXtensions

CFD Computational Fluid Dynamics

DIMM Dual In-line Memory Module

DRAM Dynamic Random Access Memory

DSL Domain Specific Language

eDSL Embedded Domain Specific Language

FLOP/s Floating point operations per second

FPGA Field Programmable Gate Array

HBM High Bandwidth Memory

ILP Instruction Level Parallelism

ISA Instruction Set Architecture

JIT Just-in-time Compilation

MCDRAM Multi-Channel DRAM

N-1 N processes writing data to a single file

N-N N processes writing data to their own files

N-M N processes writing to M files

PCIe Peripheral Component Interconnect Express

SIMD Single-instruction, multiple-data

SMT Simultaneous multi-threading

SPMD Single-program, multiple-data

SSE Streaming SIMD Extensions

SVE Scalable Vector Extensions

Changelog

September 2023

- Journal Review Paper has been submitted for consideration in Computer Physics Communications [1]. To prevent duplication and drift, Section 2 and Section 3 have been replaced with reference to the journal paper submission. These sections will contain any required additions in future. The review paper is based on the previous version of this report, with the following additions:
  - Updated discussion on general purpose programming models, with new data on usage of Fortran, C and C++ on ARCHER2 and within the US Department of Energy.
  - Added discussion on Asynchronous Many Tasking (AMT) frameworks (e.g. Charm++, LEGION, etc.).
  - Added OCCA to discussion of parallel programming models.
  - Added discussion on partitioning libraries.
  - Added a section on DSLs for Particle-based applications.
  - Added a section on Coupling Frameworks (in collaboration with STFC).
  - Updated information on assessing productivity (in addition to performance and portability).
  - Provided table of various performance studies (some of which are used in this report).

March 2023

- Updated the Evaluation Methodology section to include a reference to the new P3 Analysis Library, and the new plot style. Removed references to the box plots (which arguably add little information over the cascade plots).
- Added stdpar and Thrust to discussion on programming models, since they are evaluated for vlp4d.
- Updated data for Heat to include evaluations on Intel HD P630.
- Regenerated all cascade plots to use the new style of plot using the P3 Analysis Library.
- Added data and analysis for the vlp4d mini-application.
- Added data for mini-fem-pic taken from previous report, along with mention of the OP-PIC DSL.

November 2022

- Added some new apps of interest for evaluation (NESO and vlp4d).
- Added section regarding validation of mini-applications against parent applications – to be built upon in future iterations.

- Clarified that hipSYCL uses LLVM-based backend
- Added results for miniFE using different SYCL compilers, gathered by Shilpage et al.
- Added link to repository of apps and results under the ExCALIBUR-NEPTUNE github.

July 2022

- Addressed all reviewer comments from previous submission.
- Added Heat mini app to evaluation set.
- Included link to a repository containing all mini-apps and results.

March 2022

- Reorganisation of document, combining elements of the previous four reports, 2047358-TN-01, 2047358-TN-02, 2047358-TN-03 and 2047358-TN-04 into a single report on software approaches.
- Described new applications for evaluation, though these have not yet been evaluated.

★ **Note:** A portion of the work in this report has been submitted as a review paper to Computer Physics Communications. Consequently, we have removed much of the duplicated work and instead refer the reader to the submitted manuscript.

[1] Steven A. Wright, Christopher Ridgers, Gihan Mudalige, Zaman Lantra, Josh Williams, Andrew Sunderland, Sue Thorne, and Wayne Arter. Developing Performance Portable Simulations for the Design of a Nuclear Fusion Reactor. *Computer Physics Communications*, (Under Review) 2023

1 Context

In 2008 Roadrunner became the first supercomputer to break the PetaFLOP/s barrier. Roadrunner was an AMD Opteron powered system with PowerXCell accelerators connected to each core, making it one of the first *modern* heterogeneous systems. This heterogeneous approach has continued ever since, with a growing proportion of the fastest supercomputers in the world making use of highly-specialised computational accelerators (e.g. GPUs) alongside traditional multi-CPU hosts; and this trend looks set to continue as we cross the ExaFLOP/s barrier.

The emergence of computational accelerators has been coupled with a golden age of architectural developments [2]. Many of the systems likely to be available in the next decade will employ hierarchical parallelism, delivered by a diverse set of architectures [3, 4]. With each architecture potentially requiring a different programming model and different optimisation strategies, developing software that is portable across systems is becoming increasingly difficult.

For most large scientific simulation applications, maintaining multiple versions of a code-base is simply not a reasonable option given the significant time and effort, not to mention the expertise required. Even with multiple versions, it does not guarantee a future-proof application where the next innovation in hardware may well require yet another parallel programming model to obtain best performance for the new device. These challenges are now general and applicable equally to any scientific domain that relies on numerical simulation software using HPC systems. As a recent review for applications in the computational fluid dynamics (CFD) domain [5] elucidates, three key factors can be identified when considering the development and maintenance of large-scale simulation software, particularly aimed at production:

1. **Performance:** running at a reasonable/good fraction of peak performance on given hardware.
2. **Portability:** being able to run the code on different hardware platforms/architectures with minimum manual modifications
3. **Productivity:** the ability to quickly implement new application, features and maintain existing ones.

Over the years, attempts at developing a general programming model that delivers all three has not had much success. Auto-parallelising compilers for general purpose languages have consistently failed [6]. Compilers for imperative languages such as as C/C++ or Fortran, the dominant languages in HPC, have struggled to extract sufficient semantic information, enabling them to safely parallelise a program from all, but the simplest structures. Consequently, the programmer has been forced to carry the burden of "instructing" the compiler to exploit available parallelism in applications, targeting the latest, and purportedly greatest, hardware.

In many cases, the use of very low-level techniques, some only exposed by a particular programming model/language extension are required with careful orchestration of computation and communications to obtain the best performance. Such a deep understanding of hardware is difficult to gain, and even more so unreasonable for domain scientist/engineers to be proficient in – especially given that the expertise required rapidly

changes with the technology of the moment following hardware trends. A good example is the many-core path originally touted by Intel with accelerators such as the Xeon Phi which has been discontinued – the first US Exascale systems will now all be GPU based, with two systems containing AMD GPUs, and one containing Intel GPUs.

As such, it is near impossible to keep re-implementing large science codes for various architectures. This has led to a *separation of concerns* approach where the description of what to compute is separated from how the computation is implemented. This is in direct contrast to languages such as C or Fortran, which explicitly describe the computation.

1.1 Project NEPTUNE

The NEPTUNE (NEutrals & Plasma TUrbulence Numerics for the Exascale) project is concerned with the development of a new computational model of the complex dynamics of high temperature fusion plasma. It is an ambitious programme to develop new algorithms and software that can be efficiently deployed across a wide range of supercomputers, to help guide and optimise the design of a UK demonstration nuclear fusion power plant. The goal of the *code structure and coordination* work package within NEPTUNE is to establish a series of "best practices" on how to develop such a next-generation simulation application that is *performance portable*.

In this report, we aim to review and evaluate the key approaches and tools currently used to develop new numerical simulation applications targeting modern HPC architectures and systems, including methods of re-engineering existing codes to modernise them. We focus on applications from the plasma fusion domain and related supporting applications from engineering. Our aim is to survey and present the state-of-the-art in achieving "performance portability" for Fusion, where an application can achieve efficient execution across a wide range of HPC architectures without significant manual modifications.

As many of the applications, libraries and programming models used in this report are under active development, the data presented here is subject to change. New data is being collected and analysed all the time, and will be updated in the future where necessary. This document should therefore be considered a living document, reflecting the current state of performance portable application development focused on applications of interest for the simulation of plasma physics.

The remainder of this report is organised as follows:

Section 2 outlines the method of evaluating performance portability that will be taken throughout this report;

Section 3 discusses current approaches to performance portable scientific application development;

Section 4 describes the applications that will be used to evaluate the performance portability of various approaches to software development;

Section 5 provides evaluation data for each of these applications, and evaluates the performance portability of the various implementations;

Section 6 analyses the approaches to Exascale application development with reference to the evaluation data;

Section 7 concludes this report, providing recommendations for the NEPTUNE project.

2 Evaluation Methodology

There are numerous methods for evaluating the performance and portability of high-performance parallel applications. The review paper associated with this report [1] highlights a number of these in Section 8, along with methods to evaluate *productivity*.

In this report we focus on the performance portability of applications using the metric introduced by Pennycook et al. [7], and use the visualisation techniques outlined by Sewall et al. [8]. In some cases, where only a single implementation is available, we will use *architectural efficiency* rather than *application efficiency*. In these constrained cases, we augment our analysis with Roofline analysis [9].

3 Approaches to Exascale Application Development

Figure 1 gives a broad outline of the various components that may be involved when developing multi-physics simulation applications for execution on heterogeneous systems. Higher-level representations of physics problems (such as DSLs) allows an application to better synthesise machine-code representations for various hardware, and potentially enables more developer productivity (in many DSLs partial differential equations can be represented directly in code). Lower-level representations are more likely to be able to exploit available parallelism on various platforms, but may limit portability between systems.

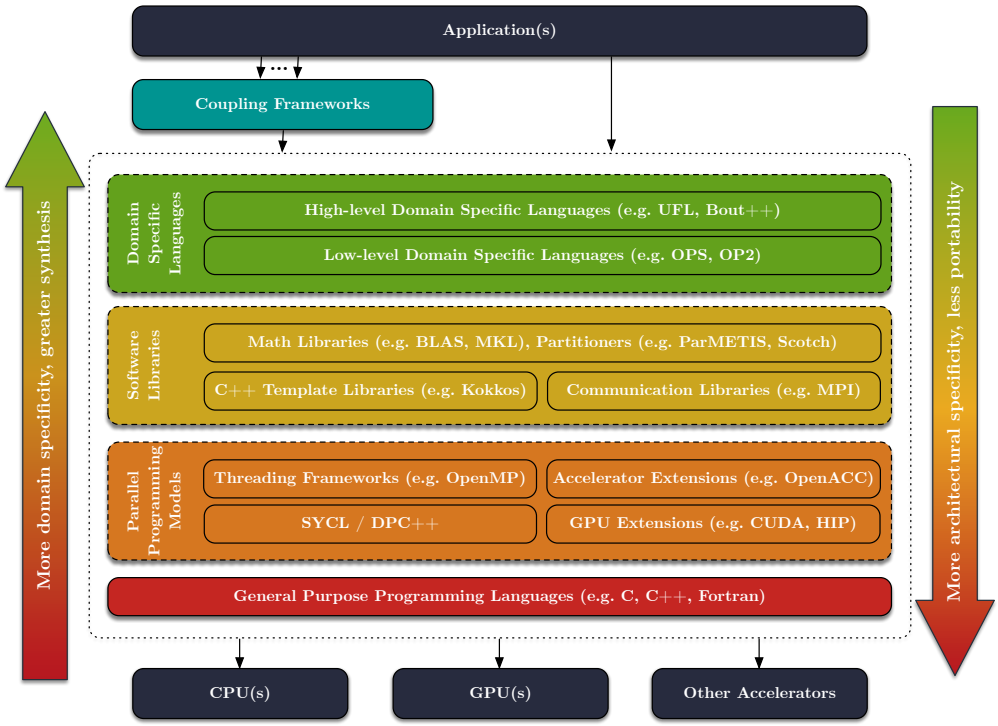


Figure 1: Overview of the potential layers in a software stack for a multi-physics simulation application.

In our review paper [1], we cover the five levels in Figure 1, in Sections 3–7; focusing on the current state-of-the-art in (i) general purpose programming languages, (ii) parallel programming models, (iii) software libraries, (iv) domain specific languages, and (v) coupling frameworks.

In future, this section will be used to address any additions to the approaches discussed.

4 Applications for Evaluation

The exploratory stage of NEPTUNE includes a number of projects that are investigating the behaviour of plasmas through proxy applications. The applications currently being used broadly fall in to two categories, *fluid models* and *particle models*. In particular, T/NA078/20 used Nektar++ to explore the performance of spectral elements, T/NA083/20 focused on building fluid referent models in both Bout++ and Nektar++, and T/NA079/20 explored particle methods with the EPOCH particle-in-cell (PIC) code. It is therefore likely that the resultant NEPTUNE software stack require both fluid and particle components with a coupling interface between.

The three aforementioned applications are the result of many years of development and typically consist of many thousands of lines of C/C++ or Fortran. They are already widely used by the UK's scientific computing community on a diverse range of problems.

Prior to the development of the NEPTUNE software stack, it is prudent to assess the wide range of available technologies, without the associated burden of redeveloping these mature simulation applications into new programming frameworks. In this project, we use a series mini-applications that implement key computational algorithms that are similar to those used by the NEPTUNE proxy applications. These mini-applications are typically limited to a few thousand lines of code and are often available implemented in a wide range of programming frameworks already.

Notable collections of such mini-applications includes Rodinia [10], UK-MAC [11], the NAS Parallel Benchmarks [12], the ECP Proxy Apps [13] and the SPEC benchmarks [14]. In this section we will discuss the applications we have identified from these benchmark suites that implement computational kernels similar to those required by NEPTUNE.

4.1 Fluid Models

As previously noted, the fluid modelling aspects of the NEPTUNE project are largely focused on the use of **Bout++** [15, 16] and **Nektar++** [17]. Bout++ is a framework for writing fluid and plasma simulations in curvilinear geometry, implemented using a finite-difference method, while Nektar++ is a framework for solving computational fluid dynamics problems using the spectral element method.

Both applications are large C++ applications designed primarily for execution across homogeneous clusters. Parallelisation across a cluster in both applications is achieved using MPI, with Bout++ additionally capable of on-node parallelism with OpenMP. GPU acceleration is under development in both applications, through RAJA and HYPRE in Bout++, and through OpenACC in Nektar++ [18].

Rather than redevelop these applications, this project has instead identified a series of mini-applications that implement similar computational schemes. Specifically, we have identified a small number of finite difference and finite element mini-apps, each of which are implemented in a range of programming models for rapid evaluation of approaches to performance portability.

Heat

Heat is a simple finite-difference application developed at the University of Bristol for their OpenMP Target training course. Besides OpenMP and OpenMP target, it has also been ported to SYCL[1].

TeaLeaf

TeaLeaf is a finite difference mini-app that solves the linear heat conduction equation on a regular grid using a 5-point stencil. It has been used extensively in studying performance portability already [19–22], and is available implemented using CUDA, HYPRE, OpenCL, PETSc and Trilinos[2].

miniFE

miniFE is a finite element mini-app, and part of the Mantevo benchmark suite [9,23–25]. It implements an unstructured implicit finite element method and is available implemented using CUDA, Kokkos, OpenMP and OpenMP with offload[3], and SYCL[4].

Laghos

Laghos is a mini-app that is part of the ECP Proxy Applications suite [25–27]. It implements a high-order curvilinear finite element scheme on an unstructured mesh. It uses HYPRE for parallel linear algebra, and is additionally available in CUDA, RAJA and OpenMP implementations[5].

vlp4d

The vlp4d mini-app is a 2+2D Vlasov-Poisson equation solver, based on the 5D plasma turbulence code, GYSELA [28]. It is implemented in C++ and has been augmented with OpenMP, OpenACC, MPI, Kokkos, Thrust, CUDA, HIP and C++ `stdpar`[6].

In future reports we will expand this evaluation set to include the following applications:

FDTD3D

FDTD3D is an implementation of Yee's method for solving Maxwell's equations, implemented as part of the OpenCL examples library, provided by NVIDIA. There are available implementations in CUDA, HIP, OpenMP and SYCL[7].

Maxwell

The Maxwell mini-app is distributed as part of the MFEM library. Since it is implemented using the MFEM library, it can target any programming model supported by MFEM[8].

hipBone

The hipBone mini-app is a GPU port of the Nekbone application. It is implemented in C++, and leverages the OCCA performance portability framework [29] to provide portability to OpenMP, CUDA and HIP[9].

[1] `https://github.com/UoB-HPC/heat_sycl`
[2] `http://uk-mac.github.io/TeaLeaf/`
[3] `https://github.com/Mantevo/miniFE`
[4] `https://github.com/zjin-lcf/oneAPI-DirectProgramming/tree/master/miniFE-sycl`
[5] `https://github.com/CEED/Laghos`
[6] `https://github.com/yasahi-hpc/P3-miniapps`
[7] `https://github.com/zjin-lcf/HeCBench`
[8] `https://mfem.org/electromagnetics/`
[9] `https://github.com/paranumal/hipBone`

4.2 Particle Methods

Particle methods in NEPTUNE have been explored using the EPOCH particle-in-cell code [30], its associated mini-app minEPOCH [31][10] and the UKAEA-developed NESO[11] application.

EPOCH is a PIC code that runs on a structured grid, using a finite differencing scheme and an implementation of the Boris push. Like Bout++ and Nektar++, EPOCH is a mature software package that is used widely by the UK science community, and thus is difficult to evaluate in alternative programming models without a significant redevelopment effort. Furthermore, EPOCH is developed in Fortran, making it increasingly difficult to adapt to many new programming models that are heavily based in C++. The mini-app variant of EPOCH, minEPOCH, is likewise developed in Fortran and thus not appropriate for this study.

NESO is a test implementation of a PIC solver developed at UKAEA for 1+1D Vlasov-Poisson. It is written in C++ using DPC++/SYCL for on-node parallelism, while off-node parallelism uses MPI. The field solve is implemented using Nektar++.

Besides NESO, there are a number of other particle-based mini-apps that may be of interest to this project, that implement similar particle schemes, backed by a variety of electric/magnetic field solvers.

CabanaPIC

CabanaPIC is a structured PIC code built using the CoPA/Cabana library for particle-based simulations [25]. Through the CoPA/Cabana library, the application can be parallelised using Kokkos for on-node parallelism and GPU use, and with MPI for off-node parallelism[12].

VPIC/VPIC 2.0

Vector Particle-in-Cell (VPIC) is a general purpose PIC code for modelling kinetic plasmas in one, two or three dimensions, developed at Los Alamos National Laboratory [32]. VPIC is parallelised on-core using vector intrinsics, on-node through pthreads or OpenMP, and off-node using MPI. VPIC 2.0 [33] adds support for heterogeneity, using Kokkos[13].

EMPIRE-PIC

EMPIRE-PIC is the particle-in-cell solver central the the ElectroMagnetic Plasma In Realistic Environments (EMPIRE) project [34]. It solves Maxwell's equations on an unstructured grid using a finite-element method, and implements the Boris push for particle movement. EMPIRE-PIC makes extensive use of the Trilinos library, and uses Kokkos as its parallel programming model [35, 36].

Mini-FEM-PIC

Mini-FEM-PIC is a mini-application that implements a particle-in-cell method on an unstructured mesh, using the finite element method. It was developed as part of this project, and is based on the *fem-pic* application by Lubos Brieda. It is implemented in C++ and can be executed in parallel using OpenMP.

[10] `https://github.com/ExCALIBUR-NEPTUNE/minepoch`
[11] `https://github.com/ExCALIBUR-NEPTUNE/NESO`
[12] `https://github.com/ECP-copa/CabanaPIC`
[13] `https://github.com/lanl/vpic`

Each of the particle-based mini-apps identified implement a PIC algorithm that is similar to that found in EPOCH. However, one weakness of this evaluation set is that three of the four applications are parallelised on-node through the Kokkos performance portability layer. In future reports we will expand this evaluation set to include the following application:

NESO

NESO is a test implementation of a PIC solver for 1+1D Vlasov-Poisson. It is implemented in C++, with DPC++/SYCL parallelism, and a field solve using Nektar++.

Sheath-PIC

Sheath-PIC is a simple 1D GPU implementation from `www.particleincell.com`. It has been ported from CUDA to HIP, OpenMP and SYCL[14].

4.3 Validation

The mini-applications chosen for this study implement only small subsections of larger applications, or algorithms that are similar in their structure. In many cases, they are solving much smaller or much simpler problems and therefore the results are likely not representative of that which is required by NEPTUNE. What is important for this study is that they are *performance representative*.

A number of methods have been explored to validate the representativeness of mini-applications and their parents. In this project, informed by the ECP Project [37], we will adopt **cosine similarity** to compare vectors of performance counter values.

For each application, we will sample the accumulated hardware counters for an entire execution. We will then form an application vector x_i, that contains the averaged hardware event counters for the last 5 seconds of execution. Two applications will be considered *similar* if the vectors that represent the applications are a short distance apart. The cosine similarity is calculated as

$$\cos(\theta) = \frac{\sum_{k=1}^{d} x_{ik} x_{jk}}{\| x_i \| \| x_j \|} \tag{1}$$

The cosine value varies from 1.0 (identical vector direction) to 0.0 (orthogonal vector direction), and the angle θ varies from 0°to 90°. If two applications are performance representative, we expect their cosine similarity angle to be closer to 0°.

Our analysis will be added to a future iteration of this report. In contrast to the ECP report, our analysis will not be based on a parent application and a representative mini-application variant, but instead on generic mini-applications and target parent applications. Because of this, we do not expect our results to conform as closely as those in the original study. Nonetheless, we expect that particular performance-sensitive counters will show the required similarity.

[14] `https://github.com/zjin-lcf/HeCBench`

5 Evaluations of Approaches

In this section we present performance data for a number of mini-applications, across a range of architectural platforms, using a range of different approaches to performance portability.

The applications chosen in each case are broadly representative of some of the algorithms of interest to NEPTUNE. In particular, the fluid-method based mini-apps implement algorithms that range from finite-difference (like Bout++ [16]) to high-order finite element or spectral element (like Nektar++ [17]). Similarly, the particle-methods mini-apps all implement the particle-in-cell method (like EPOCH [30]).

The data presented in this section, and the applications are available on github, through the linking repository: `https://github.com/ExCALIBUR-NEPTUNE/performance-portability-for-fusion`.

5.1 Heat

Heat is a benchmark from the University of Bristol that is used for teaching parallelisation. It is the simplest finite difference application used in this evaluation, and as such is mostly representative of the data access pattern, rather than the compute intensity. The data presented in this section has been collected for a 10000×10000 problem over 1000 time steps on Isambard.

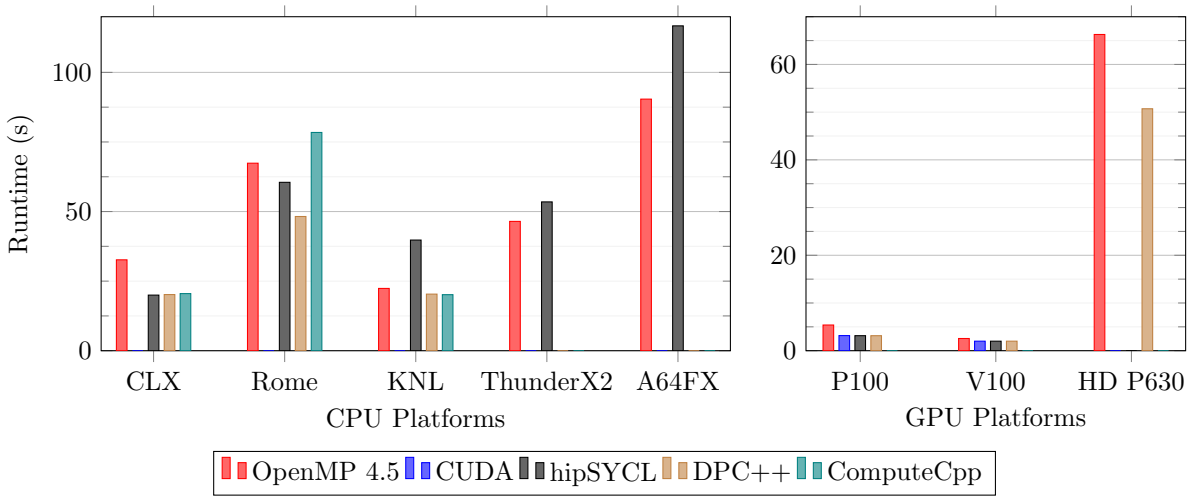


Figure 2: Heat runtime data

5.1.1 Performance

Performance data for the Heat code was collected as part of a project to evaluate three implementations of the SYCL standard. As such, there are three SYCL data points per platform, acquired with Intel's DPC++ compiler, Heidelberg's hipSYCL compiler (through a custom LLVM build), and Codeplay's ComputeCpp compiler. The runtimes achieved with each compiler can be compared to OpenMP with offload and CUDA.

The runtime data for Heat is presented in Figure 2, split for CPU and GPU platforms due to the magnitude difference in runtime on the NVIDIA GPU platforms. From this data, we can see that generally the SYCL runtimes are competitive with the OpenMP and CUDA variants, and in some cases better, regardless of compiler.

The main difference between each compiler is in the level of platform support; hipSYCL is able to target every architecture except the Intel HD P630 GPU, but on KNL and AMD Rome, its performance is worse than the same code compiled by Intel's DPC++ compiler. The ComputeCpp compiler has the worst support, being unable to target the Arm platforms or the GPUs, due to lack of an OpenCL driver.

For the two Arm platforms on Isambard (ThunderX2 and A64FX), the performance in both OpenMP and hipSYCL is relatively poor compared to alternative architectures. However, the overhead of SYCL is reasonably small (15-30% slowdown). For the x86 CPUs and the GPUs, the fastest SYCL variant matches or outperforms the OpenMP with offload variant; on GPUs the CUDA variant is still marginally faster.

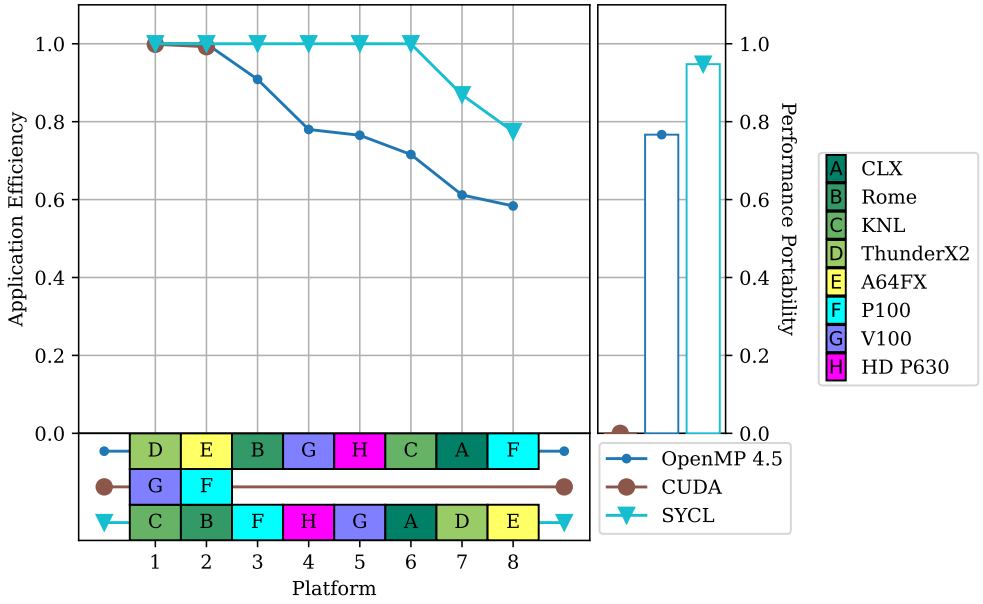


Figure 3: Cascade visualisation of performance portability of Heat

5.1.2 Portability

Figure 3 show the performance portability of the Heat application, where the data for SYCL is taken as the best performing SYCL compiler on each platform.

As can be seen from the right side of the figure, only OpenMP 4.5 and SYCL achieve *performance portability*, with SYCL typically outperforming OpenMP 4.5. Figure 3 additionally shows that as platforms are added to the evaluation set, SYCL achieves near perfect efficiency until the 7th and 8th platforms are added (the two Arm platforms in this case).

Conversely, CUDA shows the lowest portability, only being executable on the two NVIDIA GPU platforms.

As can be seen in Figure 2, DPC++ provides better performance than hipSYCL on the KNL and Rome platforms, highlighting the importance of compiler selection currently. Both the hipSYCL and DPC++ compilers are now based on the LLVM compiler infrastructure, and so it is likely that the performance of each of these compilers will eventually converge.

The simplicity of the Heat code lends itself to rapid porting efforts and so the results are a good indication of what can be achieved by any larger code using the SYCL programming model. However, as will be seen later in this report, larger codes require significantly more re-engineering to achieve similar levels of performance portability in newer programming models such as SYCL.

5.2 TeaLeaf

TeaLeaf is a finite difference mini-app that solves the linear heat conduction equation on a regular grid using a 5-point stencil, developed as part of the UK-MAC (UK Mini-App Consortium) project.

It has been used extensively in studying performance portability already [19–22], and is available implemented using CUDA, OpenACC, OPS, RAJA, and Kokkos, among others[15]. The results in this section are extracted from two of these studies, namely one by Kirk et al. [20] and one by Deakin et al. [19]. In both studies, the largest test problem size (`tea_bm_5.in`) is used, a 4000×4000 grid.

5.2.1 Performance

The study by Kirk et al. shows the execution of 8 different implementations/configurations of TeaLeaf across 3 platforms, a dual socket Intel Broadwell system, an Intel KNL system and an NVIDIA P100 system. The runtime for each implementation/configuration is presented in Figure 4. Note that in the study, some results are missing due to incompatibility (e.g. CUDA on Broadwell/KNL)[16].

The study by Deakin et al. is more recent, using a C-based implementation of TeaLeaf as its base. It consequently evaluates fewer programming models, but over a wider range of hardware, including a dual socket Intel Skylake system, both NVIDIA P100 and V100 systems, AMDs Naples CPU, and the Arm-based ThunderX2 platform. Runtime results are provided in Figure 5.

5.2.2 Portability

Both studies evaluate some portable and some non-portable implementations. In most cases, there is a non-portable implementation that achieves the lowest runtime, however this places a restriction on the hardware that it can target.

[15] `http://uk-mac.github.io/TeaLeaf/`
[16] *Hybrid* represents the best performing configuration of a MPI/OpenMP hybrid execution

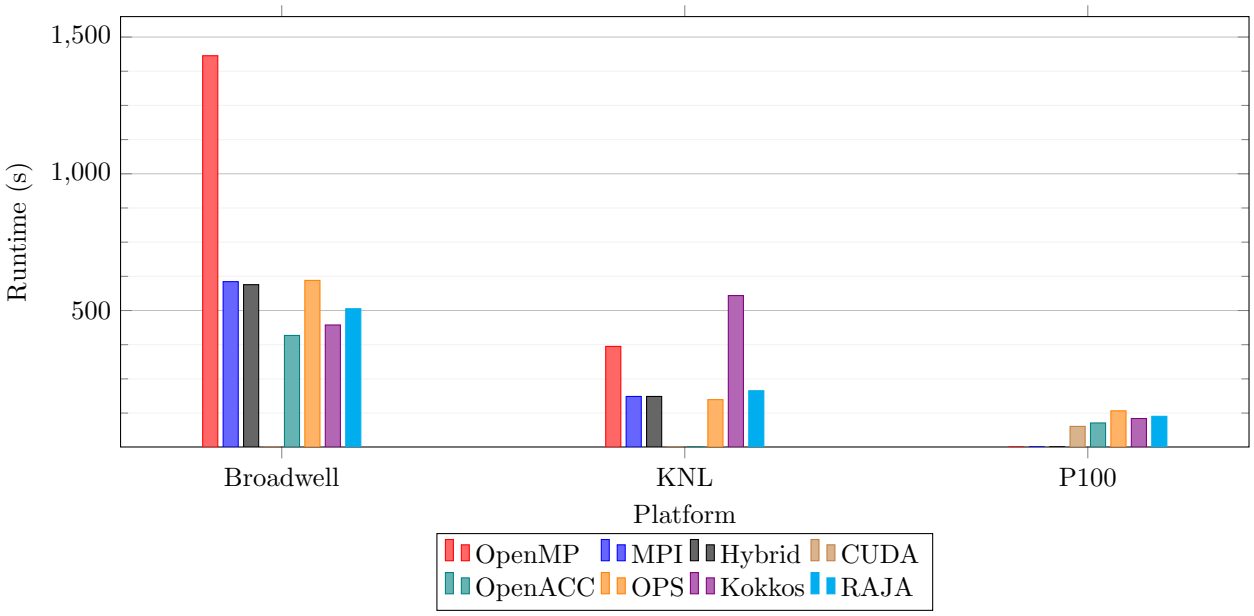


Figure 4: TeaLeaf runtime data from Kirk et al. [20]

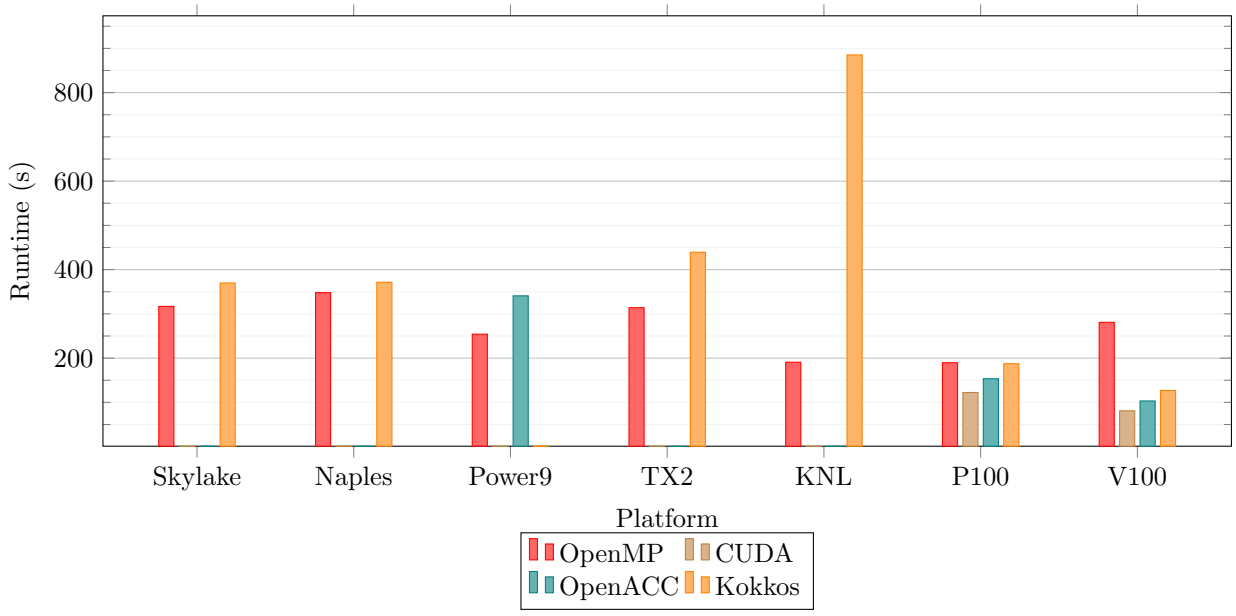


Figure 5: TeaLeaf runtime data from Deakin et al. [19]

For study by Kirk et al. [20], Figure 6 provides a visualisation of the performance portability of each approach to application development. In terms of efficiency, the non-portable approaches (CUDA, MPI, and OpenACC) achieve high efficiency, but do not extend to the full evaluation set, while the portable approaches (Kokkos, OPS and RAJA) span much of the evaluation set, but sacrifice some efficiency. Almost all approaches (except OpenMP) achieve more than 80% application efficiency on at least one platform, and in the case of RAJA and OPS, performance above 60% application efficiency is maintained across the three

platforms. Referring back to Figure 4, we can see that on the Intel KNL system, the Kokkos performance is double that of other performance portable approaches, and thus skews its portability calculation. It is likely that this is the result an unidentified issue in TeaLeaf or Kokkos at the time of evaluation. Otherwise, these three programming models each achieve similar levels of performance and, importantly, portability across different architectures.

Figure 7 show the same visualisations for the data from Deakin et al. [19]. Again, the non-portable programming model (CUDA) achieves the highest performance on its target architectures. For CPU architectures OpenMP produces the highest result, and using offload directives, portability is available to GPU devices. It should be noted that to support the use of GPU devices, there are two OpenMP implementations that must be maintained (with and without offload directives), though these results are presented together here. Much like in the previous study, the performance portability of Kokkos is affected by an anomalous result on the Intel KNL platform.

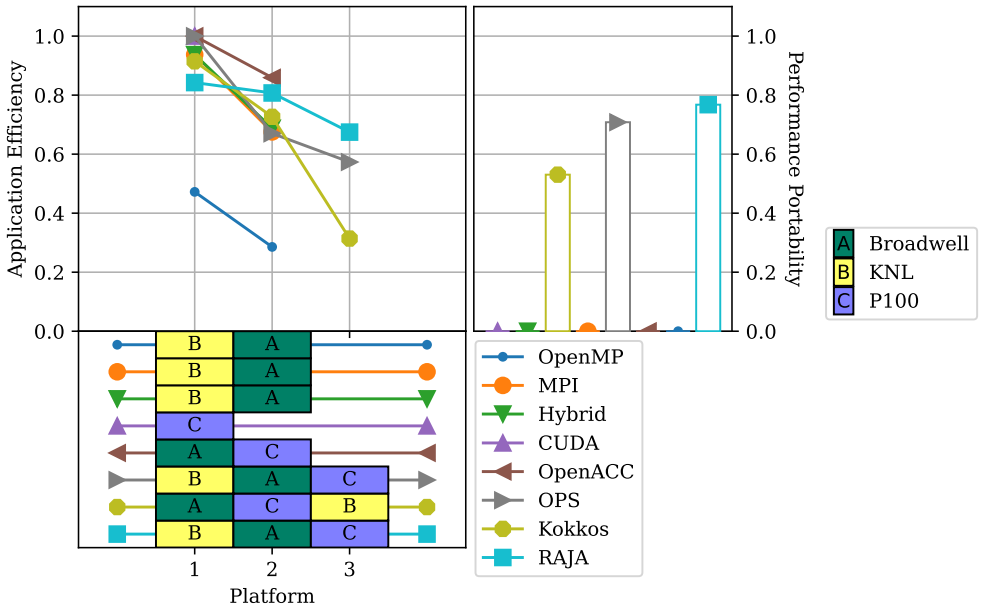


Figure 6: Cascade visualisation of performance portability from Kirk et al. [20]

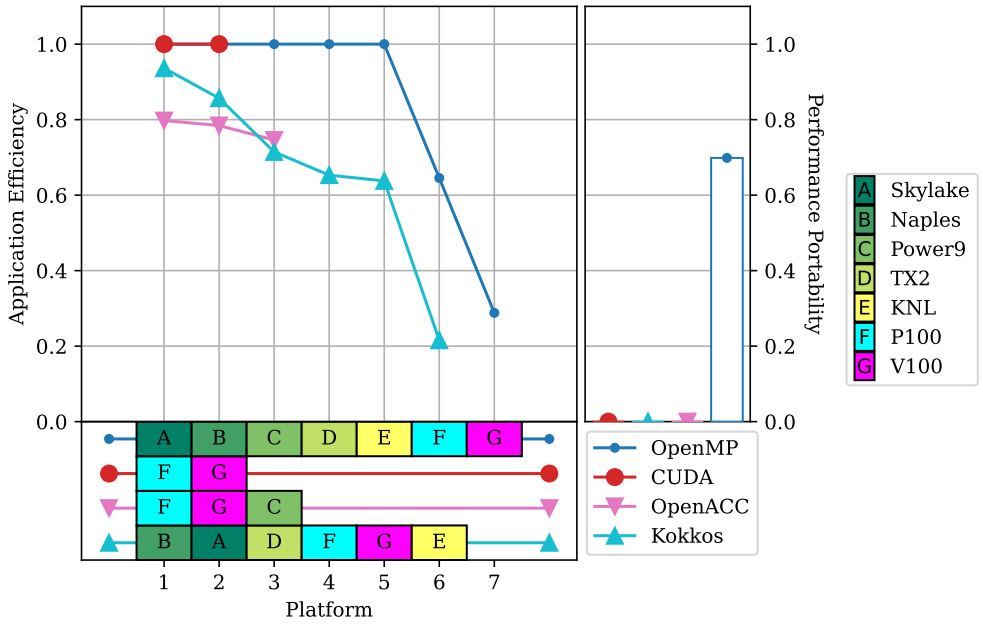


Figure 7: Cascade visualisation of performance portability from Deakin et al. [19]

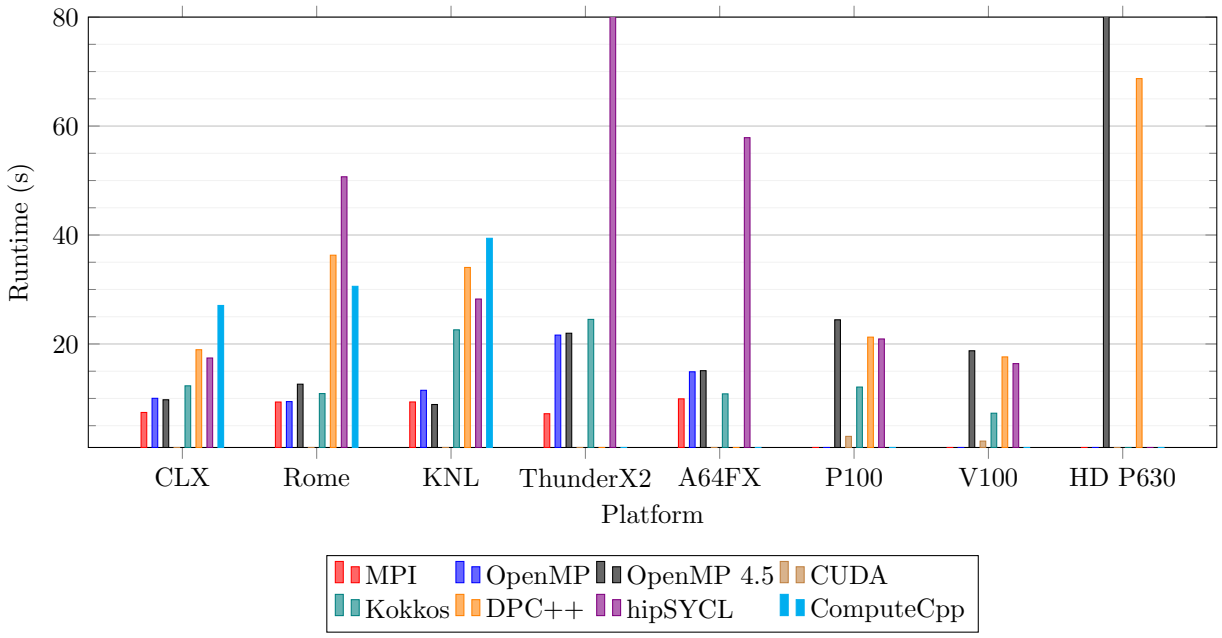


Figure 8: miniFE runtime data gathered in 2022 for a SYCL maturity study[17]

[17]Runtime data above 80s has been clipped. The runtime for hipSYCL on the ThunderX2 platform is 110.921s, while the runtime for the HD P630 through OpenMP 4.5 is 371.287s

5.3 miniFE

miniFE is a finite element mini-app, and part of the ECP Proxy apps (previously the Mantevo benchmark suite) [9, 23–25]. It implements an unstructured implicit finite element method and has versions available in CUDA, Kokkos, OpenMP (3.0+ and 4.5+) and SYCL[18].

While there are a number of data sources for miniFE data, most of these are limited in scope. Instead all data presented in this section has been newly gathered. Previous iterations of this report contained data gathered in 2021, specifically for Project NEPTUNE. In this iteration of the report, new data is presented from a 2022 study into the maturity of SYCL implementations.

In all cases, a $256 \times 256 \times 256$ problem size has been used, and all runs have been conducted on the platforms available on Isambard.

5.3.1 Performance

The raw runtime results for these runs can be seen in Figure 8. In many of the miniFE ports available, only the conjugate solver has been parallelised effectively, so the results presented represent only the timing from this kernel.

It should be noted that the SYCL data is gathered from a miniFE port that can be found as part of the oneAPI-DirectProgramming github repository[19]; this port is based on a conversion from the OpenMP 4.5 implementation of miniFE, and so no SYCL-relevant optimisation has been performed.

The previous data presented in this report contained a number of omissions due to the unavailability of compilers, or other issues. The data presented in this report resolves many of these issues, and additionally includes data for an Intel HD P630 GPU. While this GPU is not optimised for HPC workloads (since it is an embedded GPU), it provides the first glimpse of programmability of Intel's new Xe GPU line.

Figure 8 shows that the SYCL performance and portability depends largely on the compiler that is used. Interestingly, hipSYCL is often the best performing SYCL compiler (even when compared to the Intel DPC++ compiler, on Intel hardware). However, it is clear that there is a SYCL penalty on such a complex code (in contrast to Heat). Given the nature of the miniFE SYCL port, this indicates that achieving high performance for a SYCL code likely requires some optimisation after a conversion.

It is also clear from the data presented in Figure 8 that the native approaches (CUDA, MPI/OpenMP) are typically the fastest. For the two NVIDIA GPU platforms, CUDA is significantly faster than any alternative, whereas for the CPU platforms Kokkos is competitive. For the two ARM platforms (TX2 and A64FX), the SYCL performance is typically poor, likely owing to an issue with the custom LLVM compiler that was required to collect the data.

[18] `https://github.com/Mantevo/miniFE`
[19] `https://github.com/zjin-lcf/oneAPI-DirectProgramming/tree/master/miniFE-sycl`

5.3.2 Portability

Figure 9 presents a visualisation of the performance portability of miniFE, through various approaches. It is clear from the figure that on CPU platforms, MPI is the most performant (achieving nearly 100% efficiency across the 5 CPU platforms), while CUDA is the most performant on the NVIDIA GPUs. For the Intel iGPU, the most performant is SYCL (through DPC++), but the efficiency of SYCL falls away rapidly.

Only OpenMP 4.5 and SYCL are portable across the 8 platforms, but achieve a $\Phi \approx 0.2$. Typically Kokkos outperforms SYCL (except on the Intel iGPU, where Kokkos has not been executed). Unfortunately, all of the "portable" approaches achieve a median efficiency below 50%. This is in contrast to the data presented for the much simpler Heat application, and indicates the need for careful optimisation of the code.

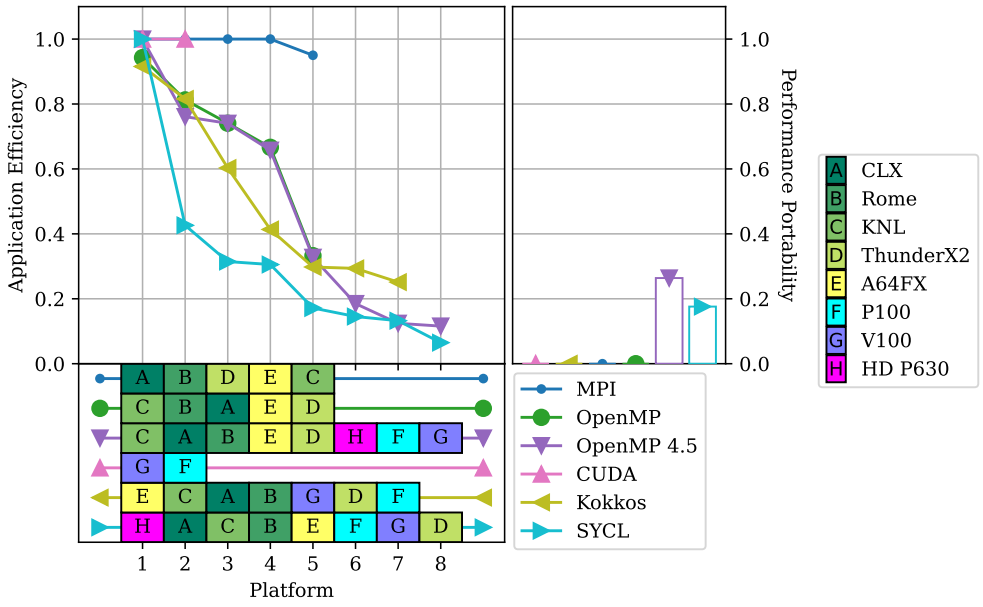


Figure 9: Cascade visualisation of performance portability of miniFE

5.4 Laghos

Laghos is a mini-app that is part of the ECP Proxy Applications suite [25–27]. It implements a high-order curvilinear finite element scheme on an unstructured mesh. The majority of the computation is performed by the HYPRE and MFEM libraries, and can thus use any programming model that is available for these libraries[20].

The results presented in this section have all been collected on the Isambard platform.

[20] `https://github.com/CEED/Laghos`

5.4.1 Performance

Figure 10 shows the runtime for Laghos, running problem #1 (Sedov blast wave), in three dimensions, up to 1.0 second of simulated time, using partial assembly (i.e., `./laghos -p 1 -dim 3 -rs 2 -tf 1.0 -pa -f`).

Across the six platforms evaluated, RAJA performance is typically in line with the fastest non-portable approach (MPI and CUDA). Since the parallelisation in Laghos is in the MFEM and HYPRE shared libraries, that were developed at LLNL alongside RAJA, that these routines are well optimised in RAJA is perhaps not surprising.

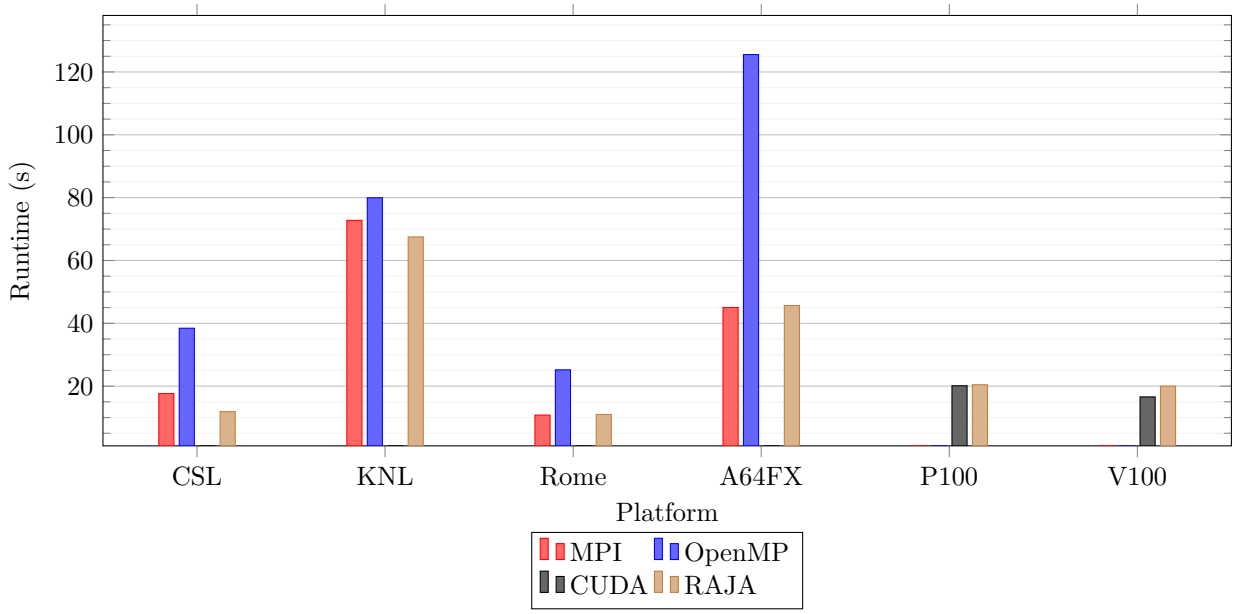


Figure 10: Laghos runtime data

5.4.2 Portability

Figure 11 demonstrates the remarkable efficiency of the RAJA MFEM and HYPRE implementations, showing consistently above 80% performance efficiency. In contrast to some of our previous results, OpenMP performs poorly across most platforms (except KNL). The difference between OpenMP and RAJA on the CPU platforms suggests that either the RAJA parallelisation on these systems is achieved through SIMD and Thread Building Blocks (TBB), or that there are performance issues in the OpenMP implementation. On the GPU platforms, CUDA does marginally outperform RAJA, but this is perhaps to be expected, given the potential overhead in using a third party performance library.

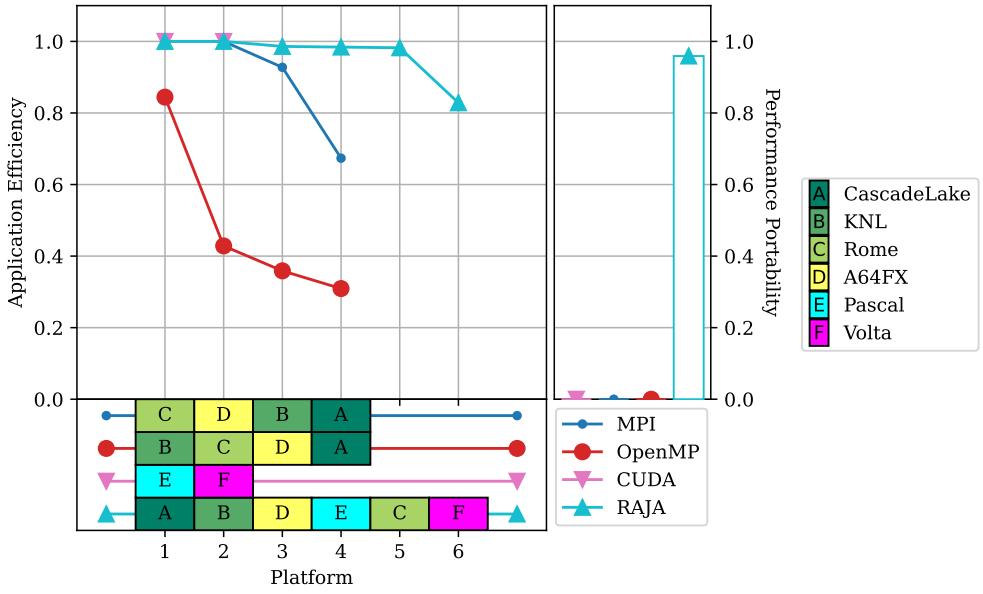


Figure 11: Cascade visualisation of performance portability of Laghos

5.5 vlp4d

The vlp4d application solves the Vlasov-Poisson equations in 4D (2D space and 2D velocity space). It is based on the 5D plasma turbulence code, GYSELA, but is miniaturised specifically for performance portability studies [38].

In this report, we have collected data running the two-dimensional Landau damping problem (SLD10), documented by Crouseilles et al. [39]. We have collected results on the Isambard system, making use of all available architectures (including the Phase 3 system).

5.5.1 Performance

Figure 12 plots the runtime of vlp4d across the 7 programming models, and 9 evaluation platforms. It should be noted that the NVIDIA and AMD GPU platforms are an order of magnitude faster than the CPU platforms, and so are plotted separately.

In the general case, OpenMP and Kokkos achieve similar performance on almost every platform, where Kokkos is marginally better on the Arm and NVIDIA architectures, and marginally worse on the Intel and AMD architectures. The two NVIDIA supported programming models (Thrust and Stdpar) perform very well on the NVIDIA platforms (through the NVHPC compiler), and are also among the best performing programming models across other platforms – though neither extends to the Arm platforms.

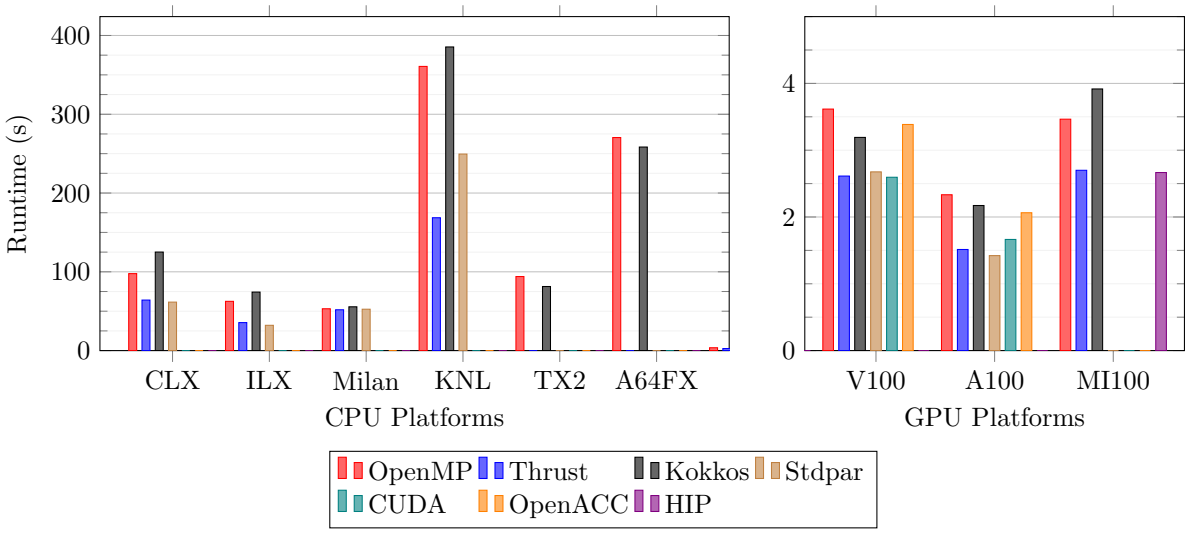


Figure 12: vlp4d runtime data

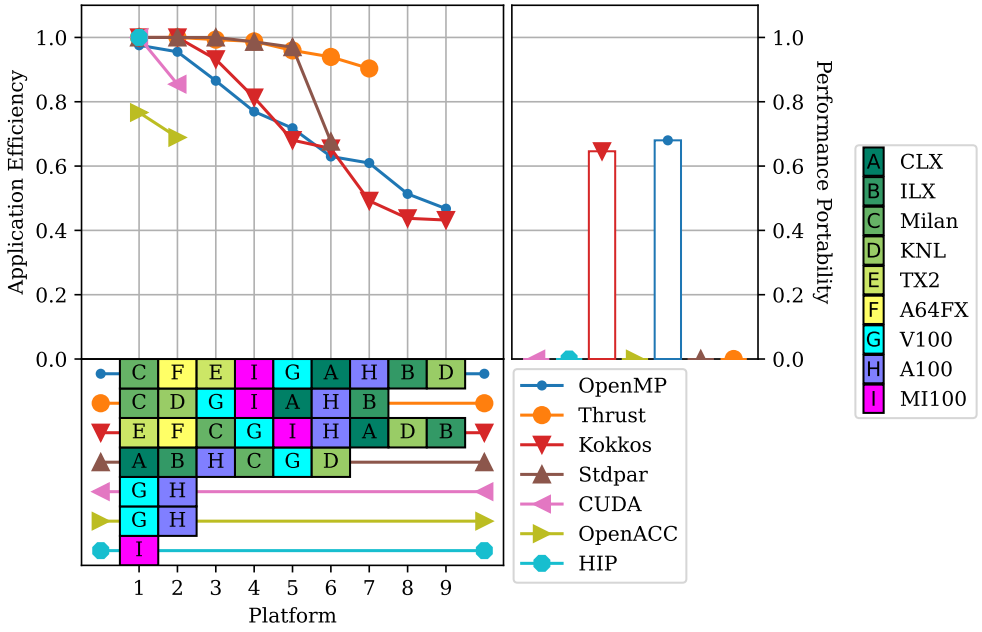


Figure 13: Cascade visualisation of performance portability of vlp4d

5.5.2 Portability

The right of Figure 13 shows that only the OpenMP and Kokkos programming models are portable to every platform evaluated, and they achieve a $\mathcal{P} \approx 0.65$; Thrust is portable to the AMD GPU, but not the Arm platforms, while Stdpar relies on the NVHPC compiler, which does not currently support the AMD or Arm platforms. The Kokkos and OpenMP programming models also show a similar trend of efficiency across platforms (and while the ordering of the platforms is not identical between both, it is similar).

Interestingly, on the A100 platform, CUDA is not the most performant programming model, with both Thrust and Stdpar achieving a lower runtime. On the AMD Instinct MI100, HIP and Thrust both achieve a similar level of performance.

As shown in Figure 13, the highest efficiency across platforms is achieved by the Thrust library and the Stdpar programming model, up to the inclusion of the Arm platforms or the AMD GPU (in the case of Stdpar).

5.6 CabanaPIC

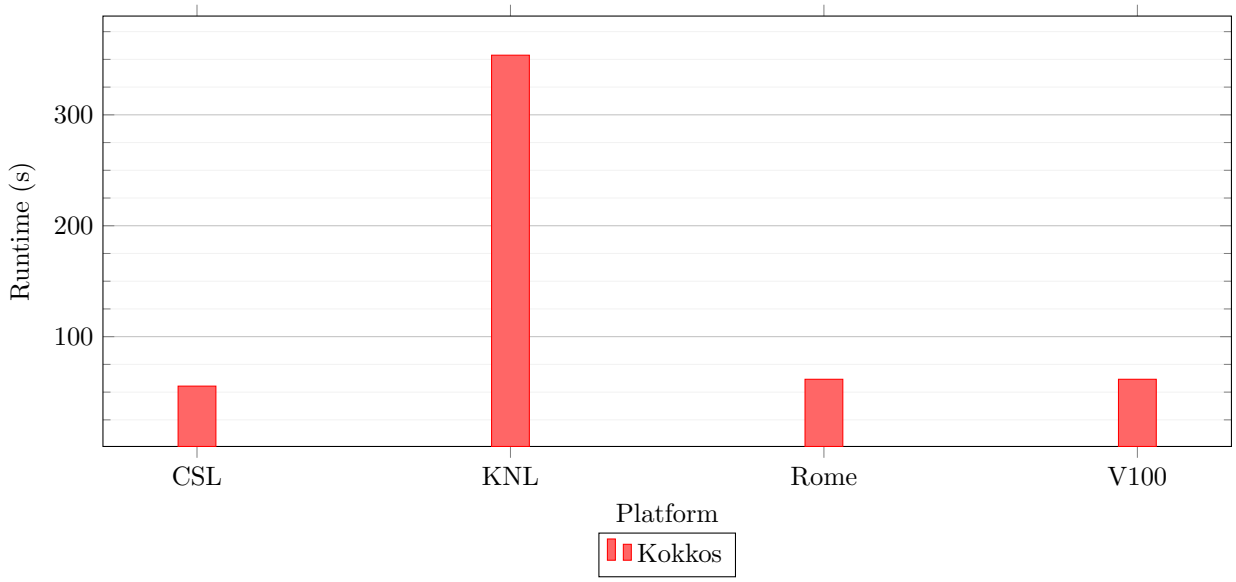


Figure 14: CabanaPIC data

CabanaPIC is a structured PIC demonstrator application built using the CoPA/Cabana [40] library for particle-based simulations [25]. CoPA/Cabana provides algorithms and data structures for particle data, while the remainder of the application is built using Kokkos as its programming model for on-node parallelism and GPU use, and MPI for off-node parallelism[21].

5.6.1 Performance

Since there is only a single implementation of CabanaPIC, it is not possible for us to evaluate how the programming model affects its performance portability, however, we can show how the performance changes between architectures.

Figure 14 shows the achieved runtime for CabanaPIC across four of Isambard's platforms, running a simple 1D 2-stream problem with 6.4 million particles.

[21] `https://github.com/ECP-copa/CabanaPIC`

Approximately equivalent performance can be seen on the Cascade Lake, Rome and V100 systems. Similar to our TeaLeaf Kokkos results on KNL, the runtime is significantly worse than expected, possibly indicating a Kokkos bug, or a configuration issue. Otherwise performance is similar on all platforms in terms of the raw runtime. Given the significantly higher peak performance of the NVIDIA V100 system, it is perhaps surprising that its performance is not significantly better. This may be due to serialisation caused by atomics, or significant data movement between the host and the accelerator; further investigation is necessary to identify this loss of efficiency.

5.7 VPIC

Vector Particle-in-Cell (VPIC) is a general purpose PIC code for modelling kinetic plasmas in one, two or three dimensions, developed at Los Alamos National Laboratory [32]. VPIC is parallelised on-core using vector intrinsics and on-node through a choice of pthreads or OpenMP. It can additionally be executed across a cluster using MPI[22]. The recently developed VPIC 2.0 [33] code has been developed to add support for heterogeneity using Kokkos to optimise the data layout and allow execution on accelerator devices.

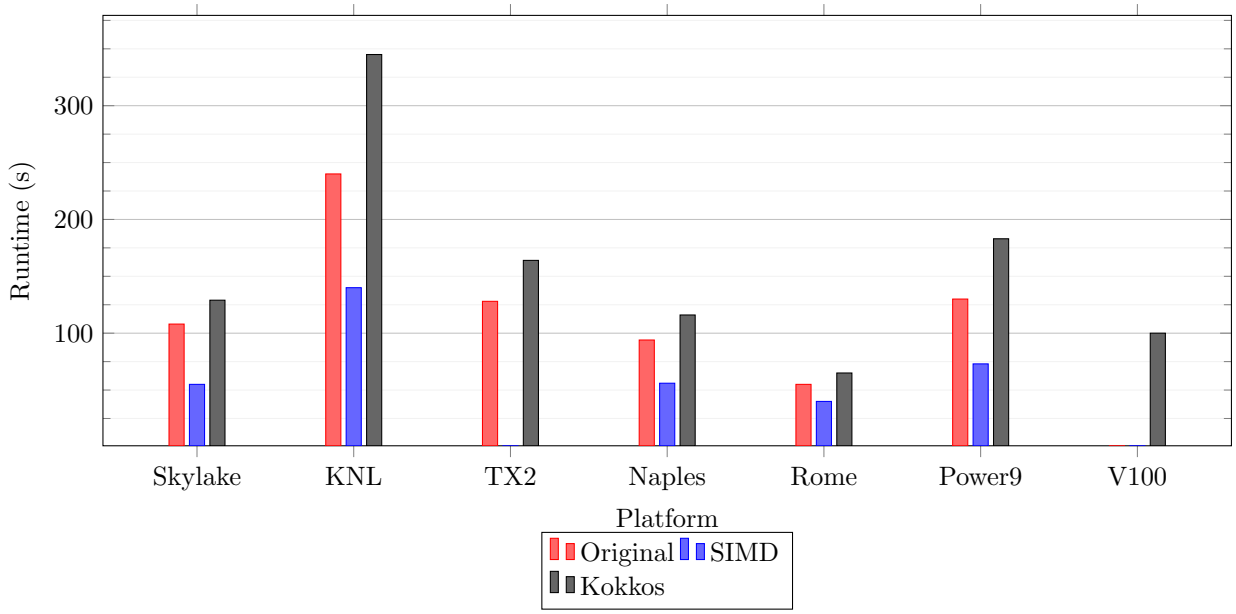


Figure 15: VPIC runtime data from Bird et al. [33]

5.7.1 Performance

Figure 15 shows the runtime for the three variants of the VPIC code running on seven platforms[23]. This data is taken from the VPIC 2.0 study, comparing the non-vectorised, vectorised and Kokkos variants of the VPIC code. In each case, the runtime is the time taken for 500 time steps, with 66 million particles.

[22] https://github.com/lanl/vpic
[23] https://globalcomputing.group/assets/pdf/sc19/SC19_flier_VPIC.pptx.pdf

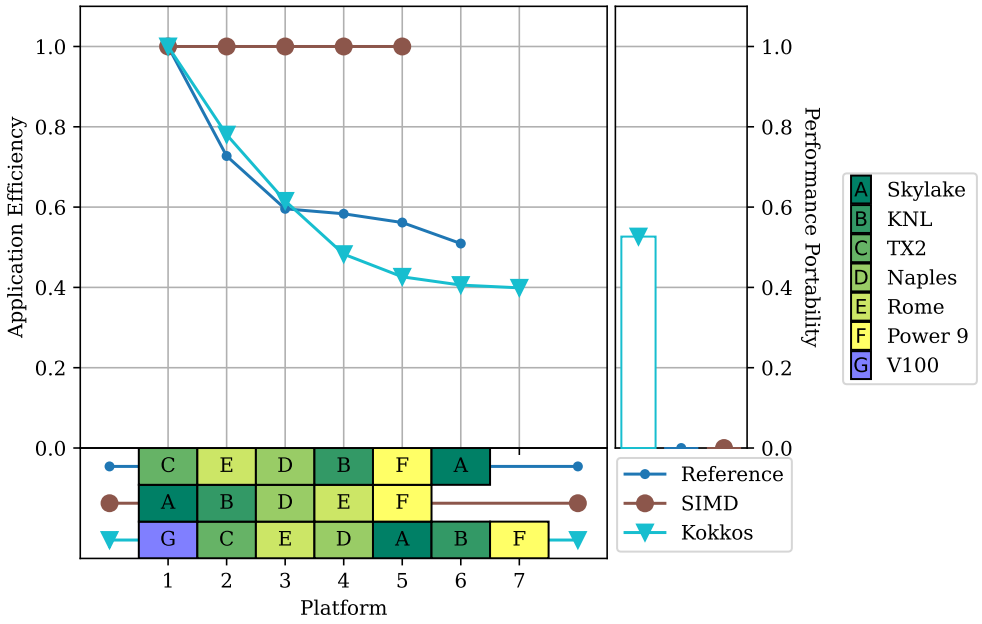


Figure 16: Cascade visualisation of performance portability of VPIC

In Figure 15 we can observe that the SIMD vectorised implementations are always the fastest for each platform, however it should be noted that each of these are hand-optimised for each individual instruction set (i.e. every implementation is platform specific). This means that, alongside the additional coding effort of writing an implementation for each platform, potential additions or fixes must also be applied to all implementation individually, significantly affecting the productivity. While the Kokkos implementation is typically the slowest on each platform, performance is usually in-line with the unvectorised original VPIC application, suggesting that the slowdown is caused by the inability of the compiler to autovectorise.

5.7.2 Portability

In terms of the performance portability of VPIC, we can see that the original and vectorised variants are only viable on the CPU architectures. Figure 16 visualises how the performance portability varies as more platforms are evaluated.

The highest performance on each of the CPU platforms comes from the vectorised variant of VPIC, as it achieves the best performance on all CPU platforms (except the ThunderX2, where no data is provided). However, since it cannot execute on the GPU platform, its performance portability is 0.

Figure 16 shows that while Kokkos performs worse than the vectorised implementation, its performance is similar the non-vectorised variant, but is also capable of execution on the V100 platform.

It should be noted that this data is from a study based on the initial implementation of VPIC using Kokkos. It is likely that these performance figures will be improved in future, potentially closing the performance gap

on the vectorised implementation, while maintaining portability to heterogeneous architectures. Indeed, a recent study presented at the PASC conference [41] has shown that the Kokkos runtime can be improved by up to 55% using Kokkos SIMD[24].

5.8 EMPIRE-PIC

EMPIRE-PIC is the particle-in-cell solver central the the ElectroMagnetic Plasma In Realistic Environments (EMPIRE) project [34]. It solves Maxwell's equations on an unstructured grid using a finite-element method, and implements the Boris push for particle movement. EMPIRE-PIC makes extensive use of the Trilinos library, and subsequently uses Kokkos as its parallel programming model [35, 36].

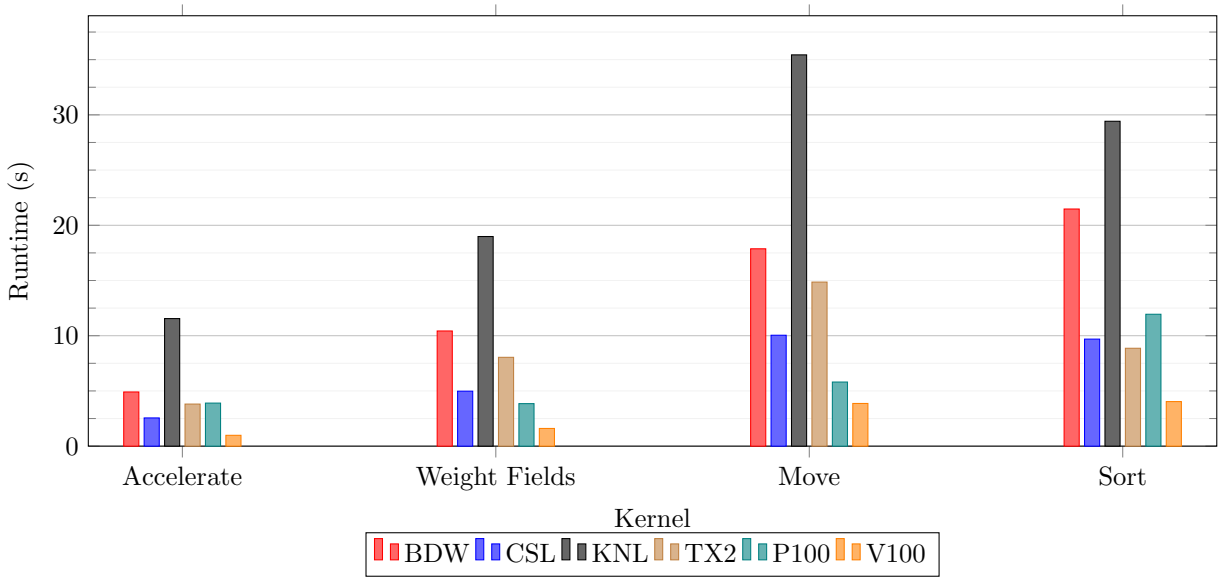


Figure 17: EMPIRE-PIC runtime data

5.8.1 Performance

The EMPIRE-PIC application is export controlled, and thus the results in this section come from the study by Bettencourt et al. [35], looking specifically at the particle kernels within EMPIRE-PIC.

Figure 17 shows the runtime of the Accelerate, Weight Fields, Move and Sort kernels within EMPIRE-PIC for an electromagnetic problem with 16 million particles (8 million H+, 8 million e-). The geometry for this problem is the tet mesh that can be seen in Figure 7 in Bettencourt et al. [35].

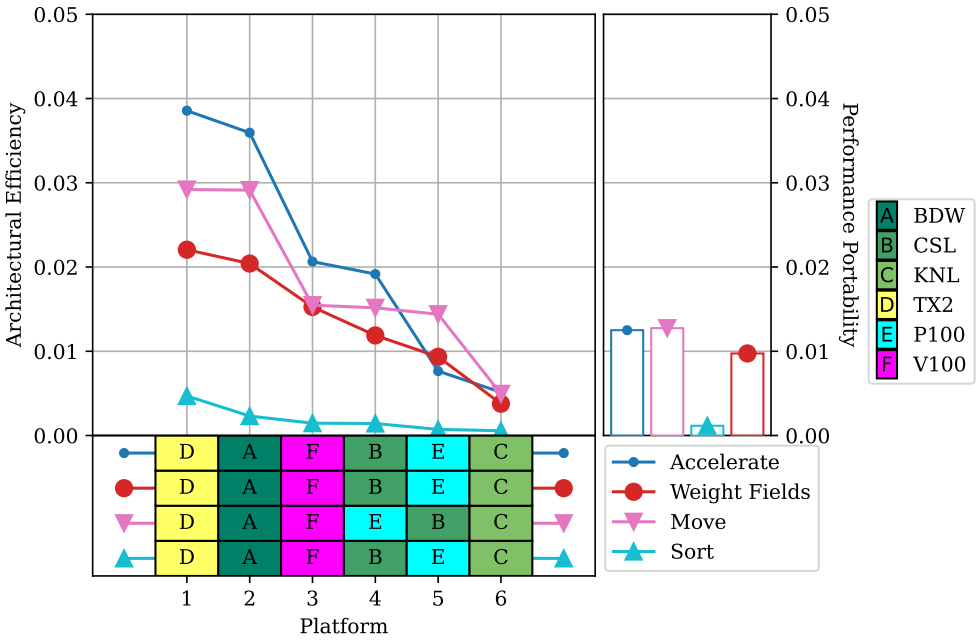


Figure 18: Cascade visualisation of performance portability for four particle kernels in EMPIRE-PIC

5.8.2 Portability

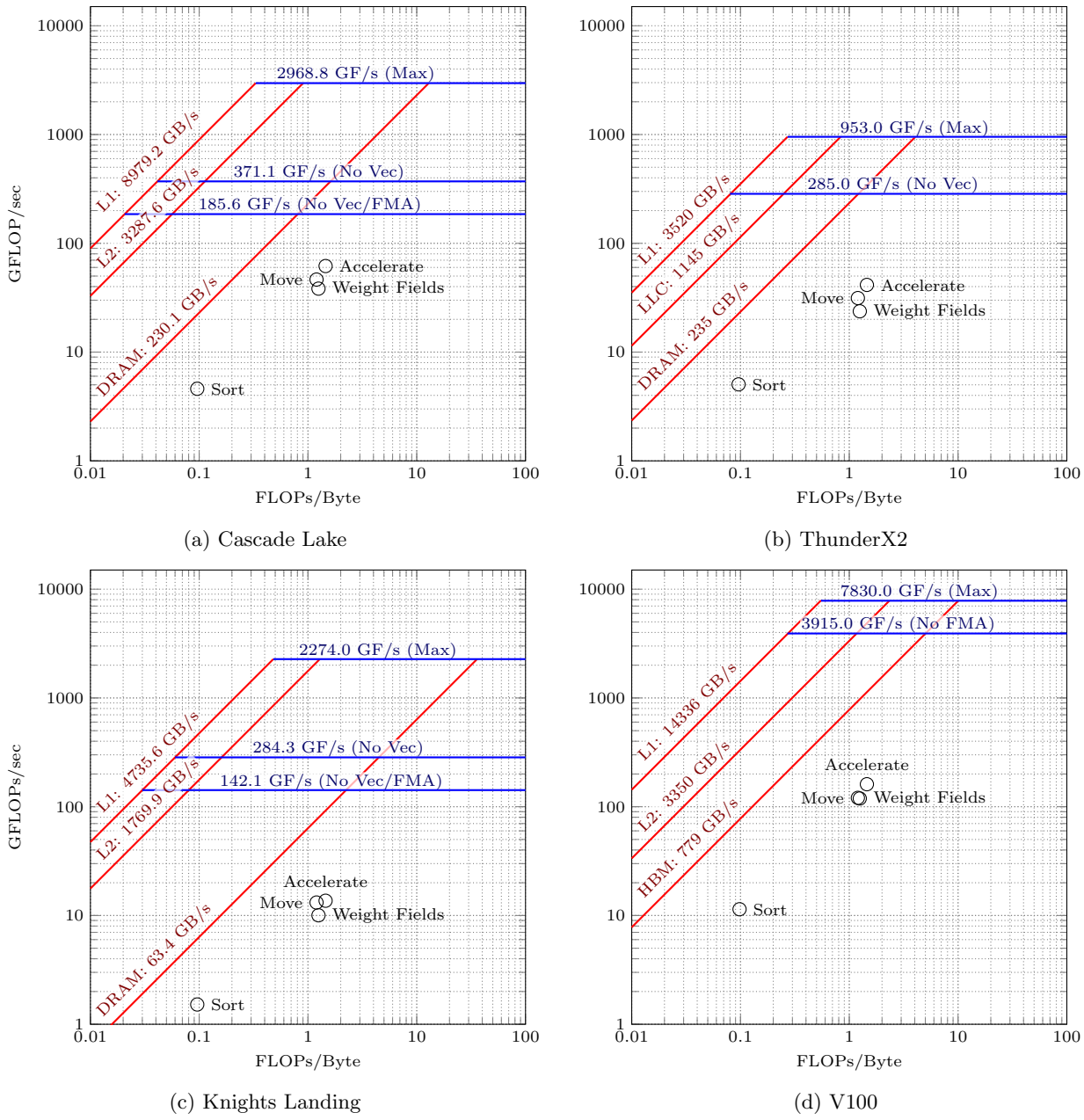
While there is only a single programming model implementation of EMPIRE-PIC, we can use the equations given in Table 2 of Bettencourt et al. [35] to calculate the FLOP/s achieved and compare this to each machines maximum floating-point performance, thus calculating the *architectural efficiency*. The equations presented assume the best case performance, where particles are evenly distributed, there is no particle migration, and they are sorted at the start of the simulation. Nevertheless, they provide an opportunity to analyse the performance portability of Kokkos for particle-based kernels.

Figures 18 provides a visualisation of EMPIRE-PIC's performance portability across six platforms[25].

It is important to note that although Figure 18 shows incredibly low efficiency, this is compared to each platform's peak performance, where a vectorised fused-multiply-add instruction must be executed each clock cycle. Achieving less than 10% of this peak performance is not unusual for a real application. In the case of the Sort kernel, the efficiency is lower still, as this is not a kernel that is bound by floating point performance.

What is clear from Figure 18 is that the variance in achieved efficiency between platforms is not large, indicating that Kokkos is able to achieve a similar portion of the available performance for EMPIRE-PIC's particle kernels. Achieved efficiency is higher on the ThunderX2 and Broadwell systems, due to less reliance on well vectorised code, and a lower available peak performance.

[24]The data in this report will be updated to reflect this in future iterations.
[25]Please note that the y-axis in each of these Figures has been scaled, since the architectural efficiency is very low.



(a) Cascade Lake

(b) ThunderX2

(c) Knights Landing

(d) V100

Figure 19: Roofline plots on four platforms, gathered using the Empirical Roofline Toolkit [42]

The data suggests that EMPIRE-PIC is not able to fully exploit the on-core parallelism available through vectorisation. Figure 19 shows roofline models for four of these platforms, with the four particle kernels plotted according to their arithmetic intensity and achieved FLOP/s.

In all cases, we can see that the application is not successfully using vectorisation (and this is confirmed by compiler reports). As stated in Bettencourt et al. [35], the control flow required to handle particles crossing element boundaries leads to warp divergence on GPUs and makes achieving vectorisation difficult

on CPUs. Nonetheless, on the Cascade Lake and ThunderX2 platforms, we are within an order of magnitude of the non-vectorised peak performance for the three main kernels, and the sort kernel (with low arithmetic intensity) is heavily affected by main memory bandwidth. For the two many-core architectures (KNL and V100), floating-point performance is further from the peak, and the performance of each kernel is further hindered by the DRAM/HBM bandwidth.

Roofline analyses, like Figure 19, are effective at demonstrating how vital to performance it is to balance efficient memory accesses with arithmetic intensity. This is especially important in PIC codes, where some of the kernels are relatively low in arithmetic intensity when compared to the amount of bytes that need to be moved to and from main memory (e.g. the Boris push algorithm requires many data accesses, but performs relatively few mathematical operations). An alternative approach to the FEM-PIC method has been explored using EMPIRE-PIC by Brown et al. [36], whereby complex particle shapes are supported using virtual particles based on quadrature rules. Using virtual particles in this manner increases the arithmetic intensity of particle kernels without requiring significantly more data to be moved from and to main memory.

5.9 Mini-FEM-PIC

The Mini-FEM-PIC application has been redeveloped from the *fem-pic* application, by Lubos Brieda, for this project. Its development is detailed in Report 2057699-TN-03-03, along with some preliminary reports that are partially repeated here.

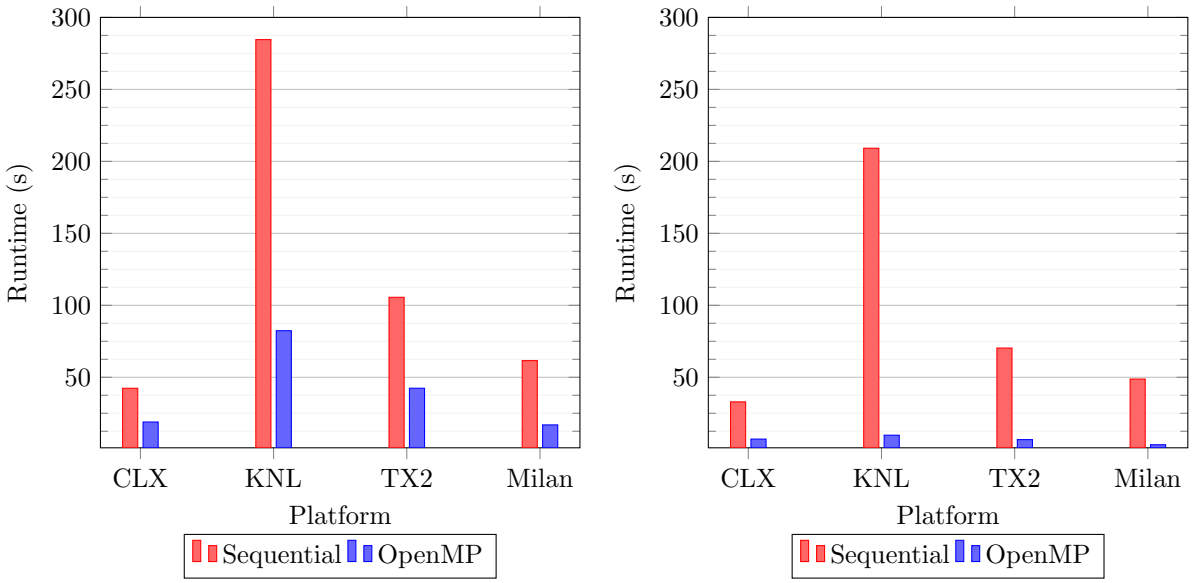
Currently the mini-application is developed in C++ and provides OpenMP directives for parallel execution. Alongside this, an OPS-like Domain Specific Language is being developed, named OP-PIC, that will allow the application to execute across a range of platforms from a single source. The development of this DSL and the results achieved are documented in Report 067270-TN-02.

The base application implements the electrostatic PIC method (i.e. it assumes that $\frac{\partial \vec{B}}{\partial t} = 0$). The mini-application is run on a test system, consisting of Deuterium ion flow through a pipe. The pipe is 4 mm in length with a 1 mm radius, and is divided up into 9337 elements with an average edge length of ∼0.2 mm. The plasma is fixed at $2 \times 10^8 K$ and the ions are injected with an input velocity of $1 \times 10^8 m/s$.

5.9.1 Performance

Both the Sequential and OpenMP variants are limited to CPU architectures, while the OPS-PIC variant can target CPU platforms through OpenMP, and NVIDIA GPU platforms through CUDA. In this report, we only provide data for the Sequential and OpenMP variants, executed on the Isambard system.

Results have been collected from Intel's Cascade Lake and KNL architectures, AMD's Rome architecture and Cavium's ThunderX2 platform. In all cases, the input is the "coarse" mesh, with 7511 elements, and ∼15,600 particles injected each time step.



(a) Total Runtime

(b) Move Particles Runtime

Figure 20: Runtime performance of Mini-FEM-PIC for the Coarse problem size on four of Isambard's platforms

Figure 20 shows the runtime on four of Isambard's CPU-based systems. The performance of our mini-application is dominated by the MoveParticles routine, and so Figure 20(b) shows the isolated runtime for this function. In each case, we plot only the fastest runtime, regardless of the number of parallel processes assigned (though in most cases this was achieved with either half or a full node – likely maximising memory bandwidth per core). OpenMP reduces the runtime on all four platforms by at least 2×, and in the case of the KNL and Milan platforms by almost 4×.

6 Analysis of Approaches

There are currently a large number of projects focused on preparing scientific applications for the complexities of Exascale. With many of the largest Supercomputers edging towards heterogeneity and hierarchical parallelism, many of these efforts are in ensuring that applications are performant *and portable* between different architectures. Section 3 outlines a wide number of options available for developing performance portable applications, and each approach comes with various advantages and disadvantages.

To date, only a small number of these approaches have seen widespread adoption, including OpenMP, Kokkos, and RAJA [19,20,43,44]. Because of the availability of mini applications that use these programming models, the majority of our evaluation has been based on these approaches. We have also conducted some preliminary work in assessing DPC++/SYCL, since adoption of this programming model is growing (owing to the backing of Intel).

6.1 Pragma-based Approaches

The two pragma-based approaches of OpenMP and OpenACC are perhaps the easiest to implement into an existing application and require only minimal code changes. Our evaluation shows that both programming models are typically performant on CPUs and GPUs, respectively, but potentially lack portability. In the case of OpenACC, which is specifically targeted at accelerator devices, this is expected; for OpenMP, it is perhaps more surprising.

The best data we have for this comes from the miniFE application, where we have runtime data for an OpenMP 3.0-compliant implementation and an OpenMP 4.5-compliant implementation. Figure 8 shows that for the CPU-only platforms, OpenMP 3.0 is competitive with (or is) the best performing miniFE variant, but does not run at all on the GPU platforms. While on some platforms, performance is lost when compared to MPI, it is a much simpler approach to parallelisation.

Figure 9 shows a cascade plot for all miniFE variants, showing that OpenMP offers good portability across the CPU platforms but no portability to accelerator devices, while The OpenMP 4.5 variant is portable to all architectures (except the Intel GPU currently). However, the performance on GPUs is significantly lacking that of native approaches, such as CUDA. Recent studies have suggested that different parallelisation strategies may be required for high performance between different platforms, and therefore it is possible that multiple implementations would need to be maintained. This can certainly be achieved within a single code base, using the preprocessor to select the correct code path, but essentially means maintaining multiple versions of each kernel.

Another useful example of the portability of OpenMP can be seen in the TeaLeaf data taken from Deakin et al. [19]. In Figure 5 OpenMP is typically shown to be performance portable, however these figures come from a C-based variant of the TeaLeaf application, in which multiple compute kernels are provided

targeting different versions of the OpenMP specification, different hardware and even different compilers[26]. This is another illustration that if we were to maintain multiple kernel implementations, we may be able to achieve good performance with a mixture of OpenMP 3.0 and 4.5 directives (though whether this approach is "portable" is questionable).

6.2 Programming Model Approaches

The next approach we have explored in this report, is the use of alternative programming models that are targeted at parallel architectures. The template libraries Kokkos and RAJA are most mature of these approaches. Both are being developed as part of the Exascale Computing Project within the US Department of Energy, at Sandia National Laboratories and Lawrence Livermore National Laboratory, respectively. They are each capable of targeting the range of hardware that is going to be present in the Aurora, Frontier and El Capitan systems, through a combination of OpenMP, CUDA, HIP and DPC++. Our initial results (and many other studies [19, 20, 43, 44]) have shown that both are typically able to deliver good and portable performance from a single source code base.

The results in Figure 6 shows this for TeaLeaf, with both Kokkos and RAJA typically being able to achieve good application efficiency over all platforms, with the exception of using multiple GPUs (which has not yet been implemented in TeaLeaf).

For the high-order FEM Laghos application, Figure 10 shows that RAJA is the only portable programming model available and is shown to be competitive with (or is) the fastest performing variant on each platform. It should be noted that Laghos is an exceptional case in our evaluation set, since portability is implemented in the HYPRE and MFEM libraries, rather than the core Laghos code itself.

For the PIC codes in our evaluation set, Kokkos is the only performance portable programming model that has been extensively used. The best source for comparison is therefore the VPIC code, where there is a vectorised CPU-only variant for comparison. The vectorisation in VPIC is largely hand-coded, with multiple versions of each kernel available for selection at compile time (depending on vector-size and vector instruction availability). Figure 15 demonstrates that while the optimal implementation on each of the CPU-based platforms is the hand-vectorised variant, the Kokkos version is competitive with the unvectorised implementation; better compiler autovectorisation may help close this performance gap in the future[27]. Importantly, the Kokkos variant can be executed across GPUs, where much of the available performance is likely to lie in post-Exascale systems.

While Kokkos and RAJA have both shown promise as approaches to performance portable application development, each also carry a small element of risk. For each API there is potentially a single point of failure – the API may be changed at short notice; support for the API or development of the library may be withdrawn at any time; and hardware backends may never be developed. Nonetheless, a high level of

[26] See: `https://github.com/UoB-HPC/TeaLeaf/tree/master/2d/c_kernels`

[27] Indeed, a similar issue was seen during the development of EMPIRE-PIC, where the compiler is not able to fully vectorise some segments of Kokkos code, despite no apparent dependencies [35]. The new SIMD feature in Kokkos should reduce this performance gap significantly [41]

support is likely to be maintained while the APIs form the backbone of many of the Department of Energy's most important post-Exascale HPC applications. There are also ongoing efforts to include parts of the API in the C++ standard[28].

In contrast to Kokkos and RAJA, the SYCL programming model is an open standard maintained by the Khronos Group. Interest in SYCL is growing rapidly, driven in part by Intel's decision to adopt the programming model for their Exascale systems, and in particular their Xe HPC accelerators (in the form of Data Parallel C++).

Due to the relative immaturity of SYCL/DPC++, there are not many NEPTUNE-relevant mini-applications available for evaluation; our evaluation has so far been limited to a simple heat diffusion code and a code conversion of the OpenMP 4.5 miniFE implementation. Figure 3 shows that for a simple code implemented in SYCL, excellent performance portability can be achieved. For a more complex case such as miniFE (see Figure 9), the performance portability of SYCL is similar to the performance portability of OpenMP 4.5. We expect that newer SYCL compilers and a more targetted optimisation effort will yield better performance; revisiting these studies periodically is central to our ongoing work.

Besides our own evaluation, there has been a number of recent efforts to explore the portability of SYCL and the maturity of SYCL compilers that offer some useful insights. Reguly et al. evaluate SYCL performance through the unstructured mesh CFD solver, MG-CFD [45]. Figure 21 shows their SYCL runtimes compared to the best observed performance on each platform; note, the Cascade Lake and Xe LP results were compiled using Intel's OneAPI compiler, all other SYCL targets were built using hipSYCL.

Similar to our own evaluation, they observe that SYCL is typically not competitive, but is able to target each architecture from a single code base. In the case of the ARM Graviton2 platform, the SYCL build is considerably worse due to the infancy of the ARM target in hipSYCL. For the two Intel platforms, the OneAPI compiler is slightly more competitive; for the Iris Xe LP (low-power) target, its runtime is competitive with a single socket Cascade Lake. On the GPU platforms, SYCL is still considerable slower than native CUDA builds, but has the advantage of being portable to the AMD and Intel GPUs.

The study by Lin et al. provides more data on the maturity of SYCL implementations by evaluating the same small set of applications periodically against the hipSYCL, Intel DPC++ and ComputeCpp compilers [46]. Their evaluations are based on three applications: BabelStream, a port of the STREAM memory benchmark for parallel programming frameworks; BUDE (Bristol University Docking Engine), a molecular dynamics application; and CloverLeaf, a 2D structured grid application. They evaluate each application on a Xeon Cascade Lake, an AMD EPYC Rome, an NVIDIA V100 and an Intel HD P630 GPU. Although their study is primarily tracking absolute performance changes with compiler version, rather than comparing to the "best case", they do also provide a brief comparison for each application.

For BabelStream, DPC++ and ComputeCpp closely match the OpenCL performance; this is not surprising since both of these compilers target the OpenCL runtime. Conversely, hipSYCL is competitive with OpenMP and Kokkos on the Cascade Lake, but is the worst performing on the Rome platform.

[28] e.g. mdspan, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0009r10.html`

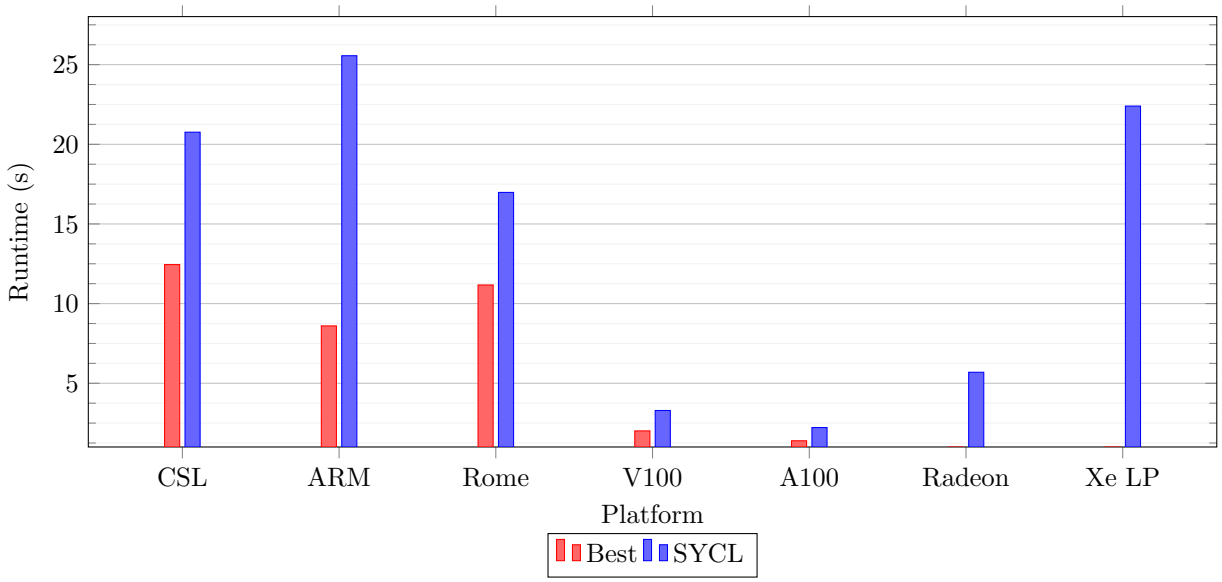


Figure 21: MG-CFD runtime data from Reguly et al. [45]

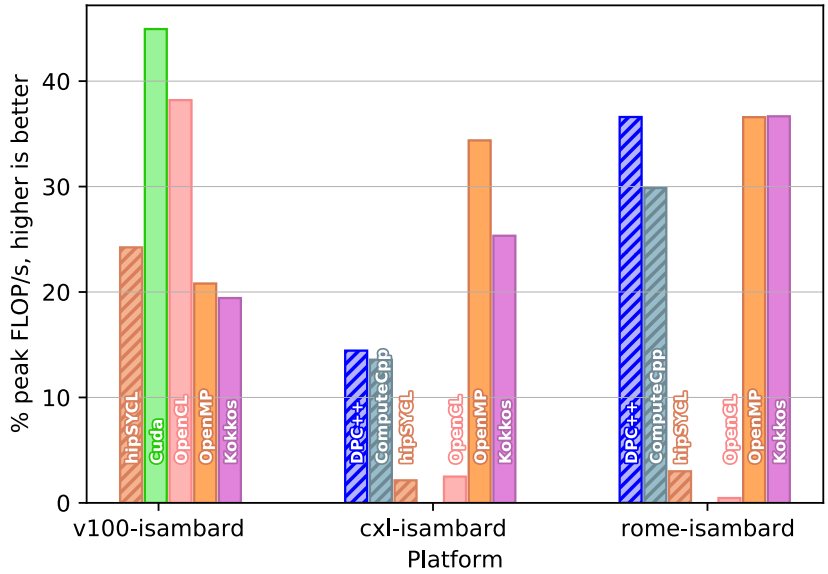


Figure 22: CloverLeaf: SYCL vs. alternative frameworks from Lin et al. [46]

For the two mini-applications the results are more varied; in some cases there are large differences between the compilers (see Figure 22). In this study, only hipSYCL was able to target the NVIDIA GPU, due to compatibility with NVIDIAs outdated OpenCL runtime. Nonetheless, the hipSYCL performance is not competitive with any of the alternatives. On the CPU platforms, hipSYCL often achieves the lowest performance of the three SYCL compilers, and DPC++ tends to outperform ComputeCpp slightly.

It is important to note that these results are based on compilers that are currently undergoing significant

engineering efforts. It is therefore likely that many of the performance gaps that currently exist will reduce in time.

6.3 High-level DSL Approaches

Many of the approaches discussed above could be considered low-level DSLs, and these approaches have formed the majority of our analysis in this project. However, we also have a small dataset for the OPS DSL, which subsequently acts as a code-generator for these lower-level DSLs/programming models.

OPS is an approach specifically targeted at structured mesh applications, and has been used to parallelise TeaLeaf to good effect. The previously seen TeaLeaf data in Figure 6 demonstrates that OPS is approximately equal with Kokkos and RAJA in terms of its performance portability. However, the process of porting an application to OPS is arguable more complex, and therefore may effect programmer *productivity*[29].

There are a number of other high-level DSLs that we have not explored in this project, but may form part of our future analyses. In particular, the Unified Form Language (UFL) that is used by both Firedrake and FEniCS is already being used in some of the NEPTUNE work packages. UFL is a DSL, embedded in Python, that allows scientists to express their equations in PDE form. The Firedrake/FEniCS packages handle the discretisation of these equations, and use PyOP2 to generate portable executable code. Although we have not explored these high-level DSLs in this project, we have analysed many of the programming models that PyOP2 can target.

6.4 Summary

It is likely that in NEPTUNE, multiple DSLs may be present, with high-level DSLs allowing scientists to express equations, and low-level DSLs and programming models targeting different parallel architectures. This project has mainly focused on the latter, since these are likely to be performance-critical.

In this project we have evaluated multiple approaches to developing performance portable software, ranging from pragma-based code annotations, through to purpose-built domain specific languages.

In our analysis we have found that pragma-based approaches like OpenMP and OpenACC are able to achieve high performance on a variety of platforms, but that OpenMP is typically not portable to GPU accelerators, and OpenACC is not portable to CPU host platforms. Although the OpenMP 4.5 standard allows for offloaded computation, achieving high performance across both CPUs and GPUs often requires different design decisions to be made. However, it is likely that performance of OpenMP 4.5-compliant codes will improve as compiler support develops.

Of the performance portable programming models explored, Kokkos and RAJA are perhaps the most mature currently, with both offering good portability for a small performance decrease. Furthermore, the APIs are

[29] See: `https://op-dsl.github.io/docs/OPS/tutorial.pdf`

relatively simple, primarily being a drop-in replacement for loop structures, meaning that the effort to port applications to these programming models is not great.

Currently, the SYCL programming model suffers many of the same issues as OpenMP 4.5. Again, it is likely that as compiler support improves, the performance penalty will lessen. Furthermore, the open-standard nature of SYCL means that it potentially carries slightly less risk than the DoE-supported Kokkos and RAJA programming models – though it should be noted that Kokkos can code-generate to SYCL/DPC++ in order to target Intel Xe GPUs.

Our evaluation of purpose-built DSLs has been limited to OPS, evaluated through the TeaLeaf application. Although it is able to offer good performance portability, it is limited in the computational methods it can be applied to, i.e., multi-block structured mesh algorithms.

7 Key Findings and Recommendations

This project has evaluated a number of approaches to performance portability, many of which have shown promise as possible approaches for NEPTUNE. The direction of HPC is clearly moving towards heterogeneity, but its not clear which software development methodology will win out.

The development of a *new* simulation code for project NEPTUNE presents an almost unique opportunity to design and build a code with Exascale execution as a primary concern.

Because of the wealth of choice in approaches to performance portability, and the required longevity of the NEPTUNE code, it is prudent to consider all available options prior to, and during, development. With this in mind, we make the following recommendations for the initial development of NEPTUNE. As the hardware and software landscape continues to evolve over the next decade, it is anticipated that this document will likewise need to evolve, and that these recommendations will tighten as appropriate.

1. Develop in C++

1.1. Focus Core Development on Modern, Standard C++

In order to enable the most opportunity for performance portable design and optimisation of NEPTUNE, our first recommendation is that the core of NEPTUNE is initially written in standard modern C++, making full use of object orientation and template metaprogramming.

At the present time, the choice of C++ carries a number of advantages over Fortran (the mainstay of scientific computing).

- Object orientation is at the core of the C++ language, encouraging encapsulation, sensible design and code reuse[30];
- Templating and template metaprogramming can enable some advanced compile-time optimisations, or compile-time code generation (thus improving code reuse);
- New features in the C++ standard are typically implemented in modern C++ compilers (e.g. Clang) much faster than equivalent Fortran compilers (e.g. Flang);
- A large number of modern mathematical and scientific libraries are written in C/C++ and provide native APIs. Although it may be possible to interface with some of these libraries with Fortran, this may come with a loss of functionality.

[30] Although Fortran introduced object orientation in the 2003 standard, it lacks many of the advanced features present in C++ [47].

In addition to the benefits of the C++ language, there are other reasons to pursue a C++ code that relate specifically to producing a performance portable application. The vast majority of new libraries, programming models and portability layers are developed with C/C++ as their first target language; this means that an application developed in C++ is more likely to be able to make use of these libraries and programming models.

A number of these libraries rely specifically on C++ features, such as template metaprogramming, meaning that C++ is not only the first target, but also the *only* target language (e.g. RAJA, Kokkos). Another example of this is in Intel's OneAPI, where although many of the libraries are language agnostic (e.g. Math Kernel Library, Data Analytics Library), the central programming language, Data Parallel C++ (DPC++), is an extension of the C++ language.

1.2. Use Open Standards and Beware of Vendor Lock-in

Alongside the recommendation to pursue ISO C++, we recommend that open standards are used where possible (followed by open source solutions). Additionally, caution is required when adopting vendor specific abstractions unless wider support is forthcoming (as is the case with Intel's DPC++).

There are a number of approaches that are open standards and should remain portable across a wide range of platforms, such as MPI, OpenMP, OpenACC and SYCL. In some cases, the support for these open standards is very good (e.g. OpenMP), and some where support significantly lags the standard (e.g. OpenACC). However, pursuing these approaches offers the best chance for NEPTUNE to remain performance portable in the future.

Alongside these programming models, there are a number of proprietary approaches that target specific hardware, such as CUDA and HIP/ROCm. These are likely to yield greater performance gains on their target platforms but are not portable approaches. One possible safeguard against this, is to use an open source middleware such as Kokkos or RAJA, which can generate native CUDA or HIP/ROCm code at compile-time.

A vendor-specific approach such as Intel's OneAPI may also strike a balance between portability and performance. Many of the libraries in OneAPI are implementations of standard libraries such as BLAS, and the programming model is heavily based on the SYCL open standard.

Typically, open standards may be less agile for targeting the latest hardware and hardware features, but proprietary approaches are likely to restrict the choice of future hardware.

2. Separation of Concerns

2.1. Select a Good High-Level Abstraction

It is possible that multiple DSLs will be employed within the NEPTUNE code, and that these DSLs will exist at different levels of abstraction. Selecting a good high-level abstraction will be vital to the success of NEPTUNE.

Domain Specific Languages exist at multiple levels of abstraction. Many programming models, such as Kokkos, RAJA and SYCL, could be considered low-level DSLs. They provide functionality targeted at exploiting the parallel hardware resources that are available on a system.

Above these low-level DSLs are programming models that are targeted at particular algorithmic domains. The OPS and OP2 libraries are two such examples that provide abstractions for representing computation over structured and unstructured meshes, respectively. The intermediate compiler can exploit the structure of the problem space to perform a number of code optimisations to improve performance.

At the highest level are languages such as UFL and BOUT++, that allow scientists to write partial differential equations (PDEs) directly into the code. At compile-time, these expressions are used to generate code in lower-level DSLs such as PyOP2 and RAJA, for execution on a parallel system.

Typically, the more abstract a DSL is the greater the space for synthesis [48]. However, adding new features to, or escaping from, a high-level DSL may be problematic. For this reason, it is important that a good high level abstraction is chosen (or developed) that allows scientists to accurately represent their science, without being overly restrictive, and that where possible, it is extensible to new operators and features, allowing scientists to step outside of the DSL without sacrificing performance.

2.2. Abstract Data Storage

Performant data structures can be very architecture dependent. Especially as we move towards heterogeneous platforms, every effort should be made to abstract data storage, such that transformations can be made that are transparent to the underlying algorithms.

Exploiting full performance on modern architectures is heavily reliant on how efficiently data is moved between main memory and the various layers of cache. For memory-bound applications, the data structures that are used to store scientific data can significantly affect performance, and the best data structure for one platform may not be the best for another.

For this reason, the NEPTUNE design should abstract data storage away from algorithms as much as possible, such that it does not harm performance. This, coupled with the use of appropriate data libraries, will ensure that data structures can be changed, without requiring significant re-engineering of key computational kernels. It will also enable compile-time transformations based on execution target.

2.3. Prototype, prototype, prototype

A well modularised design should enable key computational kernels to be extracted for prototyping. Before applying particular programming models to the NEPTUNE code, prototyping will allow rapid evaluation of emerging approaches on kernels that are performance critical.

Following programmes such as the Exascale Computing Project (ECP) and the wider adoption of approaches such as SYCL, there are currently a wealth of approaches to developing performance portable software that are in active development. Because of this, it is not entirely clear which approaches will win out.

Therefore to protect against this, it is prudent to develop NEPTUNE alongside a programme of prototyping key kernels. A well encapsulated, modular design should allow isolated kernels to be evaluated throughout development.

This will be aided by an inherent similarity in many programming models aimed at performance portability, where parallelism is largely exposed at the loop-level. As it becomes clearer which programming models are likely to be most appropriate for NEPTUNE, code changes can be implemented incrementally. In some cases, where a high-level DSL has been employed, changes in code generators will automate much of the required effort.

3. Don't Reinvent the Wheel

Code reuse should be at the heart of NEPTUNE, and this extends to the use of external libraries. There are a number of libraries that implement functionality commonly found in scientific simulation software, and NEPTUNE should make full use of these libraries where possible. Vendor-optimised versions of these libraries often exist, providing performance improvements for free.

The work in this project has primarily focused on the programming model in use for parallelisation at a node-level, given the assumption that it is highly likely that MPI will be the defacto standard for inter-node communication (the so called MPI+X model). Besides the use of the existing MPI standard, it is likely that there are a number of other libraries that can provide functionality for NEPTUNE *for free*, and it is important that these are used wherever possible.

Much of computation in NEPTUNE is likely to be in solving complex linear systems, and for that there are number of industry-standard libraries (such as LAPACK and BLAS) that are highly optimised. Where possible, these libraries should be used to provide functionality, since this reduces the technical burden and means that we can take advantage of vendor-led optimisations for free. Beside the algorithmic optimisations in these libraries, the vendor-produced implementations are often architecturally optimised.

Besides the availability of vendor-optimised libraries, the choice of some libraries may naturally encourage the adoption of particular parallel programming models. For example, Intel's OneAPI Math Kernel Library

(MKL) would motivate the use of DPC++/SYCL; the Trilinos library would perhaps motivate the use of Kokkos; the HYPRE and MFEM libraries would lend themselves to RAJA.

But, its important that the available libraries are explored by domain specialists to ensure any library chosen fits its purpose without being overly restrictive.

7.1 Future Work

The key findings and recommendations in this report are the result of an extensive study into parallel programming models and mini-applications relevant to fusion modelling. Both of these fields are constantly evolving, and so it is necessary that the content and recommendations of this report also evolve. To this end there are a number studies in progress that will enhance this report.

Specifically, we aim to:

1. Add new applications to the evaluation set (e.g. HipBone, SheathPIC, etc).
2. Evaluate our newly developed EM-PIC mini-application.
3. Enhance our evaluation of SYCL-based codes.
4. Evaluate the similarity of mini-applications to host codes such as Bout++.

References

[1] Steven A. Wright, Christopher Ridgers, Gihan Mudalige, Zaman Lantra, Josh Williams, Andrew Sunderland, Sue Thorne, and Wayne Arter. Developing Performance Portable Simulations for the Design of a Nuclear Fusion Reactor. *Computer Physics Communications*, (Under Review) 2023.

[2] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, January 2019.

[3] Thiruvengadam Vijayaraghavan, Yasuko Eckert, Gabriel H. Loh, Michael J. Schulte, Mike Ignatowski, Bradford M. Beckmann, William C. Brantley, Joseph L. Greathouse, Wei Huang, Arun Karunanithi, Onur Kayiran, Mitesh Meswani, Indrani Paul, Matthew Poremba, Steven Raasch, Steven K. Reinhardt, Greg Sadowski, and Vilas Sridharan. Design and analysis of an apu for exascale computing. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 85–96, 2017.

[4] Jack Dongarra, Steven Gottlieb, and William T. C. Kramer. Race to exascale. *Computing in Science and Engg.*, 21(1):4–5, January 2019.

[5] István Z. Reguly and Gihan R. Mudalige. Productivity, performance, and portability for computational fluid dynamics applications. *Computers & Fluids*, 199:104425, 2020.

[6] Jaswinder Pal Singh and John L Hennessy. An empirical investigation of the effectiveness and limitations of automatic parallelization. *Shared memory multiprocessing*, pages 203–207, 1992.

[7] S.J. Pennycook, J.D. Sewall, and V.W. Lee. Implications of a metric for performance portability. *Future Generation Computer Systems*, 92:947–958, 2019.

[8] Jason Sewall, S. John Pennycook, Douglas Jacobsen, Tom Deakin, and Simon McIntosh-Smith. Interpreting and visualizing performance portability metrics. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 14–24, 2020.

[9] Alan B. Williams. Cuda/GPU version of miniFE mini-application. 2 2012.

[10] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[11] UK Mini-App Consortium. Uk-mac. `http://uk-mac.github.io` (accessed April 20, 2021), 2021.

[12] David H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, Horst D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[13] Exascale Computing Project. ECP Proxy Applications. `https://proxyapps.exascaleproject.org/` (accessed April 20, 2021), 2021.

[14] Guido Juckeland, William Brantley, Sunita Chandrasekaran, Barbara Chapman, Shuai Che, Mathew Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-Mei W. Hwu, Huian Li, Matthias S. Müller, Wolfgang E. Nagel, Maxim Perminov, Pavel Shelepugin, Kevin Skadron, John Stratton, Alexey Titov, Ke Wang, Matthijs van Waveren, Brian Whitney, Sandra Wienke, Rengan Xu, and Kalyan Kumaran. SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance. In Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond, editors, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, pages 46–67. Springer International Publishing, 2015.

[15] Benjamin Daniel Dudson, Peter Alec Hill, David Dickinson, Joseph Parker, Adam Dempsey, Andrew Allen, Arka Bokshi, Brendan Shanahan, Brett Friedman, Chenhao Ma, David Schwörer, Dmitry Meyerson, Eric Grinaker, George Breyiannia, Hasan Muhammed, Haruki Seto, Hong Zhang, Ilon Joseph, Jarrod Leddy, Jed Brown, Jens Madsen, John Omotani, Joshua Sauppe, Kevin Savage, Licheng Wang, Luke Easy, Marta Estarellas, Matt Thomas, Maxim Umansky, Michael Løiten, Minwoo Kim, M Leconte, Nicholas Walkden, Olivier Izacard, Pengwei Xi, Peter Naylor, Fabio Riva, Sanat Tiwari, Sean Farley, Simon Myers, Tianyang Xia, Tongnyeol Rhee, Xiang Liu, Xueqiao Xu, and Zhanhui Wang. BOUT++, 10 2020.

[16] B D Dudson, M V Umansky, X Q Xu, P B Snyder, and H R Wilson. BOUT++: A framework for parallel plasma fluid simulations. *Computer Physics Communications*, 180:1467–1480, 2009.

[17] C.D. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. De Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot, B. Jordi, H. Xu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R.M. Kirby, and S.J. Sherwin. Nektar++: An open-source spectral/hp element framework. *Computer Physics Communications*, 192:205–219, 2015.

[18] Jan Eichstädt. Implementation of High-performance GPU Kernels in Nektar++, 2020.

[19] Tom Deakin, Simon McIntosh-Smith, James Price, Andrei Poenaru, Patrick Atkinson, Codrin Popa, and Justin Salmon. Performance portability across diverse computer architectures. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 1–13, 2019.

[20] R. O. Kirk, G. R. Mudalige, I. Z. Reguly, S. A. Wright, M. J. Martineau, and S. A. Jarvis. Achieving Performance Portability for a Heat Conduction Solver Mini-Application on Modern Multi-core Systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 834–841, Sep. 2017.

[21] Matthew Martineau, Simon McIntosh-Smith, and Wayne Gaudin. Assessing the performance portability of modern parallel programming models using tealeaf. *Concurrency and Computation: Practice and Experience*, 29(15):e4117, 2017.

[22] Simon McIntosh-Smith, Matthew Martineau, Tom Deakin, Grzegorz Pawelczak, Wayne Gaudin, Paul Garrett, Wei Liu, Richard Smedley-Stevenson, and David Beckingsale. TeaLeaf: A Mini-Application to Enable Design-Space Explorations for Iterative Sparse Linear Solvers. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 842–849, 2017.

[23] Richard Frederick Barrett, Li Tang, and Sharon X. Hu. Performance and Energy Implications for Heterogeneous Computing Systems: A MiniFE Case Study. 12 2014.

[24] Meng Wu, Can Yang, Taoran Xiang, and Daning Cheng. The research and optimization of parallel finite element algorithm based on minife. *CoRR*, abs/1505.08023, 2015.

[25] David F. Richards, Yuri Alexeev, Xavier Andrade, Ramesh Balakrishnan, Hal Finkel, Graham Fletcher, Cameron Ibrahim, Wei Jiang, Christoph Junghans, Jeremy Logan, Amanda Lund, Danylo Lykov, Robert Pavel, Vinay Ramakrishnaiah, et al. FY20 Proxy App Suite Release. Technical Report LLNL-TR-815174, Exascale Computing Project, September 2020.

[26] J. C. Camier. Laghos summary for CTS2 benchmark. Technical Report LLNL-TR-770220, Lawrence Livermore National Laboratory, March 2019.

[27] Robert Anderson, Julian Andrej, Andrew Barker, Jamie Bramwell, Jean-Sylvain Camier, Jakub Cerveny, Veselin Dobrev, Yohann Dudouit, Aaron Fisher, Tzanio Kolev, Will Pazner, Mark Stowell, Vladimir Tomov, Ido Akkerman, Johann Dahm, David Medina, and Stefano Zampini. Mfem: A modular finite element methods library. *Computers & Mathematics with Applications*, 81:42–74, 2021. Development and Application of Open-source Software for Problems with Numerical PDEs.

[28] V. Grandgirard, J. Abiteboul, J. Bigot, T. Cartier-Michaud, N. Crouseilles, G. Dif-Pradalier, Ch. Ehrlacher, D. Esteve, X. Garbet, Ph. Ghendrih, G. Latu, M. Mehrenberger, C. Norscini, Ch. Passeron, F. Rozar, Y. Sarazin, E. Sonnendrücker, A. Strugarek, and D. Zarzoso. A 5d gyrokinetic full-f global semi-lagrangian code for flux-driven ion turbulence simulations. *Computer Physics Communications*, 207:35–68, 2016.

[29] David S Medina, Amik St-Cyr, and T. Warburton. OCCA: A unified approach to multi-threading languages, 2014.

[30] T D Arber, K Bennett, C S Brady, A Lawrence-Douglas, M G Ramsay, N J Sircombe, P Gillies, R G Evans, H Schmitz, A R Bell, and C P Ridgers. Contemporary particle-in-cell approach to laser-plasma modelling. *Plasma Physics and Controlled Fusion*, 57(11):113001, sep 2015.

[31] Michael Bareford. minEPOCH3D Performance and Load Balancing on Cray XC30. Technical Report eCSE03-1, Edinburgh Parallel Computer Centre, 2016.

[32] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson. 0.374 Pflop/s Trillion-Particle Kinetic Modeling of Laser Plasma Interaction on Roadrunner. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08. IEEE Press, 2008.

[33] Robert Bird, Nigel Tan, Scott V Luedtke, Stephen Harrell, Michela Taufer, and Brian Albright. VPIC 2.0: Next Generation Particle-in-Cell Simulations. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2021.

[34] Matthew T. Bettencourt and Sidney Shields. EMPIRE Sandia's Next Generation Plasma Tool. Technical Report SAND2019-3233PE, Sandia National Laboratories, March 2019.

[35] Matthew T. Bettencourt, Dominic A. S. Brown, Keith L. Cartwright, Eric C. Cyr, Christian A. Glusa, Paul T. Lin, Stan G. Moore, Duncan A. O. McGregor, Roger P. Pawlowski, Edward G. Phillips, Nathan V. Roberts, Steven A. Wright, Satheesh Maheswaran, John P. Jones, and Stephen A. Jarvis. EMPIRE-PIC: A Performance Portable Unstructured Particle-in-Cell Code. *Communications in Computational Physics*, x(x):1–37, March 2021.

[36] Dominic A.S. Brown, Matthew T. Bettencourt, Steven A. Wright, Satheesh Maheswaran, John P. Jones, and Stephen A. Jarvis. Higher-order particle representation for particle-in-cell simulations. *Journal of Computational Physics*, 435:110255, 2021.

[37] Jeanine Cook, Omar Aaziz, Si Chen, William Godoy, Amy Powell, Gregory Watson, Courtenay Vaughan, and Avani Wildani. Quantitative performance assessment of proxy apps and parents (report for ecp proxy app project milestone adcd-504-28). 4 2022.

[38] Yuuichi Asahi, Guillaume Latu, Julien Bigot, and Virginie Grandgirard. Optimization strategy for a performance portable vlasov code. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 79–91, 2021.

[39] Nicolas Crouseilles, Guillaume Latu, and Eric Sonnendrücker. A parallel Vlasov solver based on local cubic spline interpolation on patches. *Journal of Computational Physics*, 228(5):1429–1446, 2009.

[40] Stuart Slattery, Samuel Temple Reeve, Christoph Junghans, Damien Lebrun-Grandié, Robert Bird, Guangye Chen, Shane Fogerty, Yuxing Qiu, Stephan Schulz, Aaron Scheinberg, Austin Isner, Kwitae Chong, Stan Moore, Timothy Germann, James Belak, and Susan Mniszewski. Cabana: A Performance Portable Library for Particle-Based Simulations. *Journal of Open Source Software*, 7(72):4115, 2022.

[41] Nigel Phillip Tan, Scott Vernon Luedtke, Robert Bird, Stephen Lien Harrell, Michela Taufer, and Brian James Albright. The Performance-Portability Trade-Off Challenge in Next Generation Particle-In-Cell Simulations. 6 2022.

[42] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J. Ligocki, Matthew J. Cordery, Nicholas J. Wright, Mary W. Hall, and Leonid Oliker. Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis. In Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond, editors, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, pages 129–148. Springer International Publishing, 2015.

[43] Simon McIntosh-Smith. Performance Portability Across Diverse Computer Architectures. In *P3MA: 4th International Workshop on Performance Portable Programming models for Manycore or Accelerators*, 2019.

[44] T. R. Law, R. Kevis, S. Powell, J. Dickson, S. Maheswaran, J. A. Herdman, and S. A. Jarvis. Performance portability of an unstructured hydrodynamics mini-application. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 0–12, Nov 2018.

[45] I. Z. Reguly, A. M. B. Owenson, A. Powell, S. A. Jarvis, and G. R. Mudalige. Under the Hood of SYCL – An Initial Performance Analysis With an Unstructured-mesh CFD Application. In Bradford L. Chamberlain, Ana-Lucia Varbanescu, Hatem Ltaief, and Piotr Luszczek, editors, *Proceedings of the International Supercomputing Conference (ISC 2021)*, pages 391–410. Springer International Publishing, June 2021.

[46] Wei-Chen Lin, Tom Deakin, and Simon McIntosh-Smith. On measuring the maturity of sycl implementations by tracking historical performance improvements. In *International Workshop on OpenCL*, IWOCL'21, New York, NY, USA, 2021. Association for Computing Machinery.

[47] John R. Cary, Svetlana G. Shasharina, Julian C. Cummings, John V.W. Reynders, and Paul J. Hinker. Comparison of C++ and Fortran 90 for object-oriented scientific programming. *Computer Physics Communications*, 105(1):20–36, 1997.

[48] Paul Kelly. Synthesis versus Analysis: What Do We Actually Gain from Domain-Specificity? Invited talk at The 28th International Workshop on Languages and Compilers for Parallel Computing, Available: `https://www.csc2.ncsu.edu/workshops/lcpc2015/slide/2015-09-LCPC-Keynote-PaulKelly-V03-ForDistribution.pdf`, 2015.

A Code Examples

A.1 OpenMP

Figure 23 shows a simple vector addition, where the loop iterations are distributed across OpenMP threads. The number of threads used is typically specified with the environmental variable `OMP_NUM_THREADS`, but usually will default to the number of cores available if unset. Finer control over the parallelism can be achieved with more complex annotations, such as `schedule` and `collapse`.

```
#pragma omp parallel for
for (int i = 0; i < 100; i++) {
   c[i] = a[i] + b[i];
}
```

Figure 23: OpenMP code listing

A.2 OpenMP Target Directives

An example of the same vector addition seen previously is provided in Figure 24 with `target` directives. In addition to specifying the parallel region, data mapping information is also required, indicating which data should be moved to and from an accelerator device.

```
#pragma omp target map (to:a[:size]) map (to:b[:size]) map (tofrom:c[:size])
#pragma omp teams distribute parallel for default(none)
for (int i = 0; i < 100; i++) {
   c[i] = a[i] + b[i];
}
```

Figure 24: OpenMP 4.5 using target directives

A.3 SYCL and DPC++

Figure 25 provides an equivalent vector-add written in SYCL. Similar to OpenMP with offload, data movement is expressed explicitly in the language; in the case of SYCL this is through device buffers with access specifiers.

```
sycl::queue myqueue;
std::vector h_a(100), h_b(100), h_c(100);
sycl::buffer d_a(h_a), d_b(h_b), d_c(h_c);

auto ev = myqueue.submit([&](handler &h){
   auto a = d_a.get_access<access::read>();
   auto b = d_b.get_access<access::read>();
   auto c = d_c.get_access<access::write>();
   h.parallel_for(count, kernel_functor([=](id<> item) {
      int i = item.get_global(0);
      c[i] = a[i] + b[i];
   }));
});
```

Figure 25: SYCL

A.4 Kokkos

Figure 26 outlines a vector add using Kokkos's `parallel_for` function.

```
Kokkos::parallel_for(100, KOKKOS_LAMBDA (const int& i) {
   c[i] = a[i] + b[i];
});
```

Figure 26: Kokkos

Kokkos also provides fully managed multi-dimensional arrays through its View class. Figure 27 provides a simple example of a two dimensional array in Kokkos. Because Kokkos Views are fully managed, they are allocated and reference counted, additional arguments can be provided to specify the memory space in which they are allocated, and whether to use column-major or row-major layout can be specified in code. This may allow some very simple performance optimisations to be made at a single point in an applications code.

```
const size_t num_rows = ...;
const size_t num_cols = ...;
Kokkos::View<int**> array ("some label", num_rows, num_cols);
array(0,0) = ...;
```

Figure 27: Use of `Kokkos::View` for multi-dimensional arrays

A.5 RAJA

A vector add can be implemented similarly in RAJA, as is provided in Figure 28.

```
RAJA::RangeSegment seg (0, 100);
RAJA::forall<loop_exec> (seg, [=] (int i) {
   c[i] = a[i] + b[i];
});
```

Figure 28: RAJA

Like Kokkos, RAJA provides a view class for handling multi-dimensional arrays. Figure 29 shows the use of the `RAJA::View` class on a simple two-dimensional array.

```
const int DIM = 2;
double *array = new double[num_rows * num_cols];
RAJA::View<double, RAJA::Layout<DIM> > array_view(array, num_rows, num_cols);
Aview(0,0) = ...;
...
free(array);
```

Figure 29: Use of `RAJA::View` for multi-dimensional arrays

A.6 Bout++

Bout++ provides two Domain Specific Languages (DSLs). The first is in how equations are encoded into the source, with C++ templates generating parallelised, performant code from these mathematical expressions.

For example the MHD equations (Eq. 2-5) can be expressed in C++ as in Figure 30.

$$\frac{\partial \rho}{\partial t} = -\mathbf{v} \cdot \nabla \rho - \rho \nabla \cdot \mathbf{v} \tag{2}$$

$$\frac{\partial p}{\partial t} = -\mathbf{v} \cdot \nabla p - \gamma p \nabla \cdot \mathbf{v} \tag{3}$$

$$\frac{\partial \mathbf{v}}{\partial t} = -\mathbf{v} \cdot \nabla \mathbf{v} + \frac{1}{\rho}(-\nabla p + (\nabla \times \mathbf{B}) \times \mathbf{B}) \tag{4}$$

$$\frac{\partial \mathbf{B}}{\partial t} = \nabla \times (\mathbf{v} \times \mathbf{B}) \tag{5}$$

```
ddt(rho) = -V_dot_Grad(v, rho) - rho*Div(v);
ddt(p)   = -V_dot_Grad(v, p) - g*p*Div(v);
ddt(v)   = -V_dot_Grad(v, v) + (cross(Curl(B),B) - Grad(p))/rho;
ddt(B)   = Curl(cross(v,B));
```

Figure 30: BOUT++ MHD equations implementation

A second eDSL is provided in Bout++ input files. Figure 31 shows part of an example input file.

```
[n]  # Density
height = 0.5
width = 0.05

blob1 = height * exp(-((x-0.35)/width)^2
                     - ((z/(2*pi) - 0.5)/width)^2)
blob2 = height * exp(-((x-0.15)/width)^2
                     - ((z/(2*pi) - 0.4)/width)^2)

function = 1 + blob1 + blob2
```

Figure 31: Part of a BOUT++ input file, specifying the density initial condition as a function of position in x and z.

A.7 UFL/Firedrake

Firedrake and FEniCS both use a common DSL, known as the Unified Form Language (UFL). Like Bout++, UFL allows scientists to express their equations in code, with the code generator providing the discretisation and parallelisation.

For example [31], the modified Helmholtz equation:

$$-\nabla^2 u + u = f \tag{6}$$

$$\nabla \cdot \hat{n} = 0 \qquad \text{on boundary } \Gamma \tag{7}$$

can be transformed into variational form by multiplying by a test function v and integrating over the domain Ω:

$$\int_\Omega \nabla u \cdot \nabla v + uv dx = \int v f dx + \underbrace{\int_\Gamma v \nabla u \cdot \hat{n} ds}_{\to 0 \text{ due to boundary condition}} \tag{8}$$

This can be implemented in UFL as in Figure 32.

[31] From ://www.firedrakeproject.org/demos/helmholtz.py.html

```
from firedrake import *
mesh = UnitSquareMesh(10, 10)   # Define the mesh
V = FunctionSpace(mesh, "CG", 1)  # Function space of the solution
u = TrialFunction(V)
v = TestFunction(V)
f = Function(V)  # Define a function and give it a value
x, y = SpatialCoordinate(mesh)
f.interpolate((1+8*pi*pi)*cos(x*pi*2)*cos(y*pi*2))
# The bilinear and linear forms
a = (inner(grad(u), grad(v)) + inner(u, v)) * dx
L = inner(f, v) * dx
u = Function(V)  # Re-define u to be the solution
# Solve the equation
solve(a == L, u, solver_parameters={'ksp_type': 'cg'})
```

Figure 32: UFL implementation of the Helmholtz equation

A.8 AoS vs SoA

Besides the storage of simple multi-dimensional data, it is often required to store multiple fields about a single object, for example, particle data. Figure 33 provides a simple example of particle storage using an array-of-structs (AoS) and a struct-of-arrays (SoA) approach.

```
#define N 1024
typedef struct {
  // position
  float x, y, x;
  // momentum
  float ux, uy, uz;
  // weight
  float w;
} Particle;
Particle particles[N];
// access x field from particle
particles[0].x;
```

```
#define N 1024
typedef struct {
  // position
  float x[N], y[N], x[N];
  // momentum
  float ux[N], uy[N], uz[N];
  // weight
  float w[N];
} Particles;
Particles particles;
// access x field from particle
particles.x[0];
```

Figure 33: AoS (left) vs SoA (right) for simple particle structure

The most intuitive way to store such data is typically using the AoS approach, but this may not be conducive to high performance on SIMD and SIMT systems. Conversely, the SoA approach may allow the cache lines to be used more effectively, but leads to less intuitive code. It may also be the case that different architectures favour different approaches; switching between AoS and SoA manually may be a significant undertaking.

A.8.1 Intel SDLT

Intel's SIMD Data Layout Templates (SDLT) offers a convenient way to abstract the in-memory data layout transparently to the developer. Figure 34 shows how this can be achieved with our previous example of particle storage. Accesses are expressed in an AoS form, but the accesses are performed through an SoA container.

A.8.2 VPIC and Kokkos

A similar approach, using Kokkos Views, can be found in the VPIC 2.0 application [33]. In VPIC 2.0, an enum is used to provide symbolic dereferencing of the fields in the structure to improve readability of the code (see Figure 35). Effectively this is implemented using a two-dimensional View that can then be stored using a row-major or a column-major layout to enable a switch between AoS and SoA.

```
#define N 1024

typedef struct particle_data {
  float x, y, x;
  float ux, uy, uz;
  float w;
} Particle;

SDLT_PRIMITIVE(Particle, x, y, z, ux, uy, uz, w)
...
sdlt::soa1d_container<Point2D> pContainer(N);
auto particles = pContainer.access();
#pragma omp simd
for (int i = 0; i < 1024; i++) {
   particles[i].x() = ...;
   ...
}
```

Figure 34: Intel SDLT

```
Kokkos::View<float*[7]> particles(N); // particle data
namespace particle_var {
  enum p_v { // particle member enum for clean access
    x, y, z,
    ux, uy, uz,
    w,
   };
};
View<int*> particle_indicies(N); // Particle indices
// Access x from particle 0
particles(0, particle_var::x) = ...;
```

Figure 35: Using Kokkos to convert AoS to SoA